

**02 INFORMATION ABOUT PRINCIPAL INVESTIGATORS/PROJECT DIRECTORS(PI/PD) and
co-PRINCIPAL INVESTIGATORS/co-PROJECT DIRECTORS**

Submit only ONE copy of this form for each PI/PD and co-PI/PD identified on the proposal. The form(s) should be attached to the original proposal as specified in GPG Section II.C.a. Submission of this information is voluntary and is not a precondition of award. This information will not be disclosed to external peer reviewers. **DO NOT INCLUDE THIS FORM WITH ANY OF THE OTHER COPIES OF YOUR PROPOSAL AS THIS MAY COMPROMISE THE CONFIDENTIALITY OF THE INFORMATION.**

PI/PD Name: John D Regehr

Gender: Male Female
Ethnicity: (Choose one response) Hispanic or Latino Not Hispanic or Latino

Race: (Select one or more)
 American Indian or Alaska Native
 Asian
 Black or African American
 Native Hawaiian or Other Pacific Islander
 White

Disability Status: (Select one or more)
 Hearing Impairment
 Visual Impairment
 Mobility/Orthopedic Impairment
 Other
 None

Citizenship: (Choose one) U.S. Citizen Permanent Resident Other non-U.S. Citizen

Check here if you do not wish to provide any or all of the above information (excluding PI/PD name):

REQUIRED: Check here if you are currently serving (or have previously served) as a PI, co-PI or PD on any federally funded project

Ethnicity Definition:

Hispanic or Latino. A person of Mexican, Puerto Rican, Cuban, South or Central American, or other Spanish culture or origin, regardless of race.

Race Definitions:

American Indian or Alaska Native. A person having origins in any of the original peoples of North and South America (including Central America), and who maintains tribal affiliation or community attachment.

Asian. A person having origins in any of the original peoples of the Far East, Southeast Asia, or the Indian subcontinent including, for example, Cambodia, China, India, Japan, Korea, Malaysia, Pakistan, the Philippine Islands, Thailand, and Vietnam.

Black or African American. A person having origins in any of the black racial groups of Africa.

Native Hawaiian or Other Pacific Islander. A person having origins in any of the original peoples of Hawaii, Guam, Samoa, or other Pacific Islands.

White. A person having origins in any of the original peoples of Europe, the Middle East, or North Africa.

WHY THIS INFORMATION IS BEING REQUESTED:

The Federal Government has a continuing commitment to monitor the operation of its review and award processes to identify and address any inequities based on gender, race, ethnicity, or disability of its proposed PIs/PDs. To gather information needed for this important task, the proposer should submit a single copy of this form for each identified PI/PD with each proposal. Submission of the requested information is voluntary and will not affect the organization's eligibility for an award. However, information not submitted will seriously undermine the statistical validity, and therefore the usefulness, of information received from others. Any individual not wishing to submit some or all the information should check the box provided for this purpose. (The exceptions are the PI/PD name and the information about prior Federal support, the last question above.)

Collection of this information is authorized by the NSF Act of 1950, as amended, 42 U.S.C. 1861, et seq. Demographic data allows NSF to gauge whether our programs and other opportunities in science and technology are fairly reaching and benefiting everyone regardless of demographic category; to ensure that those in under-represented groups have the same knowledge of and access to programs and other research and educational opportunities; and to assess involvement of international investigators in work supported by NSF. The information may be disclosed to government contractors, experts, volunteers and researchers to complete assigned work; and to other government agencies in order to coordinate and assess programs. The information may be added to the Reviewer file and used to select potential candidates to serve as peer reviewers or advisory committee members. See Systems of Records, NSF-50, "Principal Investigator/Proposal File and Associated Records", 63 Federal Register 267 (January 5, 1998), and NSF-51, "Reviewer/Proposal File and Associated Records", 63 Federal Register 268 (January 5, 1998).

List of Suggested Reviewers or Reviewers Not To Include (optional)

SUGGESTED REVIEWERS:

Not Listed

REVIEWERS NOT TO INCLUDE:

Not Listed

COVER SHEET FOR PROPOSAL TO THE NATIONAL SCIENCE FOUNDATION

PROGRAM ANNOUNCEMENT/SOLICITATION NO./CLOSING DATE/if not in response to a program announcement/solicitation enter NSF 11-1					FOR NSF USE ONLY
NSF 11-555		12/19/11			NSF PROPOSAL NUMBER
FOR CONSIDERATION BY NSF ORGANIZATION UNIT(S) (Indicate the most specific unit known, i.e. program, division, etc.)					1218022
CNS - COMPUTER SYSTEMS					
DATE RECEIVED	NUMBER OF COPIES	DIVISION ASSIGNED	FUND CODE	DUNS# (Data Universal Numbering System)	FILE LOCATION
12/19/2011	2	05050000 CNS	7354	009095365	12/30/2011 6:46pm S
EMPLOYER IDENTIFICATION NUMBER (EIN) OR TAXPAYER IDENTIFICATION NUMBER (TIN) 876000525		SHOW PREVIOUS AWARD NO. IF THIS IS <input type="checkbox"/> A RENEWAL <input type="checkbox"/> AN ACCOMPLISHMENT-BASED RENEWAL		IS THIS PROPOSAL BEING SUBMITTED TO ANOTHER FEDERAL AGENCY? YES <input type="checkbox"/> NO <input checked="" type="checkbox"/> IF YES, LIST ACRONYM(S)	
NAME OF ORGANIZATION TO WHICH AWARD SHOULD BE MADE University of Utah			ADDRESS OF AWARDEE ORGANIZATION, INCLUDING 9 DIGIT ZIP CODE University of Utah 75 South 2000 East Salt Lake City, UT. 841128930		
AWARDEE ORGANIZATION CODE (IF KNOWN) 0036756000					
NAME OF PRIMARY PLACE OF PERF University of Utah			ADDRESS OF PRIMARY PLACE OF PERF, INCLUDING 9 DIGIT ZIP CODE University of Utah 201 PRESIDENTS CIRCLE ROOM 201 Salt Lake City ,UT ,841128930 ,US.		
IS AWARDEE ORGANIZATION (Check All That Apply) (See GPG II.C For Definitions)		<input type="checkbox"/> SMALL BUSINESS	<input type="checkbox"/> MINORITY BUSINESS	<input type="checkbox"/> IF THIS IS A PRELIMINARY PROPOSAL THEN CHECK HERE	
		<input type="checkbox"/> FOR-PROFIT ORGANIZATION	<input type="checkbox"/> WOMAN-OWNED BUSINESS		
TITLE OF PROPOSED PROJECT CSR: Small: Beating Implementations of C++11 Concurrency Into Shape					
REQUESTED AMOUNT \$ 467,740	PROPOSED DURATION (1-60 MONTHS) 36 months	REQUESTED STARTING DATE 07/01/12	SHOW RELATED PRELIMINARY PROPOSAL NO. IF APPLICABLE		
CHECK APPROPRIATE BOX(ES) IF THIS PROPOSAL INCLUDES ANY OF THE ITEMS LISTED BELOW					
<input type="checkbox"/> BEGINNING INVESTIGATOR (GPG I.G.2)		<input type="checkbox"/> HUMAN SUBJECTS (GPG II.D.7) Human Subjects Assurance Number _____			
<input type="checkbox"/> DISCLOSURE OF LOBBYING ACTIVITIES (GPG II.C.1.e)		Exemption Subsection _____ or IRB App. Date _____			
<input type="checkbox"/> PROPRIETARY & PRIVILEGED INFORMATION (GPG I.D., II.C.1.d)		<input type="checkbox"/> INTERNATIONAL COOPERATIVE ACTIVITIES: COUNTRY/COUNTRIES INVOLVED (GPG II.C.2.j)			
<input type="checkbox"/> HISTORIC PLACES (GPG II.C.2.j)		_____			
<input type="checkbox"/> EAGER* (GPG II.D.2) <input type="checkbox"/> RAPID** (GPG II.D.1)		<input type="checkbox"/> HIGH RESOLUTION GRAPHICS/OTHER GRAPHICS WHERE EXACT COLOR REPRESENTATION IS REQUIRED FOR PROPER INTERPRETATION (GPG I.G.1)			
<input type="checkbox"/> VERTEBRATE ANIMALS (GPG II.D.6) IACUC App. Date _____ PHS Animal Welfare Assurance Number _____					
PI/PD DEPARTMENT School of Computing		PI/PD POSTAL ADDRESS 50 S. Central Campus Dr. Room 3190			
PI/PD FAX NUMBER 801-581-5843		Salt Lake City, UT 84112 United States			
NAMES (TYPED)	High Degree	Yr of Degree	Telephone Number	Electronic Mail Address	
John D Regehr	PhD	2001	801-581-4280	regehr@cs.utah.edu	
CO-PI/PD					

CERTIFICATION PAGE

Certification for Authorized Organizational Representative or Individual Applicant:

By signing and submitting this proposal, the Authorized Organizational Representative or Individual Applicant is: (1) certifying that statements made herein are true and complete to the best of his/her knowledge; and (2) agreeing to accept the obligation to comply with NSF award terms and conditions if an award is made as a result of this application. Further, the applicant is hereby providing certifications regarding debarment and suspension, drug-free workplace, lobbying activities (see below), responsible conduct of research, nondiscrimination, and flood hazard insurance (when applicable) as set forth in the NSF Proposal & Award Policies & Procedures Guide, Part I: the Grant Proposal Guide (GPG) (NSF 11-1). Willful provision of false information in this application and its supporting documents or in reports required under an ensuing award is a criminal offense (U. S. Code, Title 18, Section 1001).

Conflict of Interest Certification

In addition, if the applicant institution employs more than fifty persons, by electronically signing the NSF Proposal Cover Sheet, the Authorized Organizational Representative of the applicant institution is certifying that the institution has implemented a written and enforced conflict of interest policy that is consistent with the provisions of the NSF Proposal & Award Policies & Procedures Guide, Part II, Award & Administration Guide (AAG) Chapter IV.A; that to the best of his/her knowledge, all financial disclosures required by that conflict of interest policy have been made; and that all identified conflicts of interest will have been satisfactorily managed, reduced or eliminated prior to the institution's expenditure of any funds under the award, in accordance with the institution's conflict of interest policy. Conflicts which cannot be satisfactorily managed, reduced or eliminated must be disclosed to NSF.

Drug Free Work Place Certification

By electronically signing the NSF Proposal Cover Sheet, the Authorized Organizational Representative or Individual Applicant is providing the Drug Free Work Place Certification contained in Exhibit II-3 of the Grant Proposal Guide.

Debarment and Suspension Certification (If answer "yes", please provide explanation.)

Is the organization or its principals presently debarred, suspended, proposed for debarment, declared ineligible, or voluntarily excluded from covered transactions by any Federal department or agency?

Yes

No

By electronically signing the NSF Proposal Cover Sheet, the Authorized Organizational Representative or Individual Applicant is providing the Debarment and Suspension Certification contained in Exhibit II-4 of the Grant Proposal Guide.

Certification Regarding Lobbying

The following certification is required for an award of a Federal contract, grant, or cooperative agreement exceeding \$100,000 and for an award of a Federal loan or a commitment providing for the United States to insure or guarantee a loan exceeding \$150,000.

Certification for Contracts, Grants, Loans and Cooperative Agreements

The undersigned certifies, to the best of his or her knowledge and belief, that:

- (1) No federal appropriated funds have been paid or will be paid, by or on behalf of the undersigned, to any person for influencing or attempting to influence an officer or employee of any agency, a Member of Congress, an officer or employee of Congress, or an employee of a Member of Congress in connection with the awarding of any federal contract, the making of any Federal grant, the making of any Federal loan, the entering into of any cooperative agreement, and the extension, continuation, renewal, amendment, or modification of any Federal contract, grant, loan, or cooperative agreement.
- (2) If any funds other than Federal appropriated funds have been paid or will be paid to any person for influencing or attempting to influence an officer or employee of any agency, a Member of Congress, an officer or employee of Congress, or an employee of a Member of Congress in connection with this Federal contract, grant, loan, or cooperative agreement, the undersigned shall complete and submit Standard Form-LLL, "Disclosure of Lobbying Activities," in accordance with its instructions.
- (3) The undersigned shall require that the language of this certification be included in the award documents for all subawards at all tiers including subcontracts, subgrants, and contracts under grants, loans, and cooperative agreements and that all subrecipients shall certify and disclose accordingly.

This certification is a material representation of fact upon which reliance was placed when this transaction was made or entered into. Submission of this certification is a prerequisite for making or entering into this transaction imposed by section 1352, Title 31, U.S. Code. Any person who fails to file the required certification shall be subject to a civil penalty of not less than \$10,000 and not more than \$100,000 for each such failure.

Certification Regarding Nondiscrimination

By electronically signing the NSF Proposal Cover Sheet, the Authorized Organizational Representative is providing the Certification Regarding Nondiscrimination contained in Exhibit II-6 of the Grant Proposal Guide.

Certification Regarding Flood Hazard Insurance

Two sections of the National Flood Insurance Act of 1968 (42 USC §4012a and §4106) bar Federal agencies from giving financial assistance for acquisition or construction purposes in any area identified by the Federal Emergency Management Agency (FEMA) as having special flood hazards unless the:

- (1) community in which that area is located participates in the national flood insurance program; and
- (2) building (and any related equipment) is covered by adequate flood insurance.

By electronically signing the NSF Proposal Cover Sheet, the Authorized Organizational Representative or Individual Applicant located in FEMA-designated special flood hazard areas is certifying that adequate flood insurance has been or will be obtained in the following situations:

- (1) for NSF grants for the construction of a building or facility, regardless of the dollar amount of the grant; and
- (2) for other NSF Grants when more than \$25,000 has been budgeted in the proposal for repair, alteration or improvement (construction) of a building or facility.

Certification Regarding Responsible Conduct of Research (RCR)

(This certification is not applicable to proposals for conferences, symposia, and workshops.)

By electronically signing the NSF Proposal Cover Sheet, the Authorized Organizational Representative of the applicant institution is certifying that, in accordance with the NSF Proposal & Award Policies & Procedures Guide, Part II, Award & Administration Guide (AAG) Chapter IV.B., the institution has a plan in place to provide appropriate training and oversight in the responsible and ethical conduct of research to undergraduates, graduate students and postdoctoral researchers who will be supported by NSF to conduct research. The undersigned shall require that the language of this certification be included in any award documents for all subawards at all tiers.

AUTHORIZED ORGANIZATIONAL REPRESENTATIVE		SIGNATURE	DATE
NAME LAUREL DUNCAN		Electronic Signature	Dec 19 2011 12:30PM
TELEPHONE NUMBER 801-581-3006	ELECTRONIC MAIL ADDRESS laurel.duncan@osp.utah.edu	FAX NUMBER 801-581-3007	

* EAGER - EARly-concept Grants for Exploratory Research
 ** RAPID - Grants for Rapid Response Research

Project Summary — CSR: Small: Beating Implementations of C++11 Concurrency Into Shape

The recently approved version of the C++ standard, “C++11,” adds a concurrency model that will make it possible to write high-performance, portable code for multiprocessors. The concurrency model is large and complicated; it will not be particularly easy for compiler and library developers to get all of its corner cases right. In particular, the concurrency model interacts in subtle ways with existing compiler optimizations and also with difficult elements of the host platform such as the operating system’s thread implementation and the processor’s memory coherence mechanisms.

Proposed Research The PI proposes developing techniques to stress-test implementations of the C++11 concurrency model. This will be done by extending Csmith—the PI’s randomized test case generator that has been responsible for finding more than 400 previously unknown compiler bugs—to generate code using the C++11 memory model. Since C and C++ compilers have already been shown to incorrectly implement the `volatile` type qualifier, which serves as a trivial memory model for accessing device registers, it is strongly expected that the far more complex C++11 memory model will also be a fruitful source of bugs.

Intellectual Merit The C++11 standard fails to specify a semantics for programs that contain *data races*—insufficiently synchronized accesses to shared data items. Therefore, it will be necessary to extend Csmith (which currently only generates sequential code) to generate race-free concurrent programs using the C++11 primitives. The second research problem that must be solved is generating concurrent code with interesting *invariants*—testable properties that are guaranteed to hold across all possible executions. Third, the PI will develop one or more “hostile” execution environments that arrange for unfavorable (but permissible) activity by the memory system and thread scheduler. Together, these elements will significantly advance the state of the art in testing a very difficult aspect of compiler and runtime system implementation.

Broader Impacts The proposed work, if successful, will improve the correctness of C++11 implementations. This is important because C++ is the language of choice for many large, concurrent, system-level codes such as web browsers and Internet servers. Moreover, the C++11 memory model is likely to be incorporated into the next C standard. Of course, many important concurrent codes such as operating systems, virtual machine managers, programming language runtimes, and critical embedded systems are written in C.

Now is the time for research that will help compiler developers create robust C++11 implementations. The standard has been approved by the ISO and at least a half-dozen serious efforts are underway to implement the new standard. These implementations will rise to usable quality significantly more rapidly if aggressive stress-testing tools can be used to find bugs. The proposed tools will be open source; they will directly benefit any C++11 compiler development effort. Furthermore, the proposed research will indirectly benefit C++11 developers who use concurrency and also the users of the programs that they develop. The alternative to a stress-testing tool such as the one that will emerge from the proposed work is manual debugging of miscompiled concurrent C++ programs—a costly and difficult activity.

Key Words: compiler correctness; C++ memory model; random testing.

TABLE OF CONTENTS

For font size and page formatting specifications, see GPG section II.B.2.

	Total No. of Pages	Page No.* (Optional)*
Cover Sheet for Proposal to the National Science Foundation		
Project Summary (not to exceed 1 page)	1	_____
Table of Contents	1	_____
Project Description (Including Results from Prior NSF Support) (not to exceed 15 pages) (Exceed only if allowed by a specific program announcement/solicitation or if approved in advance by the appropriate NSF Assistant Director or designee)	15	_____
References Cited	3	_____
Biographical Sketches (Not to exceed 2 pages each)	2	_____
Budget (Plus up to 3 pages of budget justification)	5	_____
Current and Pending Support	2	_____
Facilities, Equipment and Other Resources	1	_____
Special Information/Supplementary Documents (Data Management Plan, Mentoring Plan and Other Supplementary Documents)	2	_____
Appendix (List below.) (Include only if allowed by a specific program announcement/ solicitation or if approved in advance by the appropriate NSF Assistant Director or designee)	_____	_____
Appendix Items:		

*Proposers may select any numbering mechanism for the proposal. The entire proposal however, must be paginated. Complete both columns only if the proposal is numbered consecutively.

Project Description — CSR: Small: Beating Implementations of C++11 Concurrency Into Shape

1 Motivation and Overview of the Proposed Work

Many large and important concurrent software systems are written partly or entirely in C++98:^{1,2}

- Apple’s Mac OS X
- Adobe Illustrator
- Facebook
- Google’s Chrome browser
- Microsoft Windows 7 and Internet Explorer
- Firefox
- MySQL

Similarly, many large and important concurrent software systems are implemented in C, including almost every operating system kernel and hypervisor, most Java virtual machines and many other programming language runtimes, and many embedded systems including those that control critical transportation and medical care systems.

It is ironic, then, that these languages—in which massive amounts of mission- and safety-critical concurrent code are written—have no built-in support for concurrent execution. This omission can cause problems for software development efforts. First, the compiler—not designed for concurrency—can introduce race conditions into programs that appear correct at the source level (Section 4 contains an example). Second, consider this code which implements a two-process mutual exclusion scheme using the well-known Dekker algorithm:

```
volatile int flag[2], turn;

void dekker_unlock (int i) {
    turn = 1-i;
    flag[i] = 0;
}

void dekker_lock (int i) {
    flag[i] = 1;
    while (flag[1-i]) {
        if (turn != i) {
            flag[i] = 0;
            while (turn != i) { }
            flag[i] = 1;
        }
    }
}
```

¹“C++98” refers to the 1998 version of the C++ standard. It is the dialect in which nearly all contemporary C++ code is written. The recently approved C++11 standard has not yet been entirely implemented by any compiler we are aware of.

²http://www.theregister.co.uk/2011/06/11/herb_sutter_next_c_plus_plus/

This code can be proved to be correct (that is, it provides progress, mutual exclusion, and bounded wait) as long as operations on volatile global variables are sequentially consistent.³ Therefore, this code works on uniprocessor machines (of course, a spinlock is not efficient on a uniprocessor, but at least it is correct). On the other hand, *on most multiprocessor machines this Dekker implementation is incorrect: it fails to provide mutual exclusion*. This is even the case on machines with a relatively strong memory model such as multicore Intel and AMD processors implementing the x86 and x86-64 architectures.

To fix the Dekker code, processor-specific *memory fence* operations must be added. Fences enforce ordering constraints on memory operations that would otherwise be reordered by hardware. Because memory system and fence semantics vary, code with fences is not portable, and also it can be extremely difficult to figure out where fences are required. If one errs on the side of adding too many fences, performance will suffer. If one errs on the side of adding too few, the Dekker code will fail to properly implement mutual exclusion.

Of course, the vast majority of programmers are not interested in writing platform-specific code for a weak memory model. The alternative is to target a specific concurrency library such as POSIX threads or Win32 threads. However, libraries are typically not universally available; POSIX threads are generally not found on non-UNIX platforms and Win32 is not found outside of Windows machines. Additionally, libraries such as POSIX threads fail to provide direct access to low-level hardware operations such as compare-and-swap instructions, preventing certain high-performance idioms from being expressed portably.

1.1 The C++11 Memory Model

The recently-adopted C++11 standard was designed to fix the problems discussed in the previous subsection. First, it standardizes C++ concurrency across platforms. The standard formalizes the notion of *data race* and prohibits compilers from introducing races. Second, it provides an extensive suite of powerful, low-level operations that are intended to make it possible for portable code to attain very high performance. It is believed that the next version of the C standard (the current draft is referred to as “C1X”) will incorporate a memory model very similar to the C++11 concurrent memory model.

1.2 Problem: Correctness of Implementations of the C++11 Memory Model

The problem that the proposed work aims to solve is:

1. The C++11 memory model is large and complicated, containing dozens of classes and templates, and hundreds of functions. These interact closely with OS-specific synchronization calls and hardware-specific synchronization instructions.
2. Today’s C++98 compilers will serve as the basis for all known C++11 implementations. These tools—which are large, complicated, and contain highly aggressive optimizations—were developed in the absence of any non-trivial memory model.
3. Compiler optimizations and the C++11 memory model are intimately related. For example, some transformations that were previously permissible are now illegal. Although the full

³According to Lamport [24], sequential consistency requires that “the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.”

extent of the interactions between existing compiler implementations and the new memory model is not yet totally understood, it is clear that a lot of work will be required to iron out the details. Subtle bugs are likely to persist for some time.

We have talked to compiler developers who work on LLVM, GCC, and on several important commercial compilers. None of these developers expects that implementation of the C++11 memory model will be quick or easy. In fact, they all expressed concern about interactions between existing optimization passes and constraints imposed by the new model.

1.3 Intellectual Merit — Major Research Challenges and Solution Sketch

In a nutshell, the proposed work is to randomly generate concurrent C++11 programs, compile them, and then run them on an infrastructure designed to reveal concurrency errors. This is a logical extension of our previous work that has resulted in more than 400 previously unknown compiler bugs being identified, reported, and usually fixed. However, there are several significant challenges that must be overcome.

Challenge #1: Generating Valid Code The major challenge in creating Csmith was to generate code that is *expressive* and *well-defined*. Expressive code exercises a large subset of language features, and well-defined code avoids executing any of the 191 *undefined behaviors* from the C standard (integer overflow, array and pointer errors, union errors, divide by zero, shift past bitwidth, etc.). In C++11, *data races* are an undefined behavior. The definition of data race is intricate, but the core idea is to determine if every access to each shared data item is sufficiently synchronized with respect to all other accesses to that item. Generating random, race-free programs is expected to be significantly non-trivial in the presence of arrays and pointers. Our solution will exploit the dataflow analysis that is already a part of Csmith. Essentially, we will implement a static race checker that runs in tandem with program generation.

Challenge #2: Generating Interesting Invariants Automated random testing can succeed only when useful invariants are identified in the system under test. Our previous work has exploited three invariants. First, our testing of the volatile qualifier made use of the *volatile invariant*, which is implied by the C and C++ standards. It specifies that the number of load and store operations issued to each byte of a volatile-qualified object must not change across compilers or optimization options. Second, our wrong-code bug hunting using Csmith relied on the invariant that for a well-defined test program, a checksum taken over the global variables at the end of execution must not change across compilers or optimization options. Third, randomized compiler testing relies on the simple invariant that the compiler should not crash when presented with valid input.

Randomly testing implementations of the C++ memory model will only be possible if we can generate code with interesting concurrent invariants. Of course, concurrent programs written by humans have invariants (e.g., “the `empty` flag is set iff the list is empty”). We plan to use ideas from *invariant-based program synthesis* [16] to create random programs with non-trivial invariants; Section 5 explores the proposed approach in more detail. A useful consequence of the invariant-based approach to compiler testing is that Csmith’s test cases will become *self-checking*—compiler bugs will cause them to fail with an assertion violation. In contrast, our compiler testing work up to this point has used *differential testing* where multiple compilers (or multiple optimization levels of a single compiler) were tested against each other.

Challenge #3: Exploring the Scheduling Space The space of thread interleavings is large, and finding bugs in concurrent programs boils down to exploring as much of this space as possible. Our compiler stress testing work will explore three approaches. First, we hypothesize that early progress can be made using simple mechanisms to introduce scheduling noise, for example by introducing randomized `nop` instructions or sleep calls into a compiled program using binary rewriting, and by using a second machine to send randomized network interrupts to the machine that is executing test cases. Second, there is much existing work on designing runtime systems that attempt to make concurrency errors show themselves; we will reuse this work to the maximum extent possible. Third, we will explore the idea of implementing a hostile x86-64 simulator. Although we believe that it is probably not feasible to compute a pessimal schedule for a program, hostility is an attainable goal. It will be implemented using techniques such as triggering context switches near and inside (inferred) critical sections and delaying inter-processor communication as long as possible, in effect replacing the CPU’s write buffer with a “wrong buffer.”⁴

The thesis of this proposal:

Randomized stress testing of implementations of the C++11 memory model will significantly shorten the period of time during which compilers are flaky and immature. In the long run, randomized stress testing will lead to a better understanding of corner-case behaviors of the C++11 memory model and to test suites with improved coverage of interactions between the C++11 memory model, compiler optimizers, OS thread systems, and processor memory coherence mechanisms. The intellectual work required to produce concurrent test cases is a major increment upon our existing work on randomized test case generation which has had great success in finding compiler bugs, but which is limited to sequential code.

1.4 The State of the Art is Inadequate

Existing methods for testing compiler implementations break down into three broad methods:

1. Compiling large, real applications. For example, a significant milestone in the development of any compiler is when it becomes self-hosting (can compile itself).
2. Compiling collections of small, troublesome codes (e.g., GCC’s “torture test,” which contains about 2800 programs).
3. Randomized testing. Our Csmith project is one example, but the history of this technique goes back about 50 years.

All three methods are useful and necessary. Compiler developers we have talked to, who are working on C++11 implementations, plan to use methods 1 and 2. We do not believe these methods will be sufficient to rapidly squash all of the relevant bugs (nor do the compiler developers appear to believe this). Consequently, we claim the proposed work is useful and important.

⁴The term “wrong buffer” appears to have originated with system developers at ARM; we know of no appearances of this term in print.

1.5 Why Testing Instead of Verification?

CompCert [25] is an existence proof for a verified, usable, optimizing compiler for a realistic programming language. One might ask: Why should a testing proposal be funded? Why not be more ambitious and create a proved-correct C++ compiler? There are several reasons:

- Testing is *end-to-end*, whereas formal verification typically addresses a narrow layer of the software stack. For example, in addition to finding compiler bugs, our compiler testing work has found linker bugs, assembler bugs, runtime library bugs, CPU simulator bugs, and even bugs caused by the fact that the compiler had itself been miscompiled. Formal verification cannot find these errors without specifically verifying the linker, assembler, runtime library, simulator, and the compiler that compiled the compiler under test. While these are all laudable goals, the combined effort to do these verifications is gigantic.
- Even verified compilers require testing. For example, we have discovered and reported 13 CompCert bugs. The core issue is that creating a machine-checked mathematical proof is one thing, but *proving the right thing* is a separate and very hard problem. Typically it is not possible to once and for all “prove that one has proved the right thing.”
- The C++11 language is dramatically more complex than C. CompCert does not even compile all of C, but rather only a useful subset. Creating a verified, optimizing compiler for a broadly useful subset of C++11 would be a massive undertaking, requiring (we guess) more than a few person-decades of effort.

Summary: A verified C++11 compiler is out of reach for now. But even if we had this compiler, we would still need to stress test it.

1.6 Specific Outcomes

The proposed work, if successful, will have several useful outcomes. First, and most obviously, we will create tools for stress-testing implementations of the C++11 concurrency model. *These tools will be open-source, and consequently will benefit every team that is working on a C++11 implementation* (in addition to transitively benefiting users of those compilers and users of systems built by those compilers). Second, as a side effect of developing and validating our tools, we will report bugs against open source compilers (GCC and LLVM, in particular) and also we will work with any commercial compiler vendors who seek us out.⁵ Third, we expect to be able to produce useful “litmus tests”—small test cases that are (empirically) difficult for compilers to translate correctly and that, taken as a group, provide good test coverage of interesting parts of the C++11 concurrency model. Finally, we hope to generate a better understanding of the C++11 concurrency model. By analogy, our Csmith work has (completely automatically) exposed some interesting corner cases in the C programming language, as described here: <http://blog.regehr.org/archives/558>

⁵Although we have built solid relationships with the GCC and LLVM teams, so far we have not had very good luck in building relationships with companies that produce compilers. The problem (we think) is that we represent a poor short-term value proposition: we generate zero revenue while producing bug reports that suck up valuable engineering resources. Furthermore, being outsiders, we would seem to represent the potential for causing only bad publicity—to paraphrase Dijkstra, we can find bugs but we cannot certify their absence.

2 The PI’s Prior Work on Compiler Testing

This section briefly describes our work leading up to this proposal. It also explains why we are virtually certain that the proposed work will find bugs in production-quality compilers.

Although the C and C++98 languages do not have a notion of concurrent execution, they do have a simple memory model that is designed to support reliable access to memory-mapped hardware device registers. This model is provided by the `volatile` qualifier.⁶ A volatile-qualified object must be accessed at the memory system level as it is by the C/C++ abstract machine. In other words, a read from a volatile-qualified variable in C/C++ must result in a load being issued at the machine level—the variable’s value must not be cached in a register. Similarly, a write to a volatile variable in C/C++ must result in a store at the machine level—the compiler must not suppress redundant stores, as an optimizer normally would. The volatile qualifier’s semantics in C++11 are the same as they are in C and C++98.

The PI, while teaching embedded systems courses in the early/mid 2000s, noticed that the compilers being used by students did not always respect the volatile qualifier. This was troubling. In joint work with Eric Eide, he developed a way to test compilers’ implementation of the volatile qualifier. For every compiler we tested—including commercial tools used to compile safety-critical embedded systems—we found errors in its treatment of volatile-qualified variables. Some of these results are reported in our 2008 paper [17], but since then we have tested quite a few more compilers, with the same result. In summary, no compiler that we tested was able to correctly implement the volatile qualifier, *which is basically trivial when compared to the C++11 memory model*.

Since volatile-qualified objects are not totally reliable, we developed a workaround where source code is re-written in such a way that every read from a volatile variable is changed into a call to a helper function, and every write to a volatile variable is changed into a different helper function call. We verify by hand that the helper functions perform the necessary memory operations. In a large fraction of our tests (96% of the time) this strategy was successful in working around bugs in compilers’ implementations of volatile. The workaround is successful because compilers tend to be able to perform function calls reliably, even when they cannot access volatiles reliably. Subsequently, we became interested in the 4% of cases where our workaround failed. We hypothesized that these cases were due to “regular” compiler bugs having nothing to do with the volatile qualifier. We began systematically hunting for these bugs.

Since 2008 the PI’s group has found and reported about 400 bugs in widely-used C compilers. These bugs were all found using Csmith [34], a random C program generator that is a (distant) descendant of our volatile testing tool. Most of the bugs were in the compilers’ *middle-ends*—the part of a compiler that is independent of both the language being compiled and the architecture being targeted. Despite the fact that most of these 400 bugs have been fixed, we can still crash and find wrong-code bugs in the latest version of all compilers that we routinely test (about six). CompCert [25] is the lone exception. Csmith is open-source software.⁷

⁶The rationale for `volatile` is described here: <http://groups.google.com/group/comp.std.c/msg/7709e4162620f2cd>

⁷<http://embed.cs.utah.edu/csmith/>

3 Brief Introduction to the C++11 Memory Model

Although the C++11 standard cannot be freely downloaded, a recent draft can be.⁸ Material about the concurrency model is concentrated in Chapters 1, 29, and 30. However, a “less formal” description by Boehm [5] and a paper by Boehm and Adve [7] are probably better places to begin reading. Additionally, a book about the model is expected to be released in January 2012 [32]. On the more formal side, Batty et al. [4] have formalized the C++11 memory model (but not the rest of the language). This kind of activity is essential to understanding the most subtle parts of the model.

3.1 Application Programming Interface

The API for the C++11 concurrency model divides into two major parts. The first is a thread support library. It is object oriented and uses the C++ exception system, but otherwise is not very different from Pthreads and other existing thread libraries. C++11 threads are intended to map onto OS-supported threads in a one-to-one fashion. The threads interface is contained in four header files, respectively supporting thread manipulation, mutual exclusion, condition variables, and futures.

A welcome feature of the mutex library is a facility for safely acquiring multiple locks:

```
std::unique_lock lck1(m1, std::defer_lock);
std::unique_lock lck2(m2, std::defer_lock);
std::unique_lock lck3(m3, std::defer_lock);
lock(lck1, lck2, lck3);
// manipulate shared data ...
```

The advantage of this interface is that the developer does not have to worry about deadlocking the program by acquiring locks in the wrong order.

The second major component of the C++11 API is an atomic operations library. Atomic operations are synchronization operations that are semantically equivalent to locking an object, reading and/or writing it, and then releasing the lock. The difference is that in some cases, the atomic operation may be significantly more efficient than the explicit locking approach. For example, on x86 and x86-64 processors an atomic increment of an integer value can be compiled into a single instruction.

The most interesting feature of the C++11 atomic API is that it provides portable accesses to weak memory models. The default memory ordering for all atomic operations is sequential consistency (`memory_order_seq_cst`), meaning that the sequentially consistent operation is part of a global total ordering of memory operations. This is intuitive; for example, the Dekker code from Section 1—which is incorrect on a non-sequentially-consistent multiprocessor—can be fixed in C++11 simply by declaring the synchronization variables as atomic. That is, instead of declaring them as:

```
volatile int flag[2], turn;
```

They should be declared as:

⁸<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf>

```
std::atomic<int> flag[2], turn;
```

The atomic template replaces the default integer operations (assignment, increment, etc.) with atomic and sequentially consistent code appropriate for the target platform.

The problem with sequential consistency is that it can be expensive. To support high-performance multiprocessor programming, C++11 defines five more memory orderings: `memory_order_relaxed`, `memory_order_consume`, `memory_order_acquire`, `memory_order_release`, and `memory_order_acq_rel`.

The relaxed memory ordering is the most permissive: atomic operations using it are still atomic, but no ordering constraints are imposed across threads. This ordering is perhaps not very useful by itself, but C++11 provides explicit fence operations that can be combined with relaxed consistency to build useful and performant concurrent data structures.

Using `memory_order_acquire` and `memory_order_release`, high-performance lock acquisition and release primitives can be constructed. An acquire operation is a one-way memory fence that prevents other memory operations from being moved above the acquire; a release operation is a one-way memory fence that prevents other memory operations from being moved below the release. Together, they prevent memory operations from being moved outside of a critical section—since this would defeat the purpose of the critical section. An access with `memory_order_acq_rel` behaves as both an acquire and a release. The `memory_order_consume` is similar to `memory_order_acquire`, but weaker: only values that are dependent on the lock variable (via a dependency mechanism explained in the standard) are ordered by the consume/release sequence.

3.2 Obligations on the Developer and Compiler

The fundamental obligation placed upon a developer writing C++11 programs is that she must never permit a data race to occur. Informally, a program is free of data races if it contains adequate synchronization. In more detail, the C++11 standard says:

The execution of a program contains a *data race* if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other.

The definition of *atomic* is straightforward—operations on atomic types are always atomic. The definition of *conflicting* is also straightforward: “Two expression evaluations conflict if one of them modifies a memory location and the other one accesses or modifies the same memory location.” Alas, *happens before* is more intricate. It basically follows Lamport’s definition [23] but accounts for synchronization that occurs through weak memory-model operations such as acquires and releases.

A data race, like any other undefined behavior in C++, destroys the meaning of the program. Avoiding races is relatively straightforward (if any concurrent programming problem can be called that) as long as the developer sticks with mutexes and sequentially consistent atomics. Static and dynamic race-detection tools are a fairly mature technology and will be able to help once they are adapted to support C++11.

On the other hand, the compiler cannot limit itself to the sequentially consistent case; its obligation to faithfully implement the C++11 standard is severe. The soundness of various compiler optimizations under weak memory models is a matter of current research [1]. The next section explains some of the problems.

4 What Do C++11 Concurrency Bugs Look Like?

This section uses examples to briefly illustrate what could go wrong when a C++11 compiler translates a concurrent program. The best references about the interactions between optimizers and C++11 are by Boehm [6] and by McKenney et al. [26] (the latter document refers to a port of the C++11 memory model to C, but the discussion applies to C++ as well).

Example #1 Because C++98 and C were specified in the absence of a concurrency model, it is permissible for a compiler to introduce writes to objects that were not written to by the program. For example, consider this struct and accessor function:

```
struct foo { int a:8; int b:12; char c; } x;

void update_b (void) { x.b = 1; }
```

It is perfectly legal for a C or C++98 implementation to translate `update_b()` into code that loads the entire 32-bit word containing `x`, appropriately modifies the 12 bits corresponding to `x.b`, and stores the word back into memory. In C++11 this translation is illegal because it may introduce data races with concurrent threads that are accessing field `c`. The current development version of the GNU C++ compiler, which is in a “bug fix only” state in preparation for the 4.7.0 release, miscompiles this code even if it is given compiler flags prohibiting it from introducing data races.⁹ On x86-64 the output is:

```
_Z8update_bv:
    movl  x(%rip), %eax
    andl  $-1048321, %eax
    orb  $1, %ah
    movl  %eax, x(%rip)
    ret
```

This code reads the entire struct into a register, uses a bitwise-AND and then a bitwise-OR instruction to modify the bits corresponding to `b`, and then stores the result back to memory. This is wrong in C++11.

Example #2 McKenney et al. [26] provide an illustrative example concerning constant propagation: one of the simplest, most useful compiler optimizations. Given that `x` is an atomic variable and it is known to be either 0 or 1, how might this code be optimized?

```
if (x.load(memory_order_consume)) {
    ...
} else {
    ...
}
y = 42 * x / 13;
```

⁹These flags are discussed in the GCC Wiki: <http://gcc.gnu.org/wiki/Atomic/GCCMM/ExecutiveSummary>

On platforms with expensive multiplication and division, the compiler would like to transform this code to:

```
if (x.load(memory_order_consume)) {
    ...
    y = 3;
} else {
    ...
    y = 0;
}
```

However, McKenney et al. state that “If the underlying machine preserves control-dependency ordering for writes, this optimization is perfectly legal. If the underlying machine does not preserve control-dependency ordering, then either this optimization must be avoided, a memory fence must be emitted after the load of `x`, or an artificial data dependency must be manufactured.” If compiler writers miss this subtlety, the generated code will either be suboptimal or wrong.

5 Details of the Proposed Approach

To recap, the proposed stress-testing work flow is to repeatedly (1) generate a random concurrent C++11 program, (2) compile it using a variety of possibly-buggy C++11 compilers and optimization flags, and (3) run these programs in hostile execution environments that are designed to expose interesting interleavings at the memory system level. We are *not* proposing to support all of C++11 in Csmith—that needs to be done, but it is well beyond the scope of this proposal. Rather, Csmith will generate C++11 that looks like C code with C++11 concurrency constructs added. We expect this to be sufficient to find many, but not all, bugs in C++11 concurrency implementations.

5.1 Generating Data-Race-Free Programs

Csmith already includes sophisticated machinery for generating programs that avoid undefined behaviors. Our PLDI 2011 paper [34] explains this in detail, but the essence of the solution is to interleave static analysis with program generation. First, Csmith generates a statement (e.g., `x = *y;`) that is a candidate for addition to the current program. Second, Csmith performs a number of static, dataflow-driven *safety checks* on the candidate statement in the current program context. In this example, `x` and `*y` must be compatible types and `y` must not be null or refer to an out-of-scope memory location. If the safety checks pass, the candidate is committed to the current program; otherwise, the candidate is discarded. Progress is probabilistically guaranteed because Csmith’s random choices always include at least one “easy” alternative that never fails the safety check. Loops and function calls complicate this simple picture, but the basic structure of the solution, which we call “guess and check,” applies.

We will augment Csmith’s existing dataflow analyses to support static data race detection. Plenty of good starting points are available [11, 18, 19, 21, 27, 30, 31]. Innovation may be required to perform race checking in the presence of atomics using C++11’s weak memory orderings. We do not know of a tool that checks for races in weakly ordered C++11, though it seems very likely that these will appear during the next year or two.

Given a race checker in Csmith, we believe that the guess and check approach will be sufficient to generate race-free programs. Even so, creating “good” random programs is as much an art as a science and we expect that additional refinements will be useful. First, accesses to sequentially consistent, atomic variables are always safe—the race detector does not need to be consulted before committing an access to this kind of memory. Second, Csmith’s safety checks can be simplified for data that is not shared between threads. We believe it will be useful to pre-partition variables into unshared ones that can be freely accessed by a single thread, and shared variables that must be synchronized appropriately. We expect unshared data to play an important role in compiler bug detection; for example, unshared data can expose compiler-introduced races like the one in Section 4. Also, code accessing unshared data can trigger optimization passes that may mangle nearby shared data accesses.

5.2 Generating Programs with Concurrent Invariants

Currently, just before terminating, a program generated by Csmith takes a checksum of the values held in all global variables and prints it. The checksum is invariant across all platforms making compatible choices for certain implementation-defined parameters (integer width and representation, mainly). Thus, a checksum difference indicates a compiler bug.

In general, the checksum approach will not work for concurrent programs. One approach would be to generate deterministic concurrent programs—where the checksum will again be invariant—but this seems quite restrictive, and will serve as a backup plan in case other proposed approaches do not work. Csmith currently does not track the values of scalar variables; its dataflow analysis will need to be extended to do this. We previously developed cXprop [12, 14, 15], a value-tracking abstract interpreter for concurrent C code; we do not expect it to be difficult to either repurpose cXprop or else re-implement the same techniques within Csmith. cXprop already knows how to generate assertions corresponding to the invariants that it computes. While we were debugging cXprop, an assertion violation generally signaled a cXprop bug—but of course these assertions can also catch compiler bugs. To ensure that invariants are not violated, Csmith will synthesize a separate thread that asserts that every invariant holds, in an infinite loop.

Thus far, we have discussed using the guess-and-check approach, combined with a concurrency-aware dataflow analysis, to generate concurrent code with invariants. We will also explore an alternative approach where invariants are the starting point for program synthesis, rather than a side effect of their construction. We will exploit a technique developed by Deng et al. [16] for *invariant-based synthesis* of concurrent programs. This approach shows how to start with a coarse-grained synchronization solution and some invariants, and automatically refine it into a fine-grained solution. Our approach will be very similar: we will begin with randomized invariants and coarse-grained code (implemented using thread-level locks). Then, we will randomly apply transformations that make the solution more fine-grained, while preserving the invariants. First, critical sections will be split into smaller critical sections, then they will be replaced (when possible) with sequentially-consistent operations on atomic types, and finally we will attempt to introduce relaxed memory-model operations and memory barriers.

5.3 Finding Invariant Violations

Given an executable test case containing assertions, we need a way to determine if it was compiled correctly. The easiest (but least reliable) way to do this is to run the program and wait for an

assertion violation to occur. This is not reliable because it is typically the case that running a concurrent program explores only a vanishingly small part of the space of all possible interleavings among memory operations in its threads. The hardest (but most reliable) way to find invariant violations is to run the compiled code through a formal methods-based tool that finds an execution where an invariant is violated, or else proves that such does not exist [8,20]. Although a number of research efforts have attacked the problem of program verification on relaxed memory models, all such tools that we are aware of do not scale to even medium-sized codes. The problems that must be solved to make these tools scale are extremely difficult. Thus, while we plan to take advantage of any such tools that become available, we also plan to explore alternatives.

Executing on hardware It is not difficult to perturb a program’s schedule in order to increase the probability that interesting interleavings are seen. One way is to use binary rewriting to insert randomized `nop` instructions into the compiled code. The inserted code can also use a random number generator so that delays are parameterized by a seed value. Another method is to cause interrupt handlers to fire on the machine that is doing the testing. This is easy to arrange by having a second machine send it small packets over a network link at short, random intervals. This method has the further benefit of causing substantial memory traffic due to DMA operations initiated by the network interface card.

Alglave et al.’s Litmus tool [2] is based on similar ideas: it randomizes memory locations accessed by threads and also the threads’ CPU affinities. We will explore these techniques in addition to the ones described above. Although there are certainly limits on the effectiveness of techniques like these, we believe they are very much worth trying simply because they are low-cost, easily portable across architectures, and absolutely reliable in the sense that any behavior that is observed is real—there are no false alarms when executing on real hardware.

Yet another approach to executing test cases on real hardware is being explored by the GCC developers.¹⁰ They plan to execute the code under test in one thread and an assertion checker in a separate thread (as we propose above). They will to single-step through the thread under test (in an automated fashion, using debugger support) running the full assertion check after every instruction. This will ensure that every intermediate memory state is checked. Drawbacks of this approach are that it will incur a significant slowdown and also the act of single-stepping the processor will flush the processor’s pipeline and store barrier—this could easily mask bugs.

Executing in a simulator Running test codes in a simulator reduces testing throughput, but has other advantages such as being repeatable and making it possible to simulate platforms that are unavailable or inconvenient. Although we do not know of an open-source simulator for ARM, x86, or x86-64 that faithfully implements the relevant memory model, it is possible that one exists or will exist within the next year or two. If not, we do not believe it will be too difficult to add weak memory model support to an existing simulator.

Given a weak memory model simulator, the challenge is to make it actively hostile to the program under test. One approach would be to use state-space exploration to look at many possible interleavings of memory operations. However, we believe this will be too slow to test realistic programs. Rather, it should be possible to focus the simulator’s efforts on weak points in the code under test. By analogy with race-directed software testing [29], we will explore the idea of causing the simulator to delay a thread until another processor is ready to execute code that touches

¹⁰<http://gcc.gnu.org/wiki/Atomic/GCCMM/AtomicTesting>

conflicting memory locations. Similarly, there is significant room for hostility in the implementation of the write buffer, which should act as a “wrong buffer” by delaying flushes to main memory for as long as possible.

5.4 Test Case Reduction

To keep the story simple, we have not yet mentioned one problem that we will need to work on: *test case reduction*—the process of turning a large failure-inducing program into a much smaller one. Reduction is an important part of root-cause analysis for any compiler bug, and in practice it needs to be automated. Although the results are not yet published, automated test case reduction has been an active area of research for us for several years—but only for the case of sequential code.

Automated test case reduction for concurrent codes is more difficult because automated reduction algorithms—generally based on Delta Debugging [35]—require deterministic execution. For example, Choi and Zeller’s work on isolating failure-inducing thread schedules [10] fundamentally depends on a capture/replay mechanism that makes threads execute deterministically. It is not clear how to avoid the need for determinism. It is also not clear how to get efficient deterministic replay on a weak-memory multicore; there are research solutions addressing this problem but they generally require hardware support or are very inefficient; Heydari and Azimi survey this work [22]. In summary, replay on real multiprocessors is effectively a separate research problem and we do not plan to address it.

Determinism in a simulator is trivial. Even so, test case reduction remains challenging. The fundamental difficulty in reducing the size of failure-inducing C/C++ programs is avoiding undefined behaviors. For example, in our experience it is common for a test case reducer to delete the initializer for a variable, resulting in a test case with undefined semantics. When reducing concurrent C++11 programs the problems are even worse because we must not introduce a data race during reduction.

Two solutions to avoiding undefined behaviors in general, and race conditions in particular, suggest themselves. First, we can use an external race detector to validate test cases during reduction, if one becomes available. Second, since Csmith will already know how to generate race-free (and otherwise well-defined) programs, we can leverage it to also generate smaller versions of the programs. We have already experimented with this approach for reducing sequential test cases in the C language, and it appears to be workable.

6 Broader Impacts

Blog Impact The PI is an active blogger and regularly posts technical material relating to compiler bugs, random testing, and related topics.¹¹ A blog appears to be an excellent way to reach an audience that is interested in CS research issues but who that does not actively read research papers. So far in 2011, the PI’s blog received more than 315,000 page views.

Educational Impact Alas, since the proposed work is mired in several of the most technically demanding parts of systems programming—concurrent programming language semantics, compiler optimization, operating systems, and weak memory hardware—it is not immediately clear how it

¹¹E.g. <http://blog.regehr.org/archives/category/compilers> and <http://blog.regehr.org/archives/category/software-correctness>

can be used in the classroom (in fact, the PI's original motivation for finding compiler bugs was to protect students from ever having to deal with them). Undergraduate involvement might be best accomplished at the level of independent study projects. The PI typically supervises 1–2 of these per year and the students are highly motivated and research-capable. Additionally, the PI will seek out ways to integrate with Utah's compilers course which is currently taught by Matthew Might.

Correctness Impact Research impact can be greatly amplified via widely-used open source software. The PI has already reported more than 400 compiler bugs and most of these were fixed. Although it is not likely that another 400 bugs lurk in C++11 concurrency, if the PI can find and report even 40 bugs, the benefit will be substantial. The impact of these fixes is via preventing the costly and demoralizing debugging sessions that occur when a software project gets bitten by a compiler bug. In practice, large embedded software projects are invariably bitten by these bugs [33]. The cost of these bugs is difficult to quantify because time lost to is not recorded or reported.

7 Research Plan

7.1 Schedule and Division of Work

This proposal's budget asks for support for two students. The proposed work is over three years and it divides naturally across the students.

Student 1: This student will do the work associated with generating randomized C++ test cases, including race-freedom, invariants, and Csmith-supported test case reduction. This work almost certainly matches a PhD in terms of scope and difficulty.

Student 2: This student will develop execution environments that find fault-inducing executions of compiled C++ test cases. This work is perhaps a better match for one or two MS theses, but there may turn out to be PhD-level problems here.

7.2 Evaluation Plan

To evaluate the proposed work we will answer these questions:

1. Did we find bugs in implementations of C++11 concurrency?
2. Did we produce tools that are useful for compiler developers?
3. Did we produce new insights into the design and implementation of C++11 concurrency?

We do not believe it will be hard to tell if the work has been successful or not.

8 Results from Prior NSF Support

Note: *The PI has two current NSF awards related to the Emulab: “CRI: CRD: Keeping Emulab Tuned and Humming” and “MRI: Evolutionary Development of an Advanced Distributed Testbed.” These are awards that he took over upon the untimely death of their previous PI, Jay Lepreau.*

These awards are not related to Dr. Regehr’s research and his involvement is purely administrative and at a low level of effort.

The PI’s most closely related prior NSF award is CNS–0615367: “Improving Sensor Network Software Reliability through Language, Tool, and OS Co-Design,” in collaboration with Philip Levis and Dawson Engler at Stanford University, September 2006–August 2009. The Utah portion of this grant was \$210,000, with \$360,000 going to Stanford. Our work was based on the hypothesis that the reliability of TinyOS¹²—an operating system for sensor network nodes—could be significantly improved by adapting the system itself (it is much smaller than other systems-level software artifacts like Linux), by adapting its programming language nesC, and by augmenting its development tools to add static and dynamic checks for safety properties.

Our first crack at improving TinyOS’s reliability was motivated by the observation that many bugs’ root causes were found in the interfaces between nesC components the made up TinyOS applications. Moreover, although TinyOS is broken into many components, a relatively small number of interfaces is used. We wrote a collection of *contracts* for commonly-used interfaces and also created a tool for weaving contract checks into application code as it was being compiled. We found and reported a number of bugs and wrote a paper [3] but due to some implementation limitations (clunky integration with the build process, excessive code bloat, and failing to support the newer TinyOS 2) we did not deploy this tool.

Our second effort was motivated by the problems associated with type and memory safety bugs on sensor network nodes. The research problem was taking well-known methods for trapping these errors (i.e., type safe versions of C) and making them run on a platform with 4 KB of RAM and very limited I/O. This effort resulted in Safe TinyOS [13] which was deployed in TinyOS starting in version 2.1.

The third tool to emerge from our TinyOS work was a model checker and randomized tester called T-Check, which found a number of safety and liveness bugs in TinyOS 2. We released T-Check as open-source software.¹³ The interesting part of this work was not the tool, which pretty much re-implemented known techniques, but rather the specifications of TinyOS application behavior that we wrote and also some results where we were unable to make our model checker outperform randomized testing in terms of bug-finding power.

Fourth, we created Neutron [9] which leveraged type safe execution to create a user/kernel separation in TinyOS applications, permitting applications to reboot without disturbing the kernel, *and vice versa*. The chief benefit of partial reboot is that it makes it less likely that nodes need to expend energy rebuilding soft state such as routing tables. It was an interesting challenge to create a process model on platforms with 4 KB of RAM.

Finally, we created a static analysis tool for detecting the potential for a TinyOS application to overflow its call stack. Nothing was published about this tool since it simply re-implemented ideas from our earlier work [28]. The tool is part of the TinyOS 2 distribution.¹⁴

¹²<http://www.tinyos.net/>

¹³<http://www.cs.utah.edu/~peterlee/tcheck/>

¹⁴http://docs.tinyos.net/tinywiki/index.php/Stack_Analysis

References

- [1] Jade Alglave, Daniel Kroening, John Lugton, Vincent Nimal, and Michael Tautschnig. Soundness of data flow analyses for weak memory models. In *Proc. of the 9th Asian Symposium Programming Languages and Systems (APLAS)*, pages 272–288, Kenting, Taiwan, December 2011.
- [2] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: Running tests against hardware. In *Proc. of the 17th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 41–44, Saarbrücken, Germany, March 2011.
- [3] Will Archer, Philip Levis, and John Regehr. Interface contracts for TinyOS. In *Proc. of the Intl. Conf. on Information Processing in Sensor Networks (IPSN'07), SPOTS Track*, Cambridge, MA, April 2007.
- [4] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *Proc. of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 55–66, 2011.
- [5] Hans J. Boehm. *ISO/IEC JTC1 SC22 WG21 N2480: A Less Formal Explanation of the Proposed C++ Concurrency Memory Model*. International Organization for Standardization, December 2007.
<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2480.html>.
- [6] Hans J. Boehm. *ISO/IEC JTC1 SC22 WG21 N2338: Concurrency memory model compiler consequences*. International Organization for Standardization, August 2008.
<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2338.html>.
- [7] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *Proc. of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 68–78, Tucson, AZ, USA, June 2008.
- [8] Sebastian Burckhardt and Madanlal Musuvathi. Effective program verification for relaxed memory models. In *Proc. of the 20th International Conference on Computer Aided Verification*, pages 107–120, July 2008.
- [9] Yang Chen, Omprakash Gnawali, Maria Kazandjieva, Philip Levis, and John Regehr. Surviving sensor network software faults. In *Proc. of the 22nd ACM Symposium on Operating Systems Principles (SOSP 2009)*, Big Sky, MT, USA, October 2009.
- [10] Jong-Deok Choi and Andreas Zeller. Isolating failure-inducing thread schedules. In *Proc. of the Intl. Symp. on Software Testing and Analysis (ISSTA)*, pages 210–220, Rome, Italy, July 2002.
- [11] Ravi Chugh, Jan W. Voung, Ranjit Jhala, and Sorin Lerner. Dataflow analysis for concurrent programs using datarace detection. In *Proc. of the ACM SIGPLAN 2008 Conf. on Programming Language Design and Implementation (PLDI)*, Tucson, AZ, June 2008.
- [12] Nathan Cooperider. *Data-flow analysis for interrupt-driven microcontroller software*. PhD thesis, University of Utah, 2008.

- [13] Nathan Coopriider, William Archer, Eric Eide, David Gay, and John Regehr. Efficient memory safety for TinyOS. In *Proc. of the 5th ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, pages 205–218, Sydney, Australia, November 2007.
- [14] Nathan Coopriider and John Regehr. Pluggable abstract domains for analyzing embedded software. In *Proc. of the 2006 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 44–53, Ottawa, Canada, June 2006.
- [15] Nathan Coopriider and John Regehr. Offline compression for on-chip RAM. In *Proc. of the ACM SIGPLAN 2007 Conf. on Programming Language Design and Implementation (PLDI)*, pages 363–372, San Diego, CA, June 2007.
- [16] Zhong Deng, Matthew B. Dwyer, John Hatcliff, and Masaaki Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *Proc. of the 24th Intl. Conf. on Software Engineering (ICSE)*, pages 442–452, Orlando, FL, May 2002.
- [17] Eric Eide and John Regehr. Volatiles are miscompiled, and what to do about it. In *Proc. of the 2008 Intl. Conf. on Embedded Software (EMSOFT)*, Atlanta, GA, USA, October 2008.
- [18] Dawson Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proc. of the 19th ACM Symp. on Operating Systems Principles (SOSP)*, pages 237–252, Bolton Landing, NY, October 2003.
- [19] Corman Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Proc. of the ACM SIGPLAN '00 Conf. on Programming Language Design and Implementation (PLDI)*, Vancouver, Canada, June 2000.
- [20] Ganesh Gopalakrishnan, Yue Yang, and Hemantkumar Sivaraj. QB or not QB: An efficient execution verification tool for memory orderings. In *Proc. of the 16th International Conference on Computer Aided Verification (CAV)*, pages 401–413, Boston, MA, USA, July 2004.
- [21] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *Proc. of the ACM SIGPLAN 2004 Conf. on Programming Language Design and Implementation (PLDI)*, Washington, DC, June 2004.
- [22] Ahmad Heydari and Saeed Azimi. A survey in deterministic replaying approaches in multiprocessors. *International Journal of Electrical and Computer Sciences*, 10(4), August 2010. <http://www.ijens.org/107004-5959%20IJECS-IJENS.pdf>.
- [23] Leslie Lamport. Time, clocks and ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–565, July 1978.
- [24] Leslie Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *ACM Trans. Comput. Systems*, 28:690–691, September 1979.
- [25] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

- [26] Paul E. McKenney, Clark Nelson, Hans J. Boehm, and Lawrence Crowl. *ISO/IEC JTC1 SC22 WG14 N1444: Dependency Ordering for C Memory Model*. International Organization for Standardization, March 2010.
<http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1444.htm>.
- [27] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Context-sensitive correlation analysis for detecting races. In *Proc. of the ACM SIGPLAN 2006 Conf. on Programming Language Design and Implementation (PLDI)*, June 2006.
- [28] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(4):751–778, November 2005.
- [29] Koushik Sen. Race directed random testing of concurrent programs. In *Proc. of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 11–21, June 2008.
- [30] Nicholas Sterling. WARLOCK - A static data race analysis tool. In *Proc. of the USENIX Winter 1993 Technical Conf*, pages 97–106, San Diego, CA, January 1993.
- [31] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY: static race detection on millions of lines of code. In *Proc. of the 6th joint meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC-FSE)*, pages 205–214, New York, NY, USA, 2007. ACM.
- [32] Anthony Williams. *C++ Concurrency in Action*. Manning Publications, 2012.
<http://www.manning.com/williams/>.
- [33] Michael Wolfe. How compilers and tools differ for embedded systems. In *Proc. of the Intl. Conf. on Compilers, Architecture and Synthesis for Embedded Systems*, San Francisco, CA, 2005. Keynote address. http://www.pgroup.com/lit/articles/pgi_article_cases.pdf.
- [34] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proc. of the ACM SIGPLAN 2011 Conf. on Programming Language Design and Implementation (PLDI)*, 2011.
- [35] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.

NSF Biographical Sketch — John Regehr

Professional Preparation

Kansas State University	Computer Science	BS, 1995
Kansas State University	Mathematics	BS, 1995
University of Virginia	Computer Science	MCS, 1997
University of Virginia	Computer Science	PhD, 2001
University of Utah	Computer Science	Postdoc, 2001–2003

Appointments

University of Utah	Associate Professor, School of Computing	2009–present
University of Utah	Assistant Professor, School of Computing	2003–2009
University of Utah	Adjunct Assistant Professor, School of Computing	2002–2003

Related Publications

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr, “Finding and Understanding Bugs in C Compilers.” In *Proc. of 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011)*, San Jose, CA, USA, June 2011.

<http://www.cs.utah.edu/~regehr/papers/pldi11-preprint.pdf>

Eric Eide and John Regehr, “Volatiles are miscompiled, and what to do about it.” In *Proc. of the ACM Conference on Embedded Software (EMSOFT)*, Atlanta, GA, October 2008.

<http://www.cs.utah.edu/~regehr/papers/emsoft08-preprint.pdf>

John Regehr, Nathan Coopriider, and David Gay, “Atomicity and Visibility in Tiny Embedded Systems.” In *Proc. of the PLOS 2006 Workshop on Linguistic Support for Modern Operating Systems*, San Jose, CA, October 2006.

<http://www.cs.utah.edu/~regehr/papers/plos06b.pdf>

John Regehr, “Random testing of interrupt-driven software.” In *Proc. of the ACM Conf. on Embedded Software (EMSOFT)*, Jersey City, NJ, September 2005.

<http://www.cs.utah.edu/~regehr/papers/emsoft05/>

John Regehr and Alastair Reid, “HOIST: A system for automatically deriving static analyzers for embedded systems.” In *Proc. of the 11th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Boston, MA, Oct. 2004.

<http://www.cs.utah.edu/~regehr/papers/asplos04/>

Other publications

Peng Li and John Regehr, “T-Check: Bug Finding for Sensor Networks.” In *Proc. of the International Conference on Information Processing in Sensor Networks (IPSN)*, SPOTS track, Stockholm, Sweden, April 2010. <http://www.cs.utah.edu/~regehr/papers/ipsn553s-li.pdf>

Yang Chen, Omprakash Gnawali, Maria Kazandjieva, Philip Levis, and John Regehr, “Surviving Sensor Network Software Faults.” In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP 2009)*, Big Sky, MT, USA, October 2009. <http://www.sigops.org/sosp/sosp09/papers/chen-sosp09.pdf>

Xuejun Yang, Nathan Coopriider, and John Regehr, "Eliminating the Call Stack to Save RAM." In *Proc. of the ACM Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2009)*, Dublin, Ireland, June 2009.

<http://www.cs.utah.edu/~regehr/papers/lctes062-yang.pdf>

Nathan Coopriider, William Archer, Eric Eide, David Gay, John Regehr, "Efficient Memory Safety for TinyOS." In *Proc. of the 5th ACM Conference on Embedded Networked Sensor Systems (SenSys 2007)*, Sydney, Australia, November 2007.

<http://www.cs.utah.edu/~regehr/papers/coop-sensys07.pdf>

Nathan Coopriider and John Regehr, "Offline Compression for On-Chip RAM." In *Proc. of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, San Diego, CA, June 2007.

<http://www.cs.utah.edu/~regehr/papers/pldi075-coopriider.pdf>

Synergistic Activities

I gave a talk "Exposing Difficult Compilers Bugs With Random Testing" at the 2010 GCC Developers' Summit. My student Xuejun Yang, the primary Csmith developer, gave a talk "Hardening LLVM With Random Testing" at the 2010 LLVM Developers' Meeting.

My group's Csmith tool is open source and has been responsible for finding more than 400 bugs in production-quality C compilers; we reported these bugs and most have been fixed

<http://embed.cs.utah.edu/csmith/>

Collaborators

Vikram Adve (UIUC), Venkat Chakravarthy (NetApp), Alan Davis (Utah), Bjorn De Sutter (Ghent, Belgium), Usit Duongsaa (Microsoft), Eric Eide (Utah), Dawson Engler (Stanford), David Gay (Google), Alex Groce (Oregon State), Omprakash Gnawali (University of Houston), Michael B. Jones (Microsoft), Maria Kazandjieva (Stanford), Insup Lee (University of Pennsylvania), Philip Levis (Stanford), Guodong Li (Fujitsu), Jens Palsberg (UCLA), Alastair Reid (ARM), Oleg Sokolsky (University of Pennsylvania), John A. Stankovic (University of Virginia), Xuejun Yang (Microsoft)

Graduate Advisor and Postdoctoral Sponsor

Advisor: John A. Stankovic (University of Virginia)

Postdoctoral sponsor: Jay Lepreau (University of Utah)

PhD Advisees: 5

Anton Burtsev, Yang Chen, Nathan Coopriider (now at BAE Systems), Jianjun Duan, Xuejun Yang, Lu Zhao

Postdoctoral Fellows Sponsored: 0

SUMMARY PROPOSAL BUDGET

YEAR 1

ORGANIZATION University of Utah				FOR NSF USE ONLY			
				PROPOSAL NO.	DURATION (months)		
PRINCIPAL INVESTIGATOR / PROJECT DIRECTOR John D Regehr				AWARD NO.	Proposed	Granted	
				A. SENIOR PERSONNEL: PI/PD, Co-PI's, Faculty and Other Senior Associates (List each separately with title, A.7. show number in brackets)			
	CAL	ACAD	SUMR				
1. John D Regehr - Principal Investigator	0.00	0.00	1.00	11,905			
2.							
3.							
4.							
5.							
6. (0) OTHERS (LIST INDIVIDUALLY ON BUDGET JUSTIFICATION PAGE)	0.00	0.00	0.00	0			
7. (1) TOTAL SENIOR PERSONNEL (1 - 6)	0.00	0.00	1.00	11,905			
B. OTHER PERSONNEL (SHOW NUMBERS IN BRACKETS)							
1. (0) POST DOCTORAL SCHOLARS	0.00	0.00	0.00	0			
2. (0) OTHER PROFESSIONALS (TECHNICIAN, PROGRAMMER, ETC.)	0.00	0.00	0.00	0			
3. (2) GRADUATE STUDENTS				68,250			
4. (0) UNDERGRADUATE STUDENTS				0			
5. (0) SECRETARIAL - CLERICAL (IF CHARGED DIRECTLY)				0			
6. (0) OTHER				0			
TOTAL SALARIES AND WAGES (A + B)				80,155			
C. FRINGE BENEFITS (IF CHARGED AS DIRECT COSTS)				13,960			
TOTAL SALARIES, WAGES AND FRINGE BENEFITS (A + B + C)				94,115			
D. EQUIPMENT (LIST ITEM AND DOLLAR AMOUNT FOR EACH ITEM EXCEEDING \$5,000.)							
TOTAL EQUIPMENT				0			
E. TRAVEL							
1. DOMESTIC (INCL. CANADA, MEXICO AND U.S. POSSESSIONS)				3,000			
2. FOREIGN				3,500			
F. PARTICIPANT SUPPORT COSTS							
1. STIPENDS \$ _____				0			
2. TRAVEL _____				0			
3. SUBSISTENCE _____				0			
4. OTHER _____				0			
TOTAL NUMBER OF PARTICIPANTS (0)				TOTAL PARTICIPANT COSTS		0	
G. OTHER DIRECT COSTS							
1. MATERIALS AND SUPPLIES				1,000			
2. PUBLICATION COSTS/DOCUMENTATION/DISSEMINATION				0			
3. CONSULTANT SERVICES				0			
4. COMPUTER SERVICES				0			
5. SUBAWARDS				0			
6. OTHER				0			
TOTAL OTHER DIRECT COSTS				1,000			
H. TOTAL DIRECT COSTS (A THROUGH G)				101,615			
I. INDIRECT COSTS (F&A)(SPECIFY RATE AND BASE) Facilities and Administration (Rate: 49.5000, Base: 101615)							
TOTAL INDIRECT COSTS (F&A)				50,299			
J. TOTAL DIRECT AND INDIRECT COSTS (H + I)				151,914			
K. RESIDUAL FUNDS				0			
L. AMOUNT OF THIS REQUEST (J) OR (J MINUS K)				151,914			
M. COST SHARING PROPOSED LEVEL \$ 0				AGREED LEVEL IF DIFFERENT \$			
PI/PD NAME John D Regehr				FOR NSF USE ONLY			
ORG. REP. NAME* LAUREL Duncan				INDIRECT COST RATE VERIFICATION			
		Date Checked	Date Of Rate Sheet	Initials - ORG			

1 *ELECTRONIC SIGNATURES REQUIRED FOR REVISED BUDGET

SUMMARY PROPOSAL BUDGET

YEAR 2

ORGANIZATION University of Utah				FOR NSF USE ONLY			
				PROPOSAL NO.	DURATION (months)		
PRINCIPAL INVESTIGATOR / PROJECT DIRECTOR John D Regehr				AWARD NO.	Proposed	Granted	
				A. SENIOR PERSONNEL: PI/PD, Co-PI's, Faculty and Other Senior Associates (List each separately with title, A.7. show number in brackets)			
	CAL	ACAD	SUMR				
1. John D Regehr - Principal Investigator	0.00	0.00	1.00	12,381			
2.							
3.							
4.							
5.							
6. (0) OTHERS (LIST INDIVIDUALLY ON BUDGET JUSTIFICATION PAGE)	0.00	0.00	0.00	0			
7. (1) TOTAL SENIOR PERSONNEL (1 - 6)	0.00	0.00	1.00	12,381			
B. OTHER PERSONNEL (SHOW NUMBERS IN BRACKETS)							
1. (0) POST DOCTORAL SCHOLARS	0.00	0.00	0.00	0			
2. (0) OTHER PROFESSIONALS (TECHNICIAN, PROGRAMMER, ETC.)	0.00	0.00	0.00	0			
3. (2) GRADUATE STUDENTS				69,750			
4. (0) UNDERGRADUATE STUDENTS				0			
5. (0) SECRETARIAL - CLERICAL (IF CHARGED DIRECTLY)				0			
6. (0) OTHER				0			
TOTAL SALARIES AND WAGES (A + B)				82,131			
C. FRINGE BENEFITS (IF CHARGED AS DIRECT COSTS)				14,346			
TOTAL SALARIES, WAGES AND FRINGE BENEFITS (A + B + C)				96,477			
D. EQUIPMENT (LIST ITEM AND DOLLAR AMOUNT FOR EACH ITEM EXCEEDING \$5,000.)							
TOTAL EQUIPMENT				0			
E. TRAVEL							
1. DOMESTIC (INCL. CANADA, MEXICO AND U.S. POSSESSIONS)				3,120			
2. FOREIGN				3,640			
F. PARTICIPANT SUPPORT COSTS							
1. STIPENDS \$ _____				0			
2. TRAVEL _____				0			
3. SUBSISTENCE _____				0			
4. OTHER _____				0			
TOTAL NUMBER OF PARTICIPANTS (0)				TOTAL PARTICIPANT COSTS		0	
G. OTHER DIRECT COSTS							
1. MATERIALS AND SUPPLIES				1,040			
2. PUBLICATION COSTS/DOCUMENTATION/DISSEMINATION				0			
3. CONSULTANT SERVICES				0			
4. COMPUTER SERVICES				0			
5. SUBAWARDS				0			
6. OTHER				0			
TOTAL OTHER DIRECT COSTS				1,040			
H. TOTAL DIRECT COSTS (A THROUGH G)				104,277			
I. INDIRECT COSTS (F&A)(SPECIFY RATE AND BASE) Facilities and Administration (Rate: 49.5000, Base: 104277)							
TOTAL INDIRECT COSTS (F&A)				51,617			
J. TOTAL DIRECT AND INDIRECT COSTS (H + I)				155,894			
K. RESIDUAL FUNDS				0			
L. AMOUNT OF THIS REQUEST (J) OR (J MINUS K)				155,894			
M. COST SHARING PROPOSED LEVEL \$ 0				AGREED LEVEL IF DIFFERENT \$			
PI/PD NAME John D Regehr				FOR NSF USE ONLY			
ORG. REP. NAME* LAUREL Duncan				INDIRECT COST RATE VERIFICATION			
		Date Checked	Date Of Rate Sheet	Initials - ORG			

2 *ELECTRONIC SIGNATURES REQUIRED FOR REVISED BUDGET

SUMMARY PROPOSAL BUDGET

YEAR 3

ORGANIZATION University of Utah				FOR NSF USE ONLY			
				PROPOSAL NO.	DURATION (months)		
PRINCIPAL INVESTIGATOR / PROJECT DIRECTOR John D Regehr				AWARD NO.	Proposed	Granted	
				A. SENIOR PERSONNEL: PI/PD, Co-PI's, Faculty and Other Senior Associates (List each separately with title, A.7. show number in brackets)			
	CAL	ACAD	SUMR				
1. John D Regehr - Principal Investigator	0.00	0.00	1.00	12,876			
2.							
3.							
4.							
5.							
6. (0) OTHERS (LIST INDIVIDUALLY ON BUDGET JUSTIFICATION PAGE)	0.00	0.00	0.00	0			
7. (1) TOTAL SENIOR PERSONNEL (1 - 6)	0.00	0.00	1.00	12,876			
B. OTHER PERSONNEL (SHOW NUMBERS IN BRACKETS)							
1. (0) POST DOCTORAL SCHOLARS	0.00	0.00	0.00	0			
2. (0) OTHER PROFESSIONALS (TECHNICIAN, PROGRAMMER, ETC.)	0.00	0.00	0.00	0			
3. (2) GRADUATE STUDENTS				71,250			
4. (0) UNDERGRADUATE STUDENTS				0			
5. (0) SECRETARIAL - CLERICAL (IF CHARGED DIRECTLY)				0			
6. (0) OTHER				0			
TOTAL SALARIES AND WAGES (A + B)				84,126			
C. FRINGE BENEFITS (IF CHARGED AS DIRECT COSTS)				14,739			
TOTAL SALARIES, WAGES AND FRINGE BENEFITS (A + B + C)				98,865			
D. EQUIPMENT (LIST ITEM AND DOLLAR AMOUNT FOR EACH ITEM EXCEEDING \$5,000.)							
TOTAL EQUIPMENT				0			
E. TRAVEL							
1. DOMESTIC (INCL. CANADA, MEXICO AND U.S. POSSESSIONS)				3,245			
2. FOREIGN				3,786			
F. PARTICIPANT SUPPORT COSTS							
1. STIPENDS \$ _____				0			
2. TRAVEL _____				0			
3. SUBSISTENCE _____				0			
4. OTHER _____				0			
TOTAL NUMBER OF PARTICIPANTS (0)				TOTAL PARTICIPANT COSTS		0	
G. OTHER DIRECT COSTS							
1. MATERIALS AND SUPPLIES				1,082			
2. PUBLICATION COSTS/DOCUMENTATION/DISSEMINATION				0			
3. CONSULTANT SERVICES				0			
4. COMPUTER SERVICES				0			
5. SUBAWARDS				0			
6. OTHER				0			
TOTAL OTHER DIRECT COSTS				1,082			
H. TOTAL DIRECT COSTS (A THROUGH G)				106,978			
I. INDIRECT COSTS (F&A)(SPECIFY RATE AND BASE) Facilities and Administration (Rate: 49.5000, Base: 106978)							
TOTAL INDIRECT COSTS (F&A)				52,954			
J. TOTAL DIRECT AND INDIRECT COSTS (H + I)				159,932			
K. RESIDUAL FUNDS				0			
L. AMOUNT OF THIS REQUEST (J) OR (J MINUS K)				159,932			
M. COST SHARING PROPOSED LEVEL \$ 0				AGREED LEVEL IF DIFFERENT \$			
PI/PD NAME John D Regehr				FOR NSF USE ONLY			
ORG. REP. NAME* LAUREL Duncan				INDIRECT COST RATE VERIFICATION			
		Date Checked	Date Of Rate Sheet	Initials - ORG			

3 *ELECTRONIC SIGNATURES REQUIRED FOR REVISED BUDGET

SUMMARY PROPOSAL BUDGET Cumulative

ORGANIZATION University of Utah				FOR NSF USE ONLY			
				PROPOSAL NO.	DURATION (months)		
PRINCIPAL INVESTIGATOR / PROJECT DIRECTOR John D Regehr				AWARD NO.	Proposed	Granted	
				A. SENIOR PERSONNEL: PI/PD, Co-PI's, Faculty and Other Senior Associates (List each separately with title, A.7. show number in brackets)			
	CAL	ACAD	SUMR				
1. John D Regehr - Principal Investigator	0.00	0.00	3.00	37,162			
2.							
3.							
4.							
5.							
6. () OTHERS (LIST INDIVIDUALLY ON BUDGET JUSTIFICATION PAGE)	0.00	0.00	0.00	0			
7. (1) TOTAL SENIOR PERSONNEL (1 - 6)	0.00	0.00	3.00	37,162			
B. OTHER PERSONNEL (SHOW NUMBERS IN BRACKETS)							
1. (0) POST DOCTORAL SCHOLARS	0.00	0.00	0.00	0			
2. (0) OTHER PROFESSIONALS (TECHNICIAN, PROGRAMMER, ETC.)	0.00	0.00	0.00	0			
3. (6) GRADUATE STUDENTS				209,250			
4. (0) UNDERGRADUATE STUDENTS				0			
5. (0) SECRETARIAL - CLERICAL (IF CHARGED DIRECTLY)				0			
6. (0) OTHER				0			
TOTAL SALARIES AND WAGES (A + B)				246,412			
C. FRINGE BENEFITS (IF CHARGED AS DIRECT COSTS)				43,045			
TOTAL SALARIES, WAGES AND FRINGE BENEFITS (A + B + C)				289,457			
D. EQUIPMENT (LIST ITEM AND DOLLAR AMOUNT FOR EACH ITEM EXCEEDING \$5,000.)							
TOTAL EQUIPMENT				0			
E. TRAVEL							
1. DOMESTIC (INCL. CANADA, MEXICO AND U.S. POSSESSIONS)				9,365			
2. FOREIGN				10,926			
F. PARTICIPANT SUPPORT COSTS							
1. STIPENDS \$ _____				0			
2. TRAVEL _____				0			
3. SUBSISTENCE _____				0			
4. OTHER _____				0			
TOTAL NUMBER OF PARTICIPANTS (0)				TOTAL PARTICIPANT COSTS		0	
G. OTHER DIRECT COSTS							
1. MATERIALS AND SUPPLIES				3,122			
2. PUBLICATION COSTS/DOCUMENTATION/DISSEMINATION				0			
3. CONSULTANT SERVICES				0			
4. COMPUTER SERVICES				0			
5. SUBAWARDS				0			
6. OTHER				0			
TOTAL OTHER DIRECT COSTS				3,122			
H. TOTAL DIRECT COSTS (A THROUGH G)				312,870			
I. INDIRECT COSTS (F&A)(SPECIFY RATE AND BASE)							
TOTAL INDIRECT COSTS (F&A)				154,870			
J. TOTAL DIRECT AND INDIRECT COSTS (H + I)				467,740			
K. RESIDUAL FUNDS				0			
L. AMOUNT OF THIS REQUEST (J) OR (J MINUS K)				467,740			
M. COST SHARING PROPOSED LEVEL \$ 0				AGREED LEVEL IF DIFFERENT \$			
PI/PD NAME John D Regehr				FOR NSF USE ONLY			
ORG. REP. NAME* LAUREL Duncan				INDIRECT COST RATE VERIFICATION			
		Date Checked	Date Of Rate Sheet	Initials - ORG			

C *ELECTRONIC SIGNATURES REQUIRED FOR REVISED BUDGET

Budget Justification

The budget assumes a 4% inflation rate each year. The benefits rate for faculty and staff is 37%, while the rate for graduate students is 14%.

Senior Personnel

John Regehr is an Associate Professor in the School of Computing and the Principal Investigator for this project. The budget includes one summer month of support for his time during each year of the project.

Other Personnel

Graduate Students: The budget includes support for two graduate students at 50% FTE for 9 academic months and 100% FTE for 3 summer months during each year, for the duration of the project. The graduate students will be assigned research tasks as outlined in the proposal.

Other Direct Costs

Travel: Domestic and foreign travel are requested to sponsor trips to major conferences and collaboration efforts within the project.

Materials and Supplies: The budget for materials and supplies includes such things as software, books, and technical supplies.

Indirect Costs: The University of Utah's indirect costs rates calculated from the total direct costs less capital (<\$5,000) equipment. The negotiated rate is 49.5%.

Current and Pending Support

(See GPG Section II.D.8 for guidance on information to include on this form.)

The following information should be provided for each investigator and other senior personnel. Failure to provide this information may delay consideration of this proposal.			
Investigator: John Regehr	Other agencies (including NSF) to which this proposal has been/will		
Support: <input checked="" type="checkbox"/> Current <input type="checkbox"/> Pending	<input type="checkbox"/> Submission Planned in Near Future	<input type="checkbox"/> *Transfer of Support	
Project/Proposal Title: Container for Advanced Adaptive Applications			
Source of Support: Raytheon Company Total Award Amount: \$652,535 Total Award Period Covered: 9/24/2010-9/23/2012 Location of Project: University of Utah Person-Months Per Year Committed to the Project. Cal: Acad: Sumr: 1.0			
Support: <input checked="" type="checkbox"/> Current <input type="checkbox"/> Pending	<input type="checkbox"/> Submission Planned in Near Future	<input type="checkbox"/> *Transfer of Support	
Project/Proposal Title: CRI: CRD: Keeping Emulab Tuned and Humming			
Source of Support: NSF Total Award Amount: \$1,999,873 Total Award Period Covered: 9/1/2007-8/31/2012 Location of Project: University of Utah Person-Months Per Year Committed to the Project. Cal: Acad: Sumr:			
Support: <input checked="" type="checkbox"/> Current <input type="checkbox"/> Pending	<input type="checkbox"/> Submission Planned in Near Future	<input type="checkbox"/> *Transfer of Support	
Project/Proposal Title: MRI: Evolutionary Development of an Advanced Distributed Testbed			
Source of Support: NSF Total Award Amount: \$1,689,509 Total Award Period Covered: 9/1/2007-8/31/2012 Location of Project: University of Utah Person-Months Per Year Committed to the Project. Cal: Acad: Sumr:			
Support: <input checked="" type="checkbox"/> Current <input type="checkbox"/> Pending	<input type="checkbox"/> Submission Planned in Near Future	<input type="checkbox"/> *Transfer of Support	
Project/Proposal Title: Emulab Support for ATC-NY			
Source of Support: Air Force Material Command Total Award Amount: \$0 Total Award Period Covered: 8/6/2009-4/30/2013 Location of Project: Person-Months Per Year Committed to the Project. Cal: Acad: Sumr:			
Support: <input checked="" type="checkbox"/> Current <input type="checkbox"/> Pending	<input type="checkbox"/> Submission Planned in Near Future	<input type="checkbox"/> *Transfer of Support	
Project/Proposal Title: Maintaining Emulab Software for the Lockheed Martin National Cyber Range Project			
Source of Support: Lockheed Martin Total Award Amount: \$97,824 Total Award Period Covered: August 2011 - July 2012 Location of Project: University of Utah Person-Months Per Year Committed to the Project. Cal: Acad: Sumr:			
*If this project has previously been funded by another agency, please list and furnish information for immediately preceding funding period.			

Current and Pending Support

(See GPG Section II.D.8 for guidance on information to include on this form.)

The following information should be provided for each investigator and other senior personnel. Failure to provide this information may delay consideration of this proposal.			
Investigator: John Regehr	Other agencies (including NSF) to which this proposal has been/will		
Support: <input type="checkbox"/> Current <input checked="" type="checkbox"/> Pending <input type="checkbox"/> Submission Planned in Near Future <input type="checkbox"/> *Transfer of Support			
Project/Proposal Title: CSR:Small:Beating Implementations of C++11Concurrency Into Shape			
Source of Support: NSF Total Award Amount: \$467,739 Total Award Period Covered: 7/1/2012-6/30/2015 Location of Project: University of Utah Person-Months Per Year Committed to the Project. Cal: Acad: Sumr: 1.0			
Support: <input type="checkbox"/> Current <input checked="" type="checkbox"/> Pending <input type="checkbox"/> Submission Planned in Near Future <input type="checkbox"/> *Transfer of Support			
Project/Proposal Title: SHF:Small:Collaborative Research:Diversity and Feedback in Random Testing for Systems Software			
Source of Support: NSF Total Award Amount: \$249,036 Total Award Period Covered: 7/1/2012-6/30/2015 Location of Project: University of Utah Person-Months Per Year Committed to the Project. Cal: Acad: Sumr: .50			
Support: <input type="checkbox"/> Current <input type="checkbox"/> Pending <input type="checkbox"/> Submission Planned in Near Future <input type="checkbox"/> *Transfer of Support			
Project/Proposal Title: Source of Support: Total Award Amount: \$ Total Award Period Covered: Location of Project: Person-Months Per Year Committed to the Project. Cal: Acad: Sumr:			
Support: <input type="checkbox"/> Current <input type="checkbox"/> Pending <input type="checkbox"/> Submission Planned in Near Future <input type="checkbox"/> *Transfer of Support			
Project/Proposal Title: Source of Support: Total Award Amount: \$ Total Award Period Covered: Location of Project: Person-Months Per Year Committed to the Project. Cal: Acad: Sumr:			
Support: <input type="checkbox"/> Current <input type="checkbox"/> Pending <input type="checkbox"/> Submission Planned in Near Future <input type="checkbox"/> *Transfer of Support			
Project/Proposal Title: Source of Support: Total Award Amount: \$ Total Award Period Covered: Location of Project: Person-Months Per Year Committed to the Project. Cal: Acad: Sumr:			

*If this project has previously been funded by another agency, please list and furnish information for immediately preceding funding period.

The School of Computing provides state of the art computing facilities for both instructional and research use. Both facilities share a common network infrastructure that is based on gigabit fiber core that provides desktop connections at speeds ranging from fast ethernet (100 Mbps) up to gigabit ethernet where necessary. The School's network attaches via a gigabit fiber connection directly to the campus backbone, which in turn routes traffic to Abilene (Internet 2), vBNS, and the commodity Internet.

In addition to the shared network infrastructure, the core School of Computing facility supplies many centralized services, including shared disk space (4 Terabytes), time, web/cgi, ftp, firewall, backups, printing resources, and email. Most services run on Solaris-based hosts, ranging from Netra T1's to an Enterprise 5000. Several large-scale Solaris and Linux machines are also made available for general use.

The instructional computing facility includes over 180 Unix, Linux, and Windows-based machines. Most of these machines are organized into three laboratories with the remainder being situated in graduate student offices. The Undergraduate Lab in EMCB 210 includes approximately 100 1.4 gigahertz Athlon-based PCs. The electronic classroom in MEB 3225 contains 30 Pentium-based PCs arranged into a classroom configuration. The CES/Grad Lab in MEB 3161 contains 13 PCs based on Athlon 2100+ processors and GeForce 4 video cards.

The research computing facility is a heterogeneous mix of over 300 machines, including PC's, SGI, and Sun-based hardware. The research computing facility includes major laboratories devoted to computer-aided design and graphics, computer systems, asynchronous digital systems and VLSI, robotics and vision, scientific computing and imaging, and information retrieval and natural language processing. These research laboratories contain a wide array of specialized equipment, including:

- an SGI Origin 3800 (32 processors);
- an SGI Origin 2000 Reality Monster (96 processors, 8 IR heads);
- a 200-node network testbed and emulation facility;
- an SGI Power Onyx (14 processors, 2 RE2 heads);
- a multi-source nonlinear video editing environment;
- a real-time signal processing lab;
- an image analysis lab;
- equipment for various types of custom hardware design;
- a Sarcos Dextrous Arm, Utah/MIT Dextrous Hand, and PUMA 560 robots; and
- a Sarcos Treadport locomotion interface, several SensAble Phantom haptic interfaces, Fakespace Responsive Workbench, nVision Datavisor HiRes, and a variety of position trackers.

The College of Engineering operates a research-scale integrated circuit (IC) fabrication facility that is used extensively by the School of Computing. Equipment for testing and debugging both internally and externally fabricated circuits is housed in an integrated circuit testing facility that contains state-of-the-art HP, Tektronix and Micromanipulator automated IC testing equipment.

Data Management Plan — CSR: Small: Beating Implementations of C++11 Concurrency Into Shape

The data in the proposed project is primarily in the source code for testing tools. These tools will be open-source software and we will make them available on a public hosting site, such as GitHub. Using GitHub automatically provides us with excellent backup and archiving for the code and curricular material products of the project.

Empirical testing data will often be ephemeral and reproducible by re-running experiments with known seeds. If the computational burden to recreate data is excessively large we will store data in a compressed format on the open-source repository. Given the ever-increasing power of computers, we do not expect this to be the case.

We have no unusual format or metadata requirements; test cases themselves are source code files, and testing results are standard formats produced by instrumentation tools or stored in a standard SQL database.

We do not anticipate the need to work with sensitive or confidential information; no extraordinary practices are required in order to conduct this research.

Supplementary Document
List of Personnel

1. John Regehr; University of Utah; PI