Optimizing Interrupt-Driven Embedded Software

NATHAN COOPRIDER and JOHN REGEHR
University of Utah

Software for embedded microcontroller units (MCUs) represents both an interesting opportunity and a difficult challenge for compiler optimization. Since these systems tend to be small—often limited to a few KB of on-chip RAM—highly aggressive techniques are feasible and worthwhile. On the other hand, the effectiveness of traditional dataflow analyses is limited by their inability to cope with interrupt-driven concurrency and the direct interaction of embedded software with hardware devices.

We present an integrated collection of static analysis techniques that leads to effective, whole-program optimization of C code running on MCUs. Our first main contribution is a technique supporting whole-program dataflow analysis in the presence of interrupt-driven concurrency. This model is based on an automatic classification of each data element in a system based on its role in concurrent execution, followed by an analysis that efficiently approximates the effects of preemptions by interrupt handlers. Our second contribution is a technique for flowing data through volatile-qualified objects in MCU software, which can be backed by either RAM or device registers.

We implemented our dataflow framework in a tool called cXprop. As a standalone optimizer, cXprop can reduce code size of sensor network programs based on TinyOS by 12%, while reducing their CPU usage by 8.3%. We have also used cXprop as a crucial enabler for two other projects. First, we used its analysis to drive offline RAM compression, reducing the RAM requirements of TinyOS applications by an average of 22%. Second, we used cXprop’s analysis of concurrency to greatly increase the efficiency of Safe TinyOS, our project to provide memory safety to sensor net applications.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors; C.3 [Special-purpose and Application-based Systems]: Real-time and Embedded Systems

General Terms: Performance, Languages

Additional Key Words and Phrases: Abstract interpretation, TinyOS, embedded software

1. INTRODUCTION

Embedded microcontroller units (MCUs) are small, inexpensive systems-on-chip that have found vast applicability in adding software control to vehicles, appliances, consumer electronics, medical devices, robots, wireless sensor networks, and other products. MCU sales are expected to reach US$15.3 billion by 2009 [Semiconductor Industry Association 2006]. Software for MCUs typically must fit into on-chip RAM and ROM, which are highly constrained: generally between 0.1 and 100 KB. In this domain, widespread use of C compilers, as opposed to writing software in assembly,
is a relatively recent development.

Embedded software developers need to rapidly produce code for MCUs that is efficient, reliable, and reusable. Of course, these requirements conflict. Highly efficient software can be created through manual customization and optimization, though this often results in code that is fragile, buggy, and late. On the other hand, systems can be rapidly produced by assembling existing high-quality components, but the resulting code usually uses too many resources.

Optimizing compilers can help resolve this conflict by turning modular, high-level source code into highly efficient object code. However, optimizers for C are fundamentally limited in their ability to optimize small embedded software, for two reasons. First, and most importantly, C is a sequential language that has no semantics for concurrency. Correct concurrent code can be created in C, but compilers cannot “see through” interactions between different concurrent flows in order to more effectively optimize software. Second, low-level embedded software often interacts heavily with hardware devices. These devices have no meaning in the C language and again, the compiler must treat interactions conservatively, causing missed opportunities for optimization.

Our thesis is that software for embedded MCUs can be effectively optimized using analysis techniques that incorporate sound models of interrupt-driven concurrency and hardware devices. We have developed such techniques and implemented them in cXprop, a prototype tool. cXprop is a whole-program optimizer for embedded C code that implements an integrated value flow analysis, pointer analysis, race condition detection, and accounting for the effects of interrupts. cXprop is flow sensitive but path and context insensitive.

To perform precise dataflow analysis in the presence of interrupts, two related techniques are required. First, each memory object must be classified as sequential, concurrent, or racing. Sequential objects are touched only by a single concurrent flow. Concurrent objects may be touched by multiple flows, but are properly protected by locks. Any objects that are not sequential or concurrent is considered to be racing. The second technique required to perform dataflow analysis in the presence of interrupts is a method for augmenting a control flow graph with extra edges that account for transfers of control to interrupt handlers. We show that a relatively small number of edges needs to be added, resulting in efficient analysis.

A C program typically interacts with hardware devices using memory locations qualified as volatile. The volatile qualifier specifies that the usual C memory model does not apply to qualified objects, and that each source-level read and write must be translated into exactly one load or store in the generated code. Thus, volatile objects are opaque to the compiler. A second use of the volatile qualifier is to ensure that stores to global variables in one thread or interrupt become visible to other flows, and that loads from these shared variables do not receive values cached in registers. We have developed an analysis that disambiguates and separately treats these two uses of the volatile qualifier, in order to more effectively analyze embedded software.

Although cXprop handles generic C code, our benchmark suite consists of TinyOS applications. TinyOS is a component library and minimal operating system for wireless sensor network nodes. cXprop can reduce code size of sensor network...
programs based on TinyOS by 12%, while reducing their CPU usage by 8.3%. We have also used cXprop as a crucial enabler for two other projects. First, we used its analysis to drive offline RAM compression, reducing the RAM requirements of TinyOS applications by an average of 22%. Second, we used cXprop’s analysis of concurrency to greatly increase the efficiency of Safe TinyOS, our project to provide memory safety to sensor net applications.

The contributions of this article are:

— a novel model for precise dataflow analysis in the presence of interrupt-driven concurrency,
— a technique for disambiguating different uses of the volatile qualifier, and for soundly flowing data through volatile-qualified memory objects,
— cXprop, a research prototype implementing our techniques,
— a pluggable abstract domain interface for cXprop, making it easier to experiment with different dataflow analyses

This article is organized as follows. Section 2 presents the central analysis. Section 3 describes some of our design decisions and assumptions. Section 4 describes the cXprop implementation. Section 5 discusses the abstract domains that we have implemented for cXprop. In Section 6 we evaluate the analysis and apply it to TinyOS applications. Section 7 compares our work with prior research.

2. ANALYSIS OF DATA IN CONCURRENT MCU CODE

This section describes several extensions to basic abstract interpretation [Cousot and Cousot 1977] that greatly improve its precision.

2.1 Interrupt-driven concurrency

Interrupts traditionally cause problems for static analyses. They introduce the possibility that data will race. Informally, a race exists for a variable when it may be written by one flow while simultaneously being accessed by another. Each interrupt represents a possible control flow branch at any point during a program that interrupts could be enabled. A sound static analysis of concurrent code must deal with both the racing data and the extra flow edges.

A naive approach to approximating concurrency adds control flow edges at every point where a context switch may occur. That level of granularity is impossible at the C source code level. One C statement may translate to many assembly statements, any of which could have a context switch immediately before or after. On top of this soundness issue, adding extra flow edges to the control flow graph increases the time it takes to reach a fixed point.

Inter-atomic concurrent data-flowing (ICD) includes two solutions to deal with concurrency-related issues. First, ICD identifies certain types of data which concurrency will affect and the rest may be analyzed using traditional sequential methods. Second, ICD uses a novel technique to reduce the number of extra flow edges added by the analysis. This technique conservatively interleaves the flow between atomic sections of the program.

Data classification. Applications store data in the heap, the stack, or the static portion of the data segment. Figure 1 shows a data classification for the target
applications of this dissertation. C programs may allocate data dynamically on the heap. The traditional nesC programming convention does not allow for dynamic memory allocation, so the analysis treats data on the heap as $\perp$. The analysis could be extended to track heap data more precisely by enhancing the pointer analysis and treating heap data in a similar manner as statically allocated data. Local variables, function arguments, and return values are conceptually stored on the stack. Traditional data-flow analysis techniques track this data by assuming concurrent flows do not access variables on each others’ stack (see Section 3.7).

The static portion of the data segment stores statically allocated but still mutable global variables. In practice, the compiler stores all global variables in either the code segment or the static portion of the data segment. Tracking these variables is a major goal of the analysis. Statically allocated data holds global state information important to the control flow of the program. The analysis leverages increased information about statically allocated data to improve information about other classifications of data, such as locals.

Detecting racing variables requires distinguishing between shared and unshared data. Shared data is accessed from two or more flows concurrently. Unshared data is only accessed from one flow. Shared data may race, while unshared data cannot. ICD does not classify heap data because it is $\perp$ in the analysis. The classification of statically allocated data as shared or unshared is important for race detection.

A simple analysis of concurrent code can avoid tracking any shared data. Global data is treated as unknown, or $\perp$, at all program points. While this method of avoiding the problem is conservative for all forms of concurrency, it is not precise. Providing the analysis with some information about the form of concurrency being used improves its precision. ICD does this.

**Race detection.** ICD performs race detection in order to reduce unnecessary pessimism from concurrency. By identifying which pieces of state race, the analysis may focus on the race-free aspects of the code [O’Hearn 2007].

Static lockset analysis is used by ICD to detect racing variables [Engler and Ashcraft 2003; Sterling 1993; Voung et al. 2007]. Lockset analysis checks for races
by finding shared variables accessed without holding the appropriate locks. Many applications for tiny MCUs implement a single lock by turning interrupts off to act as holding the lock. Although ICD assumes that implementation, any type of identifiable lock could be analyzed with ICD using similar techniques. ICD tracks locks by introducing a special data-flow object that contains the four-element lattice \((\top, \text{hold}, \text{release}, \bot)\). For applications which disable interrupts for locking, the abstract state of the lock is equivalent to the current state of the interrupt bit. Section 2.2 describes how the analysis tracks the interrupt bit, which is typically qualified as volatile data.

ICD divides the code into synchronous and asynchronous sections at the function level. Synchronous sections are atomic with respect to each other while asynchronous sections may interrupt each other as well as synchronous sections. For applications only using interrupt-driven concurrency and no threads, code reachable only from main is synchronous and code reachable from interrupts is asynchronous. This classification allows racing to be formally defined by three specific properties:

—accessed without locking at least once,
—written in asynchronous or unlocked synchronous code,
—accessed in asynchronous code (locked or unlocked).

These three properties capture all cases where racing may occur.

The lockset analysis in ICD determines which variables have the above racing properties. ICD uses a two element lattice to identify asynchronous code. Functions are initialized as synchronous and then determined to be asynchronous if found to be reachable from an interrupt. The lockset analysis occurs concurrently with a points-to computation so that discovered pointer information may be leveraged.

**Race-detection case analysis.** Consider all the ways an access to a variable may occur in interrupt-driven concurrent code. Figure 2 shows a single access to a variable being interrupted by a flow which also uses the variable in some way. This straightforward example captures the interesting cases for racing data.

The interrupted flow may be in the process of reading or writing the variable at the occurrence of the interrupt. The case where the interrupted flow is not accessing the variable at the time of the interrupt is uninteresting because no concurrent access occurs. This also means that the case where the interrupted flow is

---

**Fig. 2.** Classification of interesting variable access combinations and which combinations automatically indicate a race

<table>
<thead>
<tr>
<th>Access</th>
<th>Interrupt use</th>
<th>Racing?</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>read</td>
<td>no</td>
</tr>
<tr>
<td>read</td>
<td>write</td>
<td>yes</td>
</tr>
<tr>
<td>write</td>
<td>read</td>
<td>yes</td>
</tr>
<tr>
<td>write</td>
<td>write</td>
<td>yes</td>
</tr>
</tbody>
</table>

---

\(^{1}\)The terms synchronous and asynchronous have many different meanings depending on the context [Jhala and Majumdar 2007; Muttersbach et al. 2000; Benveniste et al. 2003]. This dissertation adopts the definitions used by nesC [Gay et al. 2003] for synchronous and asynchronous.

ACM Journal Name, Vol. V, No. N, Month 20YY.
accessing the pertinent variable inside an atomic section is uninteresting. Atomic sections prevent interrupting flows mid-access. The interrupted flow may be either synchronous or asynchronous code.

The interrupting flow is asynchronous code (i.e., an interrupt) which may read or write the pertinent variable sometime during its execution. Synchronous code does not interrupt other code by definition. Interrupting flows which neither read nor write the pertinent variable are not interesting because this means no concurrent access occurs.

This leaves four interesting combinations, shown in the table of Figure 2. One case does not imply a race. This is when the interrupted flow’s access is a read and the interrupting flow only reads the variable. In this case no flow can change the value mid-access, since no flow changes the value. The other three cases have one of the flows writing to the variable, and that always indicates a racing variable in these contexts.

_Interleaving atomic sections._ The analysis tracks the status of the interrupt bit (see Section 2.2). Tracking the status of the interrupt bit enables ICD, a novel analysis technique. ICD is based on the following observations:

—Code executing with interrupts disabled always runs to completion.
—Code executing with interrupts enabled is guaranteed not to modify any shared data except for variables identified as racing.

ICD exploits the interrupt-driven concurrency model in order to obtain more accurate data-flow results.

ICD then creates control flow edges from the end of every atomic section to the beginning of every interrupt handler, and from the end of every interrupt handler back to the end of every atomic section. Figure 3 illustrates the additional edges. These edges cover all possible orderings of execution which may occur between atomic sections. Each interrupt gets interpreted and the results of those interpretations get merged together to start the interpretation after the atomic section terminates. ICD takes multiple interrupt firings into account by adding control flow edges from the end of each interrupt to the start of every interrupt.

ICD adds an edge from the end of each interrupt back to its start. This extra edge covers the case for interrupts which never leave atomic sections. If the interrupt does leave an atomic section, then ICD adds an edge back to the start of the interrupt. This edge simulates firing the interrupt at the end of the atomic section, but also covers multiple firings of the interrupt as well. Additional edges to interrupts at the start of the application are not necessary. The MCU code starts with interrupts disabled. Because the application starts with the lock held, none of the concurrent flows may execute prior to releasing the lock for the first time. At the time of unlocking ICD adds the control flow edges.

Data-flow cannot be tracked through racing globals when interrupts are enabled; ICD forces racing variables to ⊥ unless interrupts are disabled. Disabling interrupts acts as locking the variable. While the lock holds the program executes in a sequential fashion until the lock is released. Racing variables cannot be soundly tracked with interrupts enabled because a control flow switch may occur at any time and change the shared variable’s value.
ICD soundly approximates interrupt behavior without adding control flow edges at every location where interrupts may fire. The extra edges abstract all possible orderings of atomic sections. The analysis of code outside atomic sections is sound because of the data classification. The analysis does not track racing variables. Read-only variables and variables accessed only in synchronous code can be soundly approximated ignoring concurrency. The only other classification, protected shared variables, will have accesses only inside atomic sections.

2.2 Tracking volatile data

Volatile data in C may change in ways unknown at the source code level. For this reason data-flow analyses traditionally treat volatile data as $\perp$. However, some volatile data may be tracked by introducing processor specific information. For each piece of volatile data, the analysis determines if it is stored in SRAM or in a device register. The analysis then attempts to track those stored in SRAM as normal data, and those stored in a device register using hardware-specific rules. Volatiles are opaque at the language level but not at the system level.

**SRAM.** The analysis contains a sound specification of all possible mutators for the data. ICD, presented in Section 2.1, soundly approximates the behavior of data in SRAM, volatile or not. Racing variables marked as volatile will be detected by the analysis and kept as $\perp$. This makes it safe to ignore the volatile type quantifier during an analysis for variables in SRAM.

**Device registers.** The analysis treats device register accesses as $\perp$ by default, but accesses may be tracked more precisely by defining their behavior inside the analysis. The additional definitions occur as cases to look for inside the symbolic execution. When the case matches, special analysis behavior for defined device registers is called. This then allows the analysis to track the values in the device register. The analysis currently contains definitions for tracking the interrupt bit.
of the status register (SREG) for the ATmega128L MCU.

Information for tracking the SREG comes from inspection of idioms used in applications and from reading the ATmega128L documentation [atmel 2002]. Inspecting the application idioms indicates the common cases currently in use. For example, atomic sections in TinyOS code start by saving the SREG into a local. TinyOS code ends an atomic section by writing the value stored in the local back to the SREG. The MCU documentation indicates alternative approaches and behavior not explicit in the code. This includes the status of the interrupt-enabled bit at the start of an interrupt and after an interrupt returns. The documentation also gives some indication of the complexity of the entire SREG behavior. Of the 136 AVR assembly instructions, 55 set bits in the SREG, but only 5 deal with the interrupt-enabled bit. This encouraged tracking only the interrupt-enabled bit.

The interrupt-enabled bit’s value may be affected in several ways. There exists an assembly instruction for setting the bit, sei, and clearing the bit, cli. Returning from an interrupt, using the reti assembly instruction, also sets the interrupt bit. The code for a “signal” interrupt begins execution with the interrupt-enabled bit cleared, while other interrupts begin with interrupts enabled. Finally, a program may access the SREG directly as device register 0x5F. A direct access can read, save, and write the SREG value. The analysis correctly handles these cases as they occur in TinyOS code.

The analysis stores an abstract value representing the status of the interrupt-enabled bit. sei, cli, reti, and interrupts all set this value in the expected ways. Reading and writing the SREG works slightly differently. A read from the SREG stores the current status of the interrupt bit, but marks it as hidden from the rest of the analysis. This protects against unexpected uses of the SREG information, such as non-standard (for TinyOS) bit masking. Writing the SREG works two ways. If the written value is a hidden and previously saved SREG value, the interrupt-enabled bit value just gets restored. If the written value is not a hidden value, then it gets masked and the appropriate bit value gets stored in the abstract interrupt-enabled bit status.

2.3 Combined analyses

Cooperating analyses improve precision. The data-flow analysis presented in this dissertation involves value-flow analysis, points-to analysis, call graph construction, and concurrency analysis. Each of these analyses leverage information learned by the others. This creates circular dependencies which make iterating between them not as precise as executing them concurrently.

Consider conditional constant propagation (CCP), a well known and widely used static analysis technique [Wegman and Zadeck 1991]. CCP provides constant propagation (CP) and dead code detection (DCD) in a single integrated analysis pass. Figure 4 shows the two lattices that CCP is based on and Figure 5 shows a program fragment that can be successfully analyzed by CCP but not by iteratively applying CP and DCD. The crux of the problem is that both CP and DCD initially make optimistic assumptions that, respectively, each variable is constant and each branch is dead. However, iteratively running the analyses loses the optimistic assumption—an integrated analysis is required.

A relationship similar to that between CP and DCD exists between value-flow
Fig. 4. Conditional constant propagation is an analysis combining the constant propagation lattice (left) with the unreachable code elimination lattice (right).

```c
1  int tricky () {
2     int x = 1;
3     int count = 0;
4     do {
5         int b = x;
6         if (b != 1)
7             x = 2;
8         count += x;
9     } while (count < 10);
10    return x;
11 }
```

Fig. 5. Code showing the importance of combined analyses. CCP finds line 7 to be dead and x to be constant, but iterating CP and DCD does not find either.

analysis, points-to analysis, call graph construction, and concurrency analysis. The value-flow analysis depends on the points-to analysis, but points-to information depends on value-flow as well. These types of dependencies mean the greatest amount of information can be obtained by executing the analyses concurrently, which is what the analysis presented in this chapter does.

3. DESIGN DECISIONS

The analysis includes several additional features, including conditional X propagation. This section describes these additional features and also outlines the assumptions inter-atomic concurrent data-flow makes about the programs it analyzes.

3.1 Conditional X propagation

The analysis supports conditional X propagation, an interface for parameterization by a user-supplied abstract domain. Most pieces of an abstract interpreter, such as fixed point computation and the control flow graph construction logic, are independent of the domain. Conditional X propagation makes use of abstract interpretation theory in the sense that domains meeting the properties for termination can be plugged in and used to analyze C code. The value domains implemented for conditional X propagation are described in Section 5.

The abstract domain interface. Appendix ERROR shows the abstract domain interface for conditional X propagation. Domains are written in OCaml [French National Institute for Research in Computer Science and Control] and can be
written with little knowledge of CIL. In addition to 32 transfer functions corresponding to low-level C operations such as casts, comparisons, and arithmetic operators, the interface has 11 utility functions for lifting a concrete value into the abstract domain, for concretizing an abstract value, for computing the meet of two abstract values, etc. Included with the 32 transfer functions are six backwards functions that can increase analysis precision by restricting abstract values in code that is executed conditionally. Consider the following code:

```c
x = rand();
if (x==0) {
    ...
} else {
    ...
}
```

The variable `x` is `⊥` at the if condition, but a precise backwards transfer function can determine that `x` is zero in the true branch and `x` is non-zero in the false branch.

My experience is that implementing domains through this interface is difficult only when the domain itself is difficult to think about. My advisor implemented a collection of transfer functions for a toy domain in under an hour, at which point I used the new domain to analyze programs such as those in the SPEC2000 suite.

The design of the domain interface, however, restricts the kinds of domains that can be plugged in: conditional X propagation domains must "look like" constant propagation. For example, all ALU operations are visible to the abstract domain, but definitions and uses of variables are not. Also, the domain interface does not support relational abstract domains such as octagons [Miné 2001].

### 3.2 Tracking arrays

The analysis provides two methods for tracking data-flow through arrays. In the first method each cell of an array stores its own abstract value. This representation effectively turns each cell of an array into a normal program variable from the analysis’ point of view. This approach increases the runtime of the analysis by an order of magnitude. This increase comes from having more state elements to update and requiring more memory.

Arrays are not typically used as arbitrary collection cells. Instead, cells typically store similar values and are referenced in similar ways. This means that storing each cell separately creates significant information redundancy. This becomes especially true when the analysis cannot determine the index into an array during a write.

In order to address these issues, the analysis uses a collapsed representation for a second method of tracking data-flow through arrays. This representation uses a single abstract value for the entire array. This abstract value represents all possible values held by any of the cells in the array. While this may appear drastically less precise than the first method, in practice the penalty in precision is minor.

### 3.3 Pointer analysis

Static analysis of real C code must include an effective method for dealing with pointers. The analysis deals with pointers by including a pluggable interface for pointer domains. A constant domain only computes must-alias relationships, while
a set domain computes both must- and may-alias relationships. The analysis uses
the set domain to capture points-to sets. Possible pointed-to objects for the target
applications only include variables, their offsets, and device registers. The analysis
could be extended to deal with heap objects as well, but I have not done this because
my target applications do not allocate memory dynamically. In addition to sets of
pointed-to objects, the abstract value for a pointer may also be “not-null.” This
value indicates that although the pointer may alias any object, it is definitely not
NULL. This special case is useful when pointers are checked for being NULL prior to
their use.

Several difficult cases arise when computing points-to sets in a
context-insensitive analysis. Conflicting contexts result in widening pointers and
the data they point to. Consider the following example:

```c
int foo (int * a) {
    int y = 42;
    return (*a) + bar(&y);
}
int bar (int * b) {
    if (*b == 7)
        return -1;
    else
        return 1;
}
int main () {
    int x=7;
    return foo(&x) + bar(&x);
}
```

main calls functions foo and bar. foo also calls bar. Inside bar, b has a points-to
set of \{&y from foo, &x from main\}. x is ⊥ inside foo’s context and y is ⊥ inside
main’s context. These contexts from different call sites merge at the start of bar,
and both x and y become ⊥. These context-insensitivity issues may result in severe
information loss in an analysis.

The analysis includes a method for dealing with this introduction of imprecision
due to context insensitivity. The basic idea is a demand-driven introduction of
contexts for memory reads and writes. Each function call represents a potential loss
of context, so the analysis saves context at each call site. Each memory read/write
traverses up the call graph to find the appropriate context and access/modify it.
Appropriate contexts for each access to a variable occur when the variable is in
scope. To be in scope at a function, the variable must either be a global variable
or one of the function’s locals or arguments.

Figure 6 illustrates a memory write using this method. The call to foo uses y
as an alias for x, but y has other aliases as well from different call sites. Merging
these different calls to foo in the analysis results in x being ⊥ inside the function.
y is passed to bar as z. bar then writes 42 to what z aliases. The analysis looks up
the call graph to foo in order to dereference z. It finds that y may alias x, but that
x is not in scope at foo. The analysis continues up the call graph to where x is in
scope. This occurs at the original call to foo. Since the memory write may alias x
in this context, the saved state of x is updated from 7 to 7 or 42.
int x = 7;
foo(&x);
foo(int * y) {
    bar(y);
}
bar(int * z) {
    *z = 42;
}

This method of dealing with memory reads and writes in a context insensitive analysis is sound. A context insensitive analysis merges many contexts together. This method allows only the relevant contexts to be considered for a read, instead of including contexts where the aliased variable is out of scope. Writing memory still requires exploring the call graph for all locations where the aliases are in scope. Each alias is updated weakly, meaning the program may or may not actually modify that location from the memory write. This conservatively approximates the behavior, but is more precise than using only the first local context.

The analysis makes two assumptions about the manipulation of NULL pointers in order to further increase precision. The target platforms implement NULL as 0, so the analysis assumes that 0 used as a pointer is NULL. The analysis assumes that pointer arithmetic or adding an offset to a NULL pointer is an error. If the analysis encounters a potential occurrence of one of these two cases, the NULL value remains in the resulting set unmodified.

3.4 Improving lockset analysis
The context-insensitive lockset analysis lacks precision by itself and produces a prohibitively large number of false positives for racing variables. In order to avoid the analysis cost of a context-sensitive analysis, the analysis leverages the observation that many applications use locking mechanisms which are lexically scoped. In other words, a function call will not change the status of the lock in the calling function. nesC’s syntax for atomic sections produces lexically scoped locks once the locking and unlocking function calls are inlined. Lexical scoping of locking mechanisms can be verified by inspection, analysis, or by knowing the code generator. Leveraging the lexical scoping of locks greatly increases the precision of the lockset analysis.

3.5 External calls
The analysis provides a mechanism for dealing with functions not defined inside the scope of the analysis, such as library functions. The user may specify a list of functions that overwrite only their arguments which the analysis then considers safe. The analysis accepts another user-provided list of functions which do not
affect data-flow state at all, called pure functions. The analysis also interprets the behavior of common library functions. The analysis labels external functions which are not safe, pure, or interpreted as unsafe and treats them pessimistically. The analysis forces all variables on the stack that have had their addresses taken to \( \bot \) on analyzing an unsafe external call.

External functions may call back into application code. For this reason, any time a function pointer is passed to an external function, any function that pointer could point to is considered called at that point as well. Because these potential function pointer calls actually would occur in an external function, they are called with the same state an external function returns: all variables with address taken are \( \bot \).

3.6 Inline assembly

The analysis handles GCC’s inline assembly extension. For example, consider the following code:

```plaintext
asm("eor %1,%2": "=r" (t0): "r" (t1), "r" (t2));
```

Colons delimit the fields “code,” “outputs,” “inputs,” and “clobbers” (clobbers is optional and is absent in this example). The only field that concerns the analysis is “outputs”—a list of C variables to which the compiler should arrange for the results of the assembly code to be written. The analysis handles this by forcing these variables to \( \bot \). In this example t0 would be killed.

3.7 Soundness

The analysis returns sound results under certain assumptions about the memory model. The C language does not define the relative locations of unrelated variables in memory. The analysis described in this chapter, along with all other C analysis tools that I know of, will return invalid results for programs that access undefined out-of-bounds memory. Since the analysis targets heap-free embedded systems, it does not attempt to model the heap. The analysis assumes concurrent flows (interrupts) must not access variables on each others’ stacks. This behavior is also undefined by the C language.

The analysis assumes an implicit order of evaluation for function calls and expressions. The C standard does not specify the order of evaluation of arguments to function calls and allows compiler writers to choose instead of mandating a behavior. Similarly, C also does not specify the order in which subexpressions of a larger C expression can be evaluated. A sound analysis of side-effecting subexpressions or function arguments would take into account all possible orders of evaluation, or all specifications. Because this is both pessimistic and computationally expensive, the analysis instead takes the approach of emulating GCC’s order of evaluation. Of course, well-behaved C programs do not depend on a particular order of evaluation.

4. IMPLEMENTATION

CXprop implements the analysis to analyze C while making it easy for developers to plug in new value propagation domains. The results of the analysis may be output or used to perform source-to-source transformations. CXprop is built on top of CIL [Necula et al. 2002].
4.1 CXprop

Figure 7 describes the structure of CXprop, its relationship with CIL, and its relationship with abstract domains. Lines in the diagram represent well-defined functional interfaces; modules can be easily replaced. The CXprop core:

- preprocesses CIL data structures (for example to assign a unique identifier to each program variable),
- performs a topological sort of the program points,
- performs the fixed point computation,
- passes obtained information to program transformations, and
- pretty-prints the transformed program with some statistics.

The CXprop core uses four other modules. CXprop’s call graph module keeps track of a whole-program call graph, including edges corresponding to function pointers, and gathers other function-level information. ICD operates in the concurrency analyzer and interacts with the CXprop core through a narrow interface. This interface allows developers to investigate more complex and precise analyses as they are developed. The CXprop core passes its information to a collection of transformations. The final module that the core uses is a symbolic executor. It manages abstract values in the program state and performs operations on them. To do this it calls out to the user-supplied abstract domain using the abstract domain interface. The symbolic executor also keeps track of points-to sets.

Figure 8 shows how various analyses interact. There are several supporting transformations (inliner, cleaner, struct splitter) which operate on the source code before passing it to the fixed point computation. In order to exploit potential synergy, several analyses then run at the same time in the same fixed point loop. After a
fixed point is reached, the analysis results then get used in different ways depending on the specified output transformations.

4.2 Data-flow representation

CXprop uses a dense data-flow representation. Conceptually, the analysis stores every variable’s abstract value at every program point. The dense representation allows for several optimizations to reduce time and space usage. First, CXprop only maintains $\top$ states at program point granularity instead of for each variable at each program point. This allows the domain implementations to have an implicit top, which simplifies the development of transfer functions. If a program state is $\top$ then every variable is $\top$ at that program state. Similarly, if a program state is not $\top$ then none of the variables are $\top$. This optimization permits CXprop to store $\bot$ implicitly in the machine state: any variable appearing in a non-$\top$ program point but not explicitly represented is $\bot$. Although these optimizations help, greater performance gains could be achieved by moving to a sparse data representation, such as SSA [Cytron et al. 1991].

4.3 Transformations

The primary transformations performed by CXprop are constant propagation of scalars and pointers, indirection elimination, dead data elimination, and dead code elimination. Although GCC already implements these transformations, CXprop performs more transformations due to its increased precision. The analysis results from CXprop are used in several other ways.
Optimizing code. Type and memory safety promote deterministic behavior and prevent silent RAM corruption in an application. Many memory safety techniques require using dynamic checks, which expend precious resources. TinyOS [Hill et al. 2000] is not type or memory safe. I worked with several others to apply CCured [Condit et al. 2003] to TinyOS in order to introduce type and memory safety [Regehr et al. 2006]. We then used cXprop to mitigate the extensive resource cost of CCured through optimization. In the second version of this work, we applied Deputy [Condit et al. 2007] to TinyOS 2 [Levis et al. 2005] and again leveraged cXprop’s optimizations [Cooprider et al. 2007].

Redundant synchronization elimination. Nested atomic sections occur in TinyOS applications and cXprop removes identifiably unnecessary ones. The nesC compiler already removes lexically nested atomic sections, but this leaves two scenarios of redundant atomic sections that cXprop’s interprocedural analysis may identify for removal. First, a conservative programmer will protect a potentially racing variable in a function he or she writes with an atomic section, but if that function is always called with the lock already held then he or she has introduced an unnecessary atomic section. Second, redundant synchronization occurs often with inlining. The function needs the lock, so the inlined version may need to keep the synchronization in some contexts, but not in others where the lock is already held. Locks for TinyOS are implemented by turning the interrupt bit off, so when an atomic section occurs and the interrupt bit is already off, cXprop removes the atomic section. This saves six instructions per synchronization (on the ATmega128L MCU) and removes a local variable since it is no longer needed to store the previous state of the interrupt bit.

Visibility. CXprop may be used to enforce visibility [Regehr et al. 2006], although it is not the most efficient solution to this problem. Visibility is a property of a programming language’s memory model that determines when values stored by one concurrent computation become visible to other computations. CXprop enforce this property by adding the volatile annotations that programmers tend to forget: it marks all asynchronous variables as volatile in the generated C code, and any pointers used to access them as pointer-to-volatile. CXprop identifies which variables fall into these categories. Visibility is therefore ensured because all loads and stores to these variables are forced to go to RAM. Memory barriers are more efficient for implementing visibility, but CXprop’s analysis and transformation provided an early proof of concept.

RAM compression. CComp extends cXprop and implements offline RAM compression [Cooprider and Regehr 2007], an automated source-to-source transformation that reduces a program’s data size. CXprop’s analysis identifies statically allocated scalars, pointers, structures, and arrays which may be encoded and packed. On a collection of embedded applications for AVR microcontrollers, the transformation reduces RAM usage by an average of 12%, in addition to a 10% reduction through a dead data elimination pass that is also driven by the analysis, for a total RAM savings of 22%.

ACM Journal Name, Vol. V, No. N, Month 20YY.
4.4 Independent passes

CXprop includes three transformations that do not rely on the main data-flow analysis. Each may execute independently of each other and of the fixed point analysis.

**Inliner.** CXprop is context insensitive. Context insensitivity makes function calls a source of imprecision. There are two approaches to dealing with this problem. Either CXprop must be made context-sensitive or the tool must reduce the number of function calls. CXprop currently take the second approach by using an inliner. Will Archer wrote an inliner for CXprop which has two modes of operation. It can either follow inlining directives specified in the code by attributes, or it can follow a heuristic to make its own inlining decisions. The first mode is appropriate for code where nobody has experimented and tuned the heuristic. In that case CXprop needs to trust the programmer. However, CXprop’s inlining heuristic may be trained by building applications with various degrees of inlining and comparing their code sizes. When a trained heuristic is available, CXprop should use it. A trained heuristic takes into account the effects of CXprop after inlining.

**Splitter.** CXprop includes an extension which performs the scalar replacement of global aggregates. CXprop analyzes individual struct fields automatically, but some transformations will not work on structures. The splitter makes the data in some global structs available for later transformations.

**Cleaner.** While CXprop performs aggressive data-flow analysis and whole-program optimizations resulting from that analysis, several transformations exist which do not require the rigorous inter-procedural data-flow analysis. Some of these transformations are peephole optimizations, some of them are more substantial. In order to perform these transformations which are independent of the main data-flow analysis, CXprop includes a source code cleaner. The cleaner is not designed to implement new techniques and is not designed to be fast.

The cleaner runs in several phases. First, CXprop runs intra-procedural parts of the cleaner as part of the inliner in order to more accurately guess the function sizes. Second, CXprop runs the full cleaner prior to its own analysis in order to simplify the code and make it easier to analyze. Third, CXprop runs the full cleaner after its analysis in order to reduce the resulting program.

The cleaner leverages optimization passes already included in the CIL distribution [Anderson 2007; Necula et al. 2002] and then supplements them with additional optimizations.

The cleaner performs several peephole optimizations:

—When the condition of an if-statement becomes constant as a result of propagation (or for some other reason), the branch is changed to straight-line code as long as there are no jumps into the dead branch.

—One side-effect of converting C to CIL is a plethora of brackets. This is due to CIL grouping statements into “blocks.” While CIL provides a method for reducing the blocks, the cleaner is much more aggressive. This results in code that is easier to read and understand.
—The cleaner performs some flow-of-control and straightening optimizations to eliminate unnecessary gotos.

—The cleaner optimizes switch-statements. These are slightly more tricky than if-statements because paths are not mutually exclusive. A program may choose to let a case fall through to the next case. The default case is troublesome as well since it is optional and may cover multiple conditions. The cleaner conservatively deals with these tricky cases. It removes dead cases, turns switches with a constant argument into straight-line code, and reduces two- or one-case switches to if-statements.

The cleaner performs two more substantial optimizations: copy propagation and unused variable elimination. The copy propagator uses CIL’s available expression extension to perform the necessary intra-procedural analysis. The unused variable elimination is a brute-force, flow-insensitive, and interprocedural approach. The cleaner looks for variable reads anywhere in the program and only removes a variable if it is never read.

5. ABSTRACT DOMAINS

CXprop supports conditional X propagation and is configured at compile time with an abstract domain. This section describes five domains included with cXprop, along with some of their advantages and disadvantages.

5.1 Constants

The lattice for constant propagation is shown on the left side of Figure 4. The transfer functions for this domain were straightforward to implement since many of them could exploit CIL’s existing constant-folding routines. Each transfer function in this domain has three basic parts:

—optionally implement a few special cases to increase precision,
—handle $\perp$ values in the inputs, and
—call out to CIL’s constant folder if both inputs are constant.

The transfer function for multiply in Figure 9 is typical. The first four patterns match special cases. The next two patterns cover handle $\perp$ values as input. The final case calls CIL’s constant folder when both arguments are constant.

Although not as precise as some of the other domains in this section, the balance between speed and precision makes it attractive for use in data-flow analysis. Storing the abstract value is relatively inexpensive, and transfer functions operating on these values are straightforward. The lattice for the domain is very shallow which results in analyses quickly arriving at fixed point.

5.2 Value set

In the value set domain, an abstract value is a set of arbitrary concrete values up to some user-defined maximum size. A value set lattice is shown in Figure 10.

Given CIL’s constant folder, a brute-force implementation of value set is not too difficult. CXprop includes a higher-order function apply_binop that takes as input two abstract values (each represented by an OCaml set) and a function for applying the concrete operation (e.g., addition) to a pair of concrete values. apply_binop
let mult d1 d2 tp =
match d1, d2 with
  Constant(z), _ (* 1 *)
| _, Constant(z) (* 2 *)
  when (isZero z) -> Constant (zero)
| Constant(z), other (* 3 *)
| other, Constant(z) (* 4 *)
  when (isInteger z = Some Int64.one) -> other
| Bottom, _ (* 5 *)
| _ , Bottom -> Bottom (* 6 *)
| Constant(e1), Constant(e2) -> (* 7 *)
  conc_to_abs (BinOp(Mult,e1,e2,tp))

Fig. 9. Transfer function for integer multiplication for the constant propagation domain

Fig. 10. An example lattice for the value set domain. The concrete domain in this example is limited to the integers from zero to three

performs a two-dimensional fold on the input sets, using the concrete operation as the folding function. The union of the two concretization operations form a set. The quadratic nature of this implementation implies that large value sets will result in very slow analysis. The upper-bound on set size serves as a widening function for the domain.

The value set domain is currently the preferred domain for cXprop. Transfer functions in this domain may be written by hand with a reasonable amount of effort. Widening can limit the analysis time, although this domain still takes significantly longer than the constant domain.

5.3 Bitwise

In the bitwise domain, shown in Figure 11, abstract values are vectors of three-valued bits: each bit is zero, one, or unknown. In contrast with the constant propagation and value set domains, bitwise transfer functions cannot be easily implemented in terms of CIL’s constant folder. A previous experience with the bitwise domain [Regehr et al. 2005] shows that complicated hand-written transfer functions are a mistake: they are tedious and error-prone to implement without sacrificing a significant amount of precision. Consequently the bitwise transfer functions used by cXprop are automatically derived using a toolchain developed
by Regehr and Duongsaa [Regehr and Duongsaa 2006]. These transfer functions are correct by construction and for most operations they are maximally precise within the constraints of the domain. Given these functions, just a few remaining operations such as casts and a join operator had to be implemented by hand.

The bitwise domain appears, at least at first, to be a good match for analyzing low-level code. Individual bits of device registers may be tracked accurately by the domain. However, storing the bit arrays and executing transfer functions is expensive. Some of the transfer functions are impossible to reason about meaningfully. Even for system code, the underlying bit representation is not exposed sufficiently at the source level to analyze.

5.4 Interval

The interval domain, shown in Figure 12, is a commonly used value propagation domain. The interval transfer functions are automatically derived by the same tool that produces the bitwise transfer functions [Regehr and Duongsaa 2006]. They are also maximally precise, and, unlike many interval implementations, they are correct in the presence of integer overflow. The derivation tool currently supports only unsigned intervals. This domain can be used to soundly analyze signed integers, but it cannot precisely analyze variables that may be both negative and positive. Since the interval lattice has height exponential in the bitwidth of the data type, the fixed point computation can take a long time. To accelerate termination, cXprop widens intervals above a user-defined width to bottom.

Many compilers use the interval domain as the next domain after the constant domain. The representation of values is simple and many of the transfer functions are straightforward, except for overflow. Unfortunately, dealing with overflow is a significant task. The implementation for cXprop results in longer running times without a proportional gain in precision.

5.5 Multi-interval

Domains may be combined for increased precision. Kevin Atkinson wrote the multi-interval domain that combines the value set and interval domains. An abstract value in this domain consists of sets of intervals. Transfer functions are derived by combining the transfer functions from the parent domains. Combining the domains does increase the precision, but only by a tiny amount. It also increases the analysis time by a large amount.

6. EVALUATION

We evaluate cXprop on TinyOS applications. TinyOS [Hill et al. 2000; Levis et al. 2005] is a component-based operating system for sensor network nodes. Components are written in nesC [Gay et al. 2003], a C dialect that the nesC compiler translates into C. cXprop may process this C output before the code passes on to the C compiler.

Programming an application in TinyOS entails connecting components together through narrow interfaces. While this is a convenient programming model, black-box reuse often leads applications to include dead code and data. Although the nesC compiler eliminates dead functions and gcc performs intraprocedural dead code removal at compile time, this is often not enough.
elimination, significant room for improvement based on interprocedural analysis still exists.

TinyOS uses a restrictive concurrency model that cXprop exploits. Most code runs in non-preemptively scheduled tasks. Interrupts may preempt tasks (and each other), but not during atomic sections. TinyOS implements atomic sections by disabling interrupts, thus the interrupt bit becomes a lock. The nesC compiler emits a warning when any global variable that can be touched by an interrupt handler is accessed outside of an atomic section, but this analysis does not take aliasing into account.

Our implementation recognizes the interrupt bit manipulation idioms of two underlying hardware platforms: the AVR-based Mica2 motes from Crossbow [Crossbow Technology, Inc.] and the MSP430-based TelosB motes from Moteiv [Moteiv Corporation].

6.1 A domain-independent precision metric

Pluggable domains make it tricky to measure cXprop’s analysis precision in a generic way. I developed a metric inspired by information theory: the fraction of information known about a program. This metric supports apples-to-apples comparisons of analysis precision across multiple abstract domains.

Each program variable has a fixed number of possible values, defined by its underlying bitwidth. For example, on the x86 architecture an integer is 32 bits and can represent $2^{32}$ different values. A value propagation analysis, such as cXprop, attempts to restrict the number of possible values for each variable. For example, if an interval-domain analysis determines that $2 \leq y \leq 9$ and $y$ is an integer, then it knows $y$ has eight possible values and $2^{32} - 8$ impossible values. The number of possible values for an abstract value is simply the cardinality of its concretization set.

The unit of measurement for information known is the information bit. I define the number of information bits for an abstract value $a$ as follows:

$$
\text{# of information bits} \overset{\text{def}}{=} \log_2 |\gamma(\bot)| - \log_2 |\gamma(a)|
$$

where $\gamma$ is the domain’s concretization function.

In the example above, $32 - 3 = 29$ information bits are known about $y$. If an analysis can show that a variable is constant, it has only one possible value, and therefore all of its bits are known. If an analysis can only conclude that a variable is $\bot$, then the number of possible values is the same as the number of natively representable values, and consequently zero bits of information are known.

The distinction between information bits and physical bits is important. First, information bits can take non-integer values. Second, information bits need not correspond directly to physical bits. For example, seven bits of information are known about an eight-bit integer if, in all executions, it can only hold the values 0xaa or 0xbb.

The definition above makes it possible to compute the amount of information known about a variable at a program point. Next I extend this metric to cover entire programs. My first attempt to do this was inspired by SSA [Cytron et al. 1991]; it examined every static assignment to every variable. However, I found
this method too dependent on program implementation details. A program may use variables to store intermediate values instead of computing a large expression through one assignment. For this reason I moved to an alternative method of computing information known for an entire program.

I define the total information known about a program to be the information known about all global variables when all their abstract values are merged together. Total information known uses global variables because they are not as affected by arbitrary code idioms and CIL transformations. A primary goal of interprocedural analysis is to obtain information about global variables. The following code illustrates total information known:

```plaintext
1: int x=0;
2: int y=0;

3: int foo (int z) {
4:    z++;
5:    if (x) {
6:        z=0;
7:    } else {
8:        z=1;
9:    }
10:    return z;
11: }

12: int main () {
13:    int a;
14:    y = a;
15:    x++;
16:    x=foo(x);
17:    return x + y;
18: }
```

This program has two global variable and I will assume the compiler targets a machine with 32-bit ints. After using the value set domain to perform an analysis of this program, cXprop determines x has a global value set of \{0,1\}. In other words, the only values x has anywhere in the program are 0 and 1. Only one bit is necessary to distinguish between two values, so only one bit is unknown. On the other hand, y is assigned an uninitialized local variable, so its global value is completely unknown. This means that for the entire program 31 of the 64 global bits are known, or 48%.

6.2 Validation

I used three techniques to assist with validating the analysis results of cXprop.

*Dynamic checking.* CXprop may add assert statements to abort execution if "dead" code is executed or if a variable contains a value outside of its analyzed abstract value. For example, if a variable z has been shown to have the interval value [2..15] at a program point, cXprop would insert the following code:

```plaintext
assert (z >= 2 && z <= 15);
```

ACM Journal Name, Vol. V, No. N, Month 20YY.
My experience has been that developing correct abstract domains is difficult without this kind of aggressive safety checking.

Dynamic data-flow. CXprop may transform the original program to gather dynamic data-flow information. When this program runs, every concrete value stored to a variable is lifted into the current abstract domain and merged with a stored abstract value for that variable and program point. Let $x_d$ be the dynamically computed abstract value for variable $x$ at a program point, $x_s$ be the statically computed abstract value at that point, and $x_i$ be the incomputable “ideal” abstract value that is the largest abstract value that soundly approximates the value of the variable over all possible executions. As long as CXprop is correct, I am guaranteed that the following lattice inequalities hold:

$$x_d \sqsubseteq x_i \sqsubseteq x_s$$

I exploit these relations to validate CXprop.

Testing. I frequently stress-tested CXprop using a random program generator [Turner 2005]. The tool successfully analyzed and transformed the vast majority of these randomly generated programs. CXprop did change the behavior of some randomly generated programs, but only those that relied on undefined behavior, for example by falling out of a function that is supposed to return a value or by relying on a particular order of evaluation of subexpressions.

Debugging CXprop is greatly simplified by Delta [Wilkerson 2003], an implementation of the Delta debugging algorithm [Zeller and Hildebrandt 2002]. This tool mechanically reduces the size of a program while preserving a user-defined “interestingness” criterion. I define a program to be interesting when it compiles correctly and also causes CXprop operating in assert mode to reach an assertion failure.

6.3 Benchmarks

I evaluate CXprop on TinyOS 2 [Levis et al. 2005] applications. Table 1 shows the TinyOS applications I use for benchmarks. I use Avrora [Titzer et al. 2005] to obtain duty cycle measurements. I use stack-estimator to obtain conservative stack usage estimates [McCartney and Sridhar 2006].

6.4 Widening decisions

The interval, value set and pointer-set domains are all parameterized by a maximum size for widening. Rather than choosing the widening functions based on limited experience and guesswork, CXprop actually finds the best maximum sizes for widening through comparison across multiple runs. I compile my benchmarks while varying one of the widening parameters. I plot the information known, code size, and analysis time in Figure 13.

CXprop uses a maximum set size of 16 to widen the value set domain and a maximum set size of four to widen the interval domain. After that point the analysis times continue to lengthen without providing additional precision or code size reduction. CXprop uses a maximum set size of 32 for our points-to sets. The pointer-to set size was kept at 32 when computing the value set and interval numbers, and the value set size was kept at 16 when computing the points-to set numbers.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>nesC files</th>
<th>nesC loc</th>
<th>C loc</th>
<th>ROM (bytes)</th>
<th>static RAM (bytes)</th>
<th>stack RAM (bytes)</th>
<th>duty cycle (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AntiTheft/Nodes</td>
<td>198</td>
<td>19927</td>
<td>25815</td>
<td>24222</td>
<td>1565</td>
<td>212</td>
<td>3.63</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>detects and reports theft of a node</td>
</tr>
<tr>
<td>AntiTheft/Root</td>
<td>191</td>
<td>21039</td>
<td>22562</td>
<td>22452</td>
<td>1537</td>
<td>214</td>
<td>3.41</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>detects and reports theft of a node</td>
</tr>
<tr>
<td>BaseStation</td>
<td>142</td>
<td>14675</td>
<td>13976</td>
<td>12862</td>
<td>1442</td>
<td>214</td>
<td>8.86</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Active Message bridge between the serial and radio links</td>
</tr>
<tr>
<td>MultihopOscilloscope</td>
<td>182</td>
<td>19973</td>
<td>22628</td>
<td>23500</td>
<td>3006</td>
<td>213</td>
<td>9.32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>periodically samples the default sensor and broadcasts a message every few readings</td>
</tr>
<tr>
<td>MViz</td>
<td>178</td>
<td>19658</td>
<td>21516</td>
<td>22508</td>
<td>1542</td>
<td>214</td>
<td>8.99</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>multihop collection network</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>121</td>
<td>11740</td>
<td>11019</td>
<td>10448</td>
<td>248</td>
<td>150</td>
<td>8.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>periodically samples the default sensor and broadcasts a message over the radio every 10 readings</td>
</tr>
<tr>
<td>RadioCountToLeds</td>
<td>117</td>
<td>11586</td>
<td>10674</td>
<td>9866</td>
<td>220</td>
<td>151</td>
<td>9.32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>broadcasts a 4 Hz counter’s value in an AM packet every time it gets updated and displays received packets on the LEDs</td>
</tr>
<tr>
<td>RadioSenseToLeds</td>
<td>121</td>
<td>11740</td>
<td>10974</td>
<td>10098</td>
<td>218</td>
<td>150</td>
<td>9.40</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>samples a platform’s default sensor at 4Hz and broadcasts this value in an AM packet while displaying received packets on the LEDs</td>
</tr>
</tbody>
</table>

Table 1. The TinyOS 2 applications I use for my benchmarks and some statistics about them

6.5 Domain comparison

**Precision.** Figure 14 compares the total information known for various benchmarks when analyzed using different abstract domains. The constant domain performs uniformly worse in terms of precision when compared to the other domains shown in the graph. This makes sense because each of the other domains subsumes the constant domain. Similarly, the multi-interval domain subsumes the value set and interval domains, so it consistently obtains the greatest precision. The value set regularly achieves high precision, but the results exaggerate its benefits. After initially discovering its favorable precision-to-analysis-time tradeoff, I ceased improving the other domains. This is probably the most significant reason for its apparent superiority in the graph.

**Code size.** Figure 15 compares the code size reduction of the resulting executables after using different pluggable abstract domains in cXprop. The graph shows similar results to Figure 14 with a slightly smaller magnitude in the differences.

**Analysis time.** The choice of abstract domain influences analysis time in two ways. The value set, bitwise, interval, and multi-interval lattices are all much taller than the constant domain. It therefore takes longer for values to descend the lattice to their fixed points. The more complicated domains also have more complicated transfer functions, which are slower. Table 2 shows the time in seconds it takes for cXprop to complete with different domains.

The geometric mean of the ratio between value set and constant analysis times is
Fig. 13. Varying the value set, interval, and pointer-set to see their effects on precision, code size, and analysis time.

Fig. 14. Percentage of information discovered by cXprop using different abstract domains.

ACM Journal Name, Vol. V, No. N, Month 20YY.
code elimination. GCC’s optimization passes are particularly effective since TinyOS already eliminates dead functions and GCC already performs intraprocedural dead-code elimination. GCC’s optimization passes are particularly effective since TinyOS applications are aggressively inlined, eliminating around 90% of static function calls in many applications. The benefit from cXprop comes from residual opportunities.

2.3. The same computation for the bitwise domain takes only 1.2 times as long. The interval domain takes 1.9 times as long as the constant domain. The multi-interval domain takes over four times as long.

Because of these dependencies, the time it takes cXprop to run with the value set domain varies from a couple seconds for Blink to almost half an hour MultihopOscilloscope. cXprop’s performance is limited by its dense data-flow representation, by the fact that it tracks global variables (which necessitates pushing around very large machine states for some programs), by ICD (which adds many implicit flow edges), and by the tracking of individual struct fields. Also, my implementation is a research prototype; its performance has not been tuned for speed.

6.6 Resource usage reduction

Figure 16 presents the results of using cXprop’s value set domain to reduce resource use for the benchmarks.

Reducing code size. The average code size reduction for the value set domain is 12%. I believe these results are good, given the following factors. First, nesC already eliminates dead functions and GCC already performs intraprocedural dead-code elimination. GCC’s optimization passes are particularly effective since TinyOS applications are aggressively inlined, eliminating around 90% of static function calls in many applications. The benefit from cXprop comes from residual opportunities.

Table 2. Time in seconds for cXprop to complete

<table>
<thead>
<tr>
<th>benchmark</th>
<th>constant</th>
<th>value set</th>
<th>bitwise</th>
<th>interval</th>
<th>multi-interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>RadioCountToLeds</td>
<td>38.01</td>
<td>69.37</td>
<td>42.37</td>
<td>61.82</td>
<td>134.89</td>
</tr>
<tr>
<td>RadioSenseToLeds</td>
<td>40.26</td>
<td>74.89</td>
<td>44.96</td>
<td>64.35</td>
<td>147.48</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>35.60</td>
<td>76.22</td>
<td>41.61</td>
<td>64.49</td>
<td>156.92</td>
</tr>
<tr>
<td>BaseStation</td>
<td>101.31</td>
<td>433.54</td>
<td>138.72</td>
<td>294.42</td>
<td>785.12</td>
</tr>
<tr>
<td>AntiTheft_Nodes</td>
<td>421.62</td>
<td>827.67</td>
<td>404.32</td>
<td>717.47</td>
<td>1836.91</td>
</tr>
<tr>
<td>MViz</td>
<td>407.41</td>
<td>874.17</td>
<td>525.12</td>
<td>643.74</td>
<td>1568.66</td>
</tr>
<tr>
<td>AntiTheft_Root</td>
<td>493.02</td>
<td>1034.50</td>
<td>625.77</td>
<td>1113.15</td>
<td>1953.62</td>
</tr>
<tr>
<td>MultihopOscilloscope</td>
<td>750.62</td>
<td>2141.50</td>
<td>1020.42</td>
<td>1666.11</td>
<td>3380.17</td>
</tr>
<tr>
<td>Average</td>
<td>285.98</td>
<td>691.48</td>
<td>355.41</td>
<td>578.19</td>
<td>1245.47</td>
</tr>
</tbody>
</table>
for interprocedural analysis. Second, minor code transformations implemented by CIL, such as the introduction of temporary variables, handicap cXprop’s efforts by increasing the size of the generated code by several percent. A number of the peephole optimizations implemented in the cleaner (Section 4.4) undo these transformations when CIL pretty-prints a C file. These partially offset CIL’s code size penalty.

Reducing duty cycle. The average duty cycle reduction for the value set domain is 8.3%. A savings in the duty cycle indicates the microcontroller sleeps more often. A sleeping microcontroller uses less energy, thus increasing the lifetime of the mote. Increasing mote lifetime is important for applications designed to run for months on AA batteries.

Reducing RAM usage. For the value set domain, the average reduction in estimated maximum stack size is 1.8% and the average static data size reduction is 2.5%. I attribute these relatively small savings to the tight coding of TinyOS 2 and the type of information learned by the analysis. Experience in earlier versions of TinyOS pointed to RAM usage as a limiting factor, so developers took great care in TinyOS 2 to conserve RAM. This leaves less dead data for cXprop to eliminate. Also, cXprop may determine that a variable only uses a limited number of information bits, but unless it is actually constant or dead then cXprop’s default transformations cannot optimize the variable. Also, interprocedural information learned by cXprop often can be learned by GCC too when analyzing heavily inlined applications such as those created by TinyOS 2.

6.7 Analysis feature comparison

I compare various aspects of my analysis by turning off features. I then examine the resulting executables of the handicapped analyses. The names given for describing handicaps are explained below.

—cXprop — No handicapping features
—Address taken — Using an address taken analysis instead of a points-to analysis
—No concurrency — Do not use ICD. No data-flow analyzed through global variables
Fig. 17. Reduction in code size from optimizations with certain analysis features turned off

—No volatiles — No data-flow analyzed through volatile variables
—No arrays — No data-flow analyzed through arrays
—No assume nesC — Do not assume lexical scoping for locks

Figure 17 shows the code size comparison of handicapped analyses. Turning off the analysis of all global variables and not assuming nesC scoping for locks are the two handicaps with the greatest effects on code size transformation. Many of the analysis improvements over GCC disappear when cXprop analyzes only local variables. One of the primary goals of cXprop is to analyze global variables, so taking away that ability has the expected result of reducing the amount of code found to be dead. On a related note, ICD plays a vital role in the analysis of global data, and tracking the interrupt bit plays a vital role in ICD. Not leveraging the lexical scoping of locks reduces the analysis’ ability to track the interrupt bit, which pessimizes ICD. This results in extra control flow edges and more racing variables.

Figure 18 shows the comparison of information known for the various handicapped analyses. This graph shows similar patterns of behavior for information known by the analyses, except for the “no assume nesC” option. Taking into account the lexical scoping of locks improves the analysis of local data. Since the whole-program view of the metric emphasizes global data, ignoring the lexical scoping does not hurt the metric much. The metric merges all points of the program together, which naturally introduces some pessimism.

Table 3 holds the running time in seconds of the various handicapped analyses on the benchmarks. This data shows that handicapping the analysis can increase the speed of the analysis. There exists a tradeoff between analysis time and precision.

A few of the handicaps did not have as negative of an effect as I initially suspected. Not tracking information flow through arrays does not impact code size significantly because many arrays act as buffers to store input information. This is also the reason using an address-taken analysis instead of the points-to analysis has such a small effect on the analysis of TinyOS 2 code. The aliased objects tend to hold input information which cannot be known by the analysis and is therefore \( \perp \) anyway. Finally, the benchmark TinyOS 2 applications use only three volatile variables—a task queue, an index to the head of the task queue, and an index to the tail of the task queue. The information obtained from these variables by the analysis does not impact dead code elimination for these applications.
7. RELATED WORK

This article builds on our initial work with dataflow analysis of MCU based systems [Cooprrider and Regehr 2006].

A vast amount of research on data-flow analysis exists; we therefore discuss and cite only the most related work here.

Analysis of concurrent software. Previous work on data-flow analysis for thread-based systems [Dwyer et al. 2004; Rinard 2001] may be ported to analyze interrupt-driven systems [Regehr and Cooprrider 2007], but new concurrency abstractions are necessary to analyze interrupt-driven MCU code with adequate precision for Safe TinyOS and RAM compression. ICD embodies these new abstractions, as detailed in Section 2.

Feng et al. introduce a formal semantics for program certification which encompasses interrupts [Feng et al. 2008]. This semantics is used to certify a preemptive thread implementation and provides a formal foundation for reasoning about interrupt-based software. Their certification required two days of interactive work.
by an experienced user. This is reasonable for certification, but not for inclusion in a compiler. ICD could be implemented in a compiler.

RADAR [Chugh et al. 2008] automatically converts a sequential data-flow analysis into a concurrent analysis. It does this by leveraging a race detection engine. RADAR factors all reasoning about concurrency into the race detector. This provides simplicity and generality, but sacrifices some precision. Because ICD considers interleavings in addition to racing data, it can generate data-flow facts from concurrent flows. RADAR can only remove data-flow facts for variables it determines are racing. RADAR is a general tool for concurrent analysis, whereas ICD targets interrupt-driven MCU code.

Several static analysis tools for interrupt-driven software deal with concurrency to some degree. nesC [Gay et al. 2003] provides a race detector for TinyOS applications. The detector ignores aliasing in order to achieve faster compilation times. Palsberg et al. [Palsberg and Ma 2002] introduce a calculus of threadless interrupt-driven systems for determining stack usage. Stacktool [Regehr et al. 2005] extends that work. Stacktool targets AVR assembly code and adds interrupt flow edges after every assembly instruction where interrupts are enabled. Stacktool does not track program memory, focusing instead on registers relevant to stack use. cX-prop targets C, uses ICD to reduce added flow edges due to interrupts, and tracks values stored in program memory. Several model checkers exist which deal with interrupt-driven concurrency [Ball et al. 2006; Engler and Ashcraft 2003; Henzinger et al. 2003; Mercer and Jones 2005], but ICD avoids model checking and focuses on compiler techniques.

Volatile. I am not aware of other work which tracks data-flow through volatile data. I initially implemented the tracking of volatile data to support RAM compression [Cooprider and Regehr 2007].

Pluggable abstract domains. Generalized constant propagation [Verbrugge et al. 1996] extends constant propagation to handle intervals. “conditional X propagation” can be considered to be a generalization of generalized constant propagation.

PAG [Martin 1998] automatically generates a data-flow analyzer based on inputs written in domain-specific languages describing the domain lattice, the transfer functions, a language-describing grammar, and a fixed point solution method. cX-prop is far less configurable, providing only pluggable abstract domains. However, as far as I know, cXprop supports a richer variety of value-propagation domains and techniques for dealing with concurrency than does PAG. Also, while the PAG research evaluated a number of different fixed point algorithms, I have focused on comparing the behavior of different value propagation domains.

An alternative and less automatic approach than PAG is to provide a set of composable pieces in a library [Dwyer and Clarke 1996]. This library uses well-defined interfaces to allow the user to compose transfer functions, lattices, flow graphs, and a fixed point solver. Dwyer and Clarke present a more abstract and theoretical tool, whereas cXprop pragmatically focuses on analyzing embedded software written in C.

Pointer analysis. Abstract values in the value set and pointer set domains are sets of explicit concrete values. Balakrishman and Reps [Balakrishman and Reps
used value set analysis to analyze pointers in executable code, but they use the term differently than I do. Their abstract values are sets of reduced interval congruences—a highly specialized domain tuned to match x86 addressing modes.

Saha and Ramakrishnan [Saha and Ramakrishnan 2005] present a demand-driven points-to analysis. Their analysis focuses on dealing with incremental changes to the program and is context and flow insensitive. Zheng and Rugina present a demand-driven alias analysis which which does not compute points-to sets and is context and flow insensitive [Zheng and Rugina 2008]. cXprop uses demand-driven techniques to deal with context insensitivity instead of aliases. Cxprop avoids a context-sensitive pointer analysis [Emami et al. 1994; Lattner et al. 2007; Wilson and Lam 1995] in order to combine the pointer analysis with its client analyses [Hind 2001].

Information bits. Although my “information bits” evaluation metric for cXprop is novel, inspiration for it comes from the evaluation of the Bitwise compiler [Stephenson et al. 2000], which compares static interval domain results with dynamically gathered data-flow information. Several other techniques [Ananian and Rinard 2003; Budiu et al. 2000; Razdan and Smith 1994] use compiler analyses to reduce bitwidths, but the information bits metric uniquely uncouples the precision from the underlying bit representation. Methods also exist for measuring the precision of transfer functions [Pierro and Wiklicky 2001], but this did not seem to be amenable to the cross-domain comparisons that I wanted to make.

8. CONCLUSIONS

cXprop is a tool for performing analysis and transformation on C programs for interrupt-driven MCUs. It implements several new techniques including a novel concurrency model, data-flow through volatiles, and pluggable abstract domains.

The success of our novel concurrency model shows the importance of using data classifications together with control-flow information when formulating an analysis. Our identification of available information inside volatile qualified data illustrates some short-comings of that qualifier. Pluggable abstract domains demonstrate the utility of domains beyond traditional choices. cXprop reduces TinyOS 2 application code size by an average of 12% and duty cycle by an average of 8.3%.

cXprop is open-source software and may be downloaded from http://www.cs.utah.edu/~coop/research/cxprop/.

ACKNOWLEDGMENTS

We thank the CIL developers and maintainers for their help. We thank Will Archer for developing the inlining pass and Kevin Atkinson for developing the multinterval domain. We also thank Eric Eide, Alastair Reid, and Ben Titzer for their helpful comments on this work. This work is supported by National Science Foundation CAREER Award CNS-0448047.

REFERENCES

32 ·


ACM Journal Name, Vol. V, No. N, Month 20YY.


ACM Journal Name, Vol. V, No. N, Month 20YY.


ACM Journal Name, Vol. V, No. N, Month 20YY.