Efficient Type and Memory Safety for Tiny Embedded Systems

John Regehr Nathan Cooprider Will Archer Eric Eide

University of Utah, School of Computing
{regehr, coop, warcher, eeide}@cs.utah.edu

Abstract

We report our experience in implementing type and memory safety in an efficient manner for sensor network nodes running TinyOS: tiny embedded systems running legacy, C-like code. A compiler for a safe language must often insert dynamic checks into the programs it produces; these generally make programs both larger and slower. In this paper, we describe our novel compiler toolchain, which uses a family of techniques to minimize or avoid these run-time costs. Our results show that safety can in fact be implemented cheaply on low-end 8-bit microcontrollers.

1 Introduction

Type safety and memory safety are useful in creating reliable software. In practice, however, safety comes at a price. The safe operation of a program usually cannot be assured by static methods alone, so dynamic checks must be inserted into the program to assure safety at run time.

To be practical in the development of resource-constrained embedded systems, the run-time costs of these checks must be extremely low. Consider that two of the most popular sensor network node types are the Mica2, based on an 8-bit Atmel AVR processor with 4 KB of RAM and 128 KB of flash memory, and the TelosB, based on a 16-bit TI MSP430 processor with 10 KB of RAM and 48 KB of flash. Programmers of such devices are loathe to surrender any resources to achieve the benefits that arise from safety. Moreover, developers are generally unwilling to use an entirely new programming language to obtain these benefits.

In this paper, we investigate whether language-based safety can be made suitable for the development of tiny embedded systems: in particular, systems based on microcontrollers and programmed in C-like languages. We describe our novel toolchain for programming these systems: our tools leverage the properties of tiny embedded systems to minimize the run-time overheads of safety. For example, small system size means that whole-program optimization is feasible, a well-defined concurrency model greatly minimizes the locks needed to maintain safety, and a static memory allocation model means that garbage collection is not required. The goal of our present work is to *characterize and minimize the run-time overheads of safety*. Utilizing the benefits of safety—e.g., to implement operating systems for embedded devices [8]—is outside the scope of this paper.

Our research builds directly on three existing projects: TinyOS, CCured, and cXprop. We use these tools in combination to create "Safe TinyOS" applications for the Mica2 and TelosB platforms.

TinyOS [5] is a popular set of components for implementing embedded software systems. Components are written in nesC [3], a C dialect that is translated into plain C. Components generally allocate memory statically: this policy helps programmers avoid difficult bugs.

CCured [6] is a source-to-source C compiler that transforms programs into ones that obey type and memory safety. As described above, these properties are generally enforced through a combination of static analyses and dynamic checks. In our toolchain, CCured processes the output of the nesC compiler.

Finally, cXprop [2] is an aggressive whole-program dataflow analyzer that we developed for C. It understands the TinyOS concurrency model, and in our toolchain, we use cXprop to optimize the safe programs produced by CCured. Like CCured, cXprop is based on CIL [7].

2 Efficient Safety for TinyOS Applications

Figure 1 shows our toolchain that produces Safe TinyOS applications. The crux of the problem is to ensure that the output of CCured is made as efficient as possible, and safe even in the presence of interrupt-driven concurrency.

2.1 Whole-program optimization

To ensure safety at run time, CCured must insert a check before every operation that cannot be proven safe at compile time. This potentially leads to a large number of checks and a significant increase in code size. CCured's internal optimizers reduce the number of checks that must be inserted into a program, but as we show in Section 3.1, many checks remain.

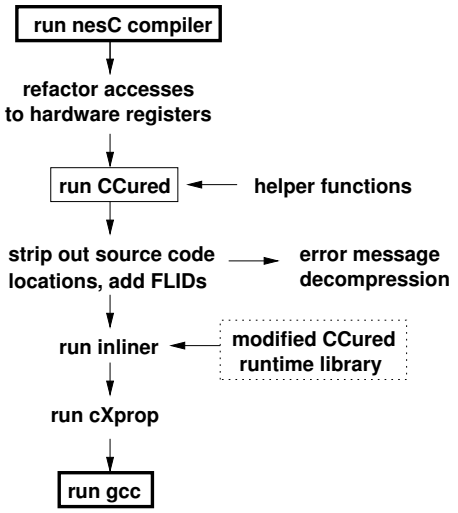

Figure 1: Our toolchain for Safe TinyOS applications. Boxed tools are ones we did not develop. The first and final steps—the nesC and C compilers—are the original TinyOS toolchain.

To reduce CCured's footprint, we post-process CCured's output with cXprop. Unlike CCured's optimizer, which only attempts to remove its own checks, cXprop will remove any part of a program that it can show is dead or useless. To support the current work, we improved cXprop in the following ways.

Concurrency analysis: Instead of relying on nesC's concurrency analysis, as we previously did [2], we implemented our own race condition detector that is conservative (nesC's analysis does not follow pointers) and slightly more precise. The results of this analysis, in addition to supporting sound analysis of concurrent code, supports the elimination of nested atomic sections and the avoidance of the need to save the state of the interrupt-enable bit for non-nested atomic sections.

Pointer analysis: We implemented a flow sensitive, field sensitive pointer analyzer that supports both may-alias and must-alias relations. May-alias information supports precise dataflow analysis while must-alias information supports strong updates and helps eliminate useless indirections after functions are inlined.

Function inlining: We wrote a source-to-source function inliner in CIL, which has several benefits. First, it greatly increases the precision of cXprop by adding some context sensitivity. Second, inlining before running GCC results in roughly 5% smaller executables than does relying on GCC to inline exactly the same functions, because GCC runs its inliner too late in the compilation process.

Dead code elimination: The DCE pass in GCC is not very strong; for example it fails to eliminate some of the trash left over after functions are inlined. Our stronger DCE pass does a better job, resulting in 3–5% improvement in code size. Also we implemented a copy propagation pass that eliminates useless variables and increases cXprop's dataflow analysis precision slightly.

2.2 Handling concurrency

CCured enforces safety for sequential programs only. Interrupt-driven embedded programs can invalidate CCured's invariants by, for example, overwriting a pointer between the time it is bounds-checked and the time it is dereferenced. Furthermore, programmers often expect that pointer updates occur atomically. CCured's *fat pointers*,[1] however, generally cannot be updated atomically without explicit locking.

We addressed these problems by taking advantage of the nesC programming model, which underlies all TinyOS programs. In TinyOS applications, nearly all variable accesses are already atomic due to the nature of nesC's two-level concurrency model—non-preemptive tasks and preemptive interrupt handlers—or due to explicit `atomic` sections in the code. These variables need no extra protection to preserve safety. Some variables are accessed non-atomically, however, and the nesC compiler outputs a list of these variables when a TinyOS application is compiled. We modified the CCured compiler to input this list and then insert locks around safety-critical section (i.e., our injected dynamic checks) that involve one or more non-atomic variables. We also needed to suppress uses of the `norace` nesC keyword, which causes the compiler to ignore potential race conditions.

2.3 Adapting the CCured runtime library

The CCured runtime library is substantial (several thousand source lines), and there are three problems with using it as-is on small devices. First, dependencies on OS services such as files and signals are woven into the runtime in a fine-grained way. Second, the runtime contains x86 dependencies: e.g., several of CCured's checks cast a pointer to an unsigned 32-bit integer to verify that it is aligned on a four-byte boundary. On the Mica2 and TelosB platforms, however, pointers are 16 bits and do not require four-byte alignment. Third, although the RAM and ROM overheads of the runtime are negligible for a PC, they are prohibitive for a sensor network node. For example, a version of the CCured runtime modified just enough to compile for a Mica2 mote used 1.6 KB of RAM, 40% of the total, and 33 KB of code memory, 26% of the total.

We removed the OS and x86 dependencies by hand and used an existing compile-time option to have CCured

[1] A fat pointer is implemented by three addresses: the pointer's value and the bottom and top of the pointer's allowable range. CCured may replace an "ordinary" pointer with a fat pointer in order to implement dynamic safety checks.

drop support for garbage collection. TinyOS applications have a static memory allocation model and do not require GC. Then, we applied our improved DCE analysis mentioned above to eliminate parts of the runtime that are unused by the TinyOS program being compiled. Together these techniques reduce the overhead of the CCured runtime library to 2 bytes of RAM and 314 bytes of ROM, for a minimal TinyOS application running on the Mica2 platform.

3 Evaluation

This section quantifies the cost of making TinyOS applications safe using our toolchain. Our duty cycle results are from Avrora [12], a cycle-accurate simulator for networks of Mica2 motes. We used a pre-release version of CCured 1.3.4, Avrora from current CVS as of February 2006, and TinyOS 1.x from current CVS.

3.1 Eliminating safety checks

CCured attempts to add as few dynamic safety checks as possible, and it also optimizes code to remove redundant and unnecessary checks. Our cXprop tool and GCC eliminate additional checks. To measure the effectiveness of these different optimizers, we transformed application source code so that for each check inserted by the CCured compiler, a unique string would be passed to the run-time failure handler. If an optimizer proves that the failure handler is unreachable from a given check, then the string that we added becomes unreferenced and a compiler pass eliminates it. The checks remaining in an executable can therefore be determined by counting the surviving unique strings.

Figure 2 compares the power of four ways of optimizing Safe TinyOS applications: (1) GCC by itself; (2) the CCured optimizer, then GCC; (3) the CCured optimizer, then cXprop without inlining, then GCC; and (4) the CCured optimizer, then cXprop with inlining, then GCC. The results show that the last technique, employing both cXprop and our custom inliner, is best by a significant margin. It was always the most effective, and it was the only strategy that always removed most of the checks. GCC alone was always the least effective, but we were nevertheless surprised that it could eliminate so many checks. These are primarily the "easy" checks such as redundant null-pointer checks. The CCured optimizer also removes easy checks and consequently it is not much more effective than GCC alone. Without inlining, cXprop is also not much more effective than GCC at removing checks. Although cXprop optimizes aggressively, it is hindered by context insensitivity: its analysis merges information from all calls to a given check, such as CCured's null-pointer check, making it far less likely that the check can be found to be useless. Inlining the checks provides the necessary context sensitivity.

3.2 Code size

Figure 3(a) shows the effect of various permutations of our toolchain on the code size of TinyOS applications, relative to the original, unsafe applications.

Similar to the safety check metric, the data show that cXprop and its inliner must be applied to achieve the best results. The first (leftmost) bar for each application shows that simply applying CCured to a TinyOS application increases its code size by approximately 20–90%. The second bar is even higher and shows the effect of moving the strings that CCured uses in error messages (file names, function names, etc.) from SRAM into flash memory. The third shows the effect of using CCured's "terse" option, which suppresses source location data in error messages. This reduces code size but the resulting error messages are much less useful. The fourth bar shows the effect of compressing error messages using FLIDs (as described in our technical report [8]). The fifth and sixth bars show that optimizing an application using cXprop, without and with an inlining pass, results in significant code size reductions.

The seventh bar shows that inlining and optimizing the *unsafe* application typically reduces its code size by 10–25%. Thus, cXprop represents a tradeoff: it can typically optimize a safe program so that its code size is near that of the unsafe original, or it can shrink the unsafe program. One might reasonably measure the cost of safety against the original baseline or the "new baseline" established by cXprop. Our ongoing work seeks to improve cXprop and thereby reduce both measured costs.

3.3 Data size

The value of cXprop is again apparent when we measure the cost of safety in terms of static data size. The first three bars of each group in Figure 3(b) show that simply applying CCured to a TinyOS application results in unacceptable RAM overhead. Many of the overheads are outrageously high—thousands of percent—though we have clipped the graph at 100%. The fourth bar shows that using FLIDs reduces RAM overhead substantially because many strings from the CCured runtime are eliminated. The fifth and sixth bars show that cXprop reduces RAM overhead still more, primarily through dead-variable elimination. The rightmost bar in each group shows that cXprop reduces the amount of static data for *unsafe* applications slightly, by propagating constant data into code and removing unused variables.

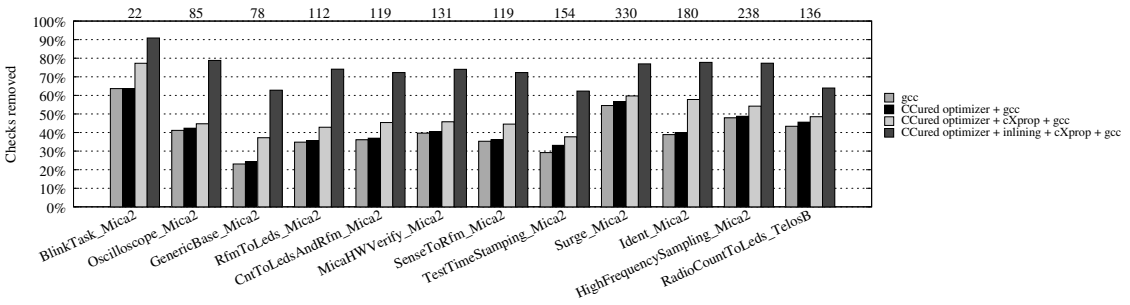

Figure 2: Percentage of checks inserted by the CCured compiler that can be eliminated using four different combinations of tools. The numbers at the top indicate the number of checks originally introduced by CCured.

3.4 Processor use

The efficiency of a sensor network application is commonly evaluated by measuring its *duty cycle*: the percentage of time that the processor is awake. For each application, we created a reasonable sensor network context for it to run in, and ran it for three simulated minutes.

Figure 3(c) shows the change in duty cycle across different versions of our applications. In general, CCured by itself slows an application by a few percent, while cXprop by itself speeds an application up by 3–10%. We were surprised to learn that using cXprop to optimize safe applications generally results in code that is about as fast as the *unsafe* original program—our original baseline. However, we again see a tradeoff: cXprop can eliminate the CPU cost of safety relative to the original baseline, or it can optimize the unsafe program. In future work we will see how further improvements to cXprop, designed to reduce the cost of safety, continue to apply to unsafe programs in general.

4 Related Work

As far as we know, until now no safe version of C has been run on sensor network nodes or any other embedded platform with similar resource constraints. However, other safe languages have been in use for a long time: e.g., Java Card [10] for smart cards based on 8-bit microcontrollers, and Esterel [1] for implementing state machines on small processors or directly in hardware. Despite the existence of these languages, most embedded software is implemented in unsafe languages.

We know of three ongoing efforts to bring the benefits of safe execution to sensor network applications. First, t-kernel [4] is a sensor net OS that supports untrusted native code without trusting the cross-compiler. It sacrifices backwards compatibility with TinyOS and was reported to make code run about 100% slower. Second, Rengaswamy et al. [9] provide memory protection in the SOS sensor network OS. The SOS protection model is weaker than ours: it emulates course-grained hardware protection, rather than providing fine-grained memory safety. Finally, Virgil [11] is a new safe language for tiny embedded systems such as sensor network nodes. Like TinyOS, Virgil is designed around static resource allocation, and like Java Card it supports objects.

5 Conclusion

We have reported our experience in developing, applying, and evaluating a tool chain for type- and memory-safe embedded systems. We have shown that these features can be supported for "Safe TinyOS" programs, and we have described our techniques that collectively enable safe and efficient programs for tiny microcontrollers. Our results show that language-based safety can be practical even for systems as small as sensor network nodes—in fact, safe, optimized TinyOS applications often use less CPU time than their unsafe, unoptimized counterparts. Our ongoing research seeks to further reduce the run-time costs of safety and thereby make its benefits applicable to a wider range of resource-constrained embedded systems.

References

[1] G. Berry. The foundations of Esterel. In *Proof, language, and interaction: essays in honour of Robin Milner*, pages 425–454. MIT Press, 2001.

[2] N. Cooprider and J. Regehr. Pluggable abstract domains for analyzing embedded software. In *Proc. LCTES*, June 2006.

[3] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. PLDI*, pages 1–11, June 2003.

[4] L. Gu and J. A. Stankovic. t-kernel: a translative OS kernel for sensor networks. Technical Report CS-2005-09, Dept. of Computer Science, Univ. of Virginia, 2005.

[5] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proc. ASPLOS*, pages 93–104, Nov. 2000.

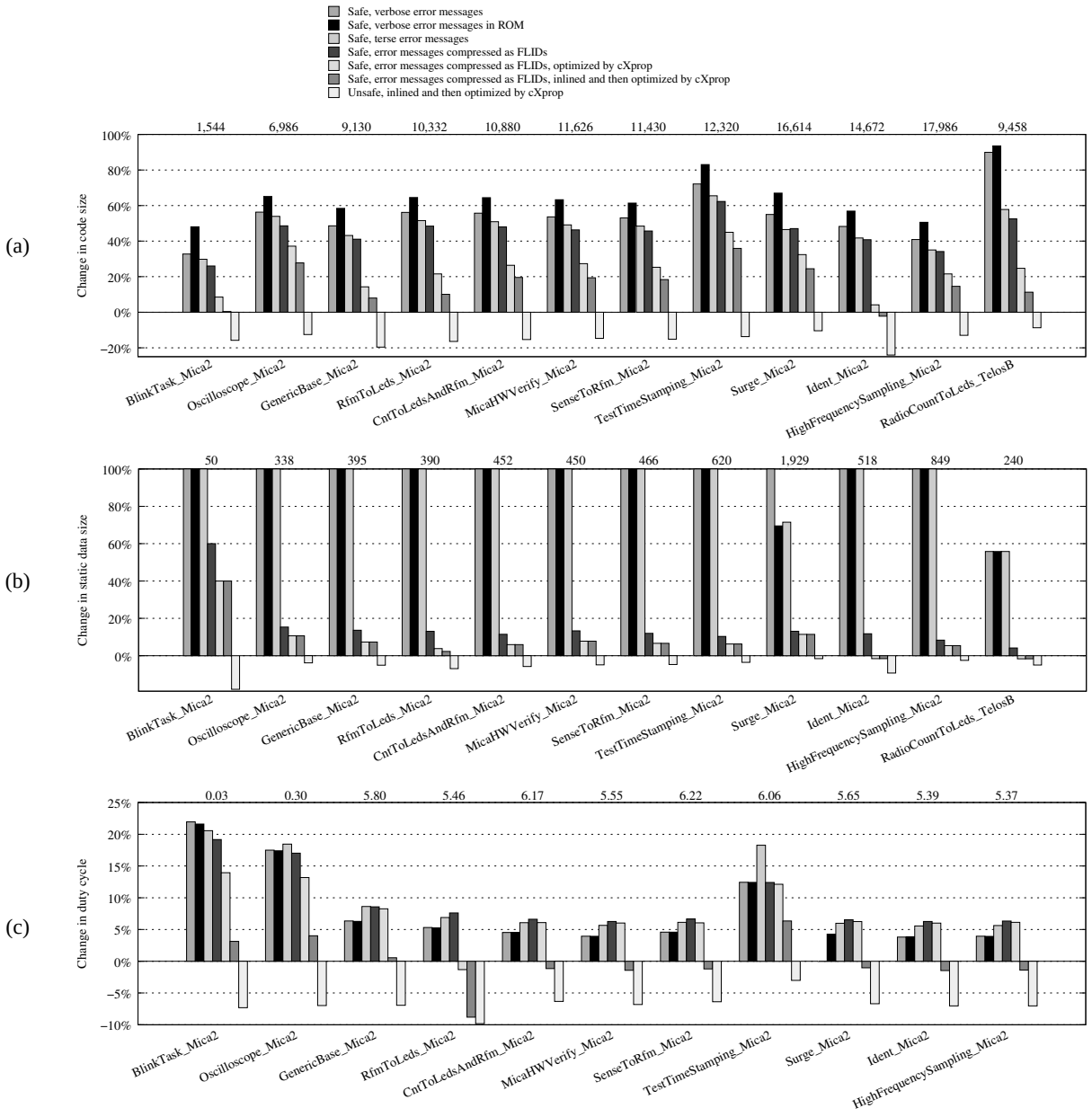

Figure 3: Resource usage relative to a baseline of the default unsafe, unoptimized TinyOS application. Numbers at the top of each graph show the resource usage of the baseline application, in bytes for memory graphs and in percent for the duty cycle graph.

[6] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *ACM TOPLAS*, 27(3), May 2005.

[7] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. Intl. Conf. on Compiler Construction (CC)*, pages 213–228, Apr. 2002.

[8] J. Regehr, N. Cooprider, W. Archer, and E. Eide. Memory safety and untrusted extensions for TinyOS. Technical Report UUCS–06–007, Univ. of Utah, June 2006.

[9] R. Rengaswamy, E. Kohler, and M. B. Srivastava. Software based memory protection in sensor nodes. Technical Report TR-UCLA-NESL-200603-01, Networked and Embedded Systems Lab., UCLA, Mar. 2006.

[10] Sun Microsystems. Java Card Spec. 2.2.2, Mar. 2006.

[11] B. L. Titzer. Virgil: Objects on the head of a pin. In *Proc. OOPSLA*, Oct. 2006. To appear.

[12] B. L. Titzer, D. Lee, and J. Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Proc. IPSN*, Apr. 2005.