View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by The University of Utah: J. Willard Marriott Digital Library
THE UNIVERSITY OF UTAH
Author Manuscript
UU IR Author Manuscript

Pluggable Abstract Domains for Analyzing Embedded Software

Nathan Cooprider John Regehr
School of Computing, University of Utah
{coop,regehr}@cs.utah.edu

Abstract

Many abstract value domains such as intervals, bitwise, constants, and value-sets have been developed to support dataflow analysis. Different domains offer alternative tradeoffs between analysis speed and precision. Furthermore, some domains are a better match for certain kinds of code than others. This paper presents the design and implementation of cXprop, an analysis and transformation tool for C that implements "conditional X propagation," a generalization of the well-known conditional constant propagation algorithm where X is an abstract value domain supplied by the user. cXprop is interprocedural, context-insensitive, and achieves reasonable precision on pointer-rich codes. We have applied cXprop to sensor network programs running on TinyOS, in order to reduce code size through interprocedural dead code elimination, and to find limited-bitwidth global variables. Our analysis of global variables is supported by a novel concurrency model for interrupt-driven software. cXprop reduces TinyOS application code size by an average of 9.2% and predicts an average data size reduction of 8.2% through RAM compression.

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

Keywords Abstract interpretation, TinyOS, embedded software

1. Introduction

Conditional constant propagation (CCP) [29] is a well-known program analysis that exploits the synergy between constant propagation and dead code elimination. The synergy exists because propagating constants may result in the discovery of additional dead code and eliminating dead code may reveal additional constants. Conditional constant propagation is widely used because it is efficient and precise: in a single pass it finds at least as many constants and pieces of dead code as would be found by iteratively alternating between constant propagation and dead code elimination until no further benefits were attained by either analysis.

This paper describes cXprop: our tool that implements *conditional X propagation*. Conditional X propagation is a generalization of CCP that permits user-defined value-propagation analyses to be substituted for the constant propagation analysis. So far, in addition to constant propagation, we have implemented parity, intervals, bitwise (abstract values are vectors of three-valued bits), and bounded-size sets of arbitrary values. Our implementation is based on CIL [21] and handles all of C. cXprop is interprocedural and performs whole-program analysis even when the input consists of multiple compilation units. Because cXprop keeps track of points-to sets, it achieves reasonable precision on pointer-rich codes. Implementing a new static analysis in cXprop is straightforward: the user simply writes a collection of OCaml functions conforming to an *abstract domain interface* that we developed.

cXprop can operate in three modes. In the first mode the analysis transforms code in the expected way: dead code is eliminated and constant variables are replaced with literal constants. In the second mode, `assert` statements are added that dynamically validate the soundness of our analysis: execution is aborted if "dead" code is executed or if any variable contains a value outside of its analyzed abstract value. In the third mode, code is transformed in such a way that its execution gathers dynamic dataflow information.

Because cXprop targets embedded software, it must return sound results when analyzing concurrent programs. To this end cXprop supports three models of concurrency. The first avoids modeling global variables: this is sound but imprecise. The other two concurrency models increase analysis precision by exploiting characteristics of interrupt-driven embedded software.

We have used cXprop to analyze embedded and desktop applications. When applied to TinyOS applications, cXprop reduces code size by an average of 9.2% and finds that 8.2% of the space allocated to global variables is unnecessary.

The contributions of this paper are:

- cXprop, a C dataflow analysis supporting pluggable abstract domains,
- two novel concurrency models for analyzing global variables in the presence of interrupts and nesC's `atomic` sections,
- a quantitative comparison of four abstract domains, and
- an evaluation of cXprop as a tool for reducing code size for sensor network applications running on TinyOS.

This paper is organized as follows. Section 2 presents background material. Section 3 describes the design decisions and assumptions behind cXprop. Section 4 describes the cXprop implementation. Section 5 discusses the abstract domains that we have developed for cXprop. In Section 6 we evaluate the analysis and apply it to TinyOS applications. Section 7 compares our work with prior research and Section 8 presents our future work.

2. Background

cXprop is an abstract interpreter [5] that builds on conditional constant propagation and CIL. It targets TinyOS applications, but can handle all of C.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
LCTES'06 June 14–16, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-362-X/06/0006. . . $5.00.

University of Utah Institutional Repository
Author Manuscript

UU IR Author Manuscript

UU IR Author Manuscript

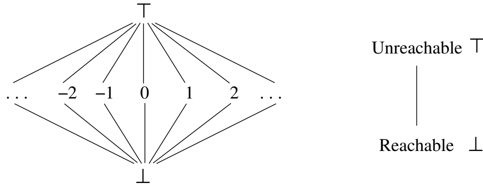

Figure 1. Conditional constant propagation is an analysis combining the constant propagation lattice (left) with the unreachable code elimination lattice (right)

```
 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   }
```

Figure 2. CCP finds line 7 to be dead and `x` to be constant, but iterating CP and DCE does not find either

2.1 Conditional constant propagation

CCP provides constant propagation (CP) and dead code elimination (DCE) in a single integrated analysis pass. Figure 1 shows the two lattices that CCP is based on and Figure 2 shows a program fragment that can be successfully analyzed by CCP but not by iteratively applying constant propagation and dead code elimination. The crux of the problem is that both CP and DCE initially make optimistic assumptions that, respectively, each variable is constant and each branch is dead. However, the optimistic assumption is lost when the analyses are run iteratively—an integrated analysis is required. CCP is widely used. For example, it is implemented in recent versions of gcc.

2.2 CIL

CIL is a parser, typechecker, and intermediate representation for C that greatly simplifies the implementation of analyses and transformations. One of CIL's most useful features is that, in addition to supporting ANSI C, it supports most of the quirks and features of the GNU and Microsoft C dialects. CIL also has the ability to merge a collection of `.c` files together, simplifying whole-program analysis.

2.3 TinyOS

TinyOS is a component-based operating system for sensor network nodes. Components are written in nesC [11], a C dialect that is translated into C by the nesC compiler. This output may be processed by cXprop before being passed 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 elimination, there is still significant room for improvement based on interprocedural analysis.

TinyOS is based on a restrictive concurrency model that is exploited by cXprop. Most code runs in *tasks* that are scheduled non-preemptively. *Interrupts* may preempt tasks (and each other), but not during *atomic* sections. Atomic sections are implemented by disabling interrupts. 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.

3. Design of cXprop

cXprop performs a context-insensitive forward dataflow analysis parameterized by a user-supplied abstract domain. It may then perform one of a number of source-to-source transformations. The three transformations currently supported are conditional constant propagation, assertion-based dynamic validation of the soundness of the static analysis, and dynamic dataflow analysis.

cXprop is designed to analyze C while making it easy for developers to plug in new value propagation domains. cXprop is more aggressive than traditional compiler-based analyses since it propagates values through global variables and structs.

3.1 Pluggable domains

Abstract interpretation [5] provides a general framework for building and reasoning about dataflow analyses. In particular, analyses can be shown to be sound and to terminate for abstract domains that have the following properties:

- The domain's abstraction and concretization functions are monotonic and form a Galois insertion.
- The domain's transfer functions are monotonic and soundly approximate their corresponding concrete operations.
- The domain has no infinite descending chains.

Although analyses require domains with no infinite descending chains, this property can be relaxed by employing a widening function. A widening function jumps to the end of a chain after it has been traversed some pre-determined amount.

Most of an abstract interpreter, such as fixpoint computation and the control flow graph construction logic, is independent of the domain. cXprop makes use of abstract interpretation theory in the sense that domains meeting the above restrictions can be plugged in and used to analyze C code. The domains and cXprop's domain interface are described in Section 5.

When describing abstract domains, we adopt the convention commonly used in the compiler literature [8, 29]. That is, we put the abstract value with the empty concretization set at top ($\top$) of the lattice and the value with the universal concretization set at the bottom ($\bot$).

3.2 Dataflow representation

cXprop uses a dense dataflow representation. Conceptually, every variable's abstract value is stored at every program point. The dense representation allows for several optimizations to reduce time and space usage. First, we only maintain $\top$ states at program point granularity instead of for each variable at each program point. This allows our domain implementations to have an implicit top, which simplifies the development of transfer functions. If a program state is $\top$ then every variable should be $\top$ at that program state. Similarly, if a program state is not $\top$ then none of the variables can be $\top$. This optimization permits us to also 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, we expect that greater performance gains will be achieved by moving to an SSA-based [7] representation. This will be straightforward as it affects only our core dataflow engine, leaving most of the analyzer (in particular, the user-supplied abstract domains) unchanged.

University of Utah Institutional Repository
Author Manuscript

INSTITUTIONAL REPOSITORY
THE UNIVERSITY OF UTAH

UU IR Author Manuscript

UU IR Author Manuscript

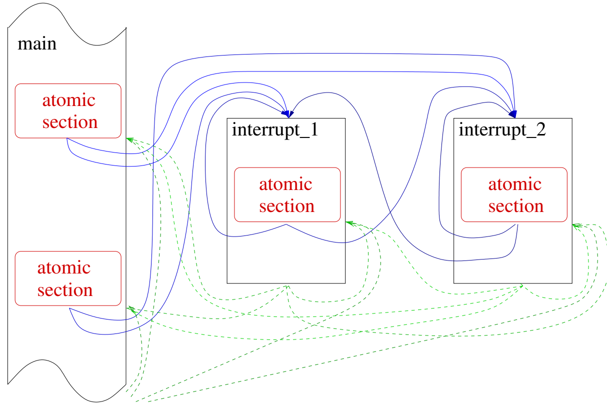

Figure 3. To analyze interrupt-driven software, cXprop adds implicit flow edges to each interrupt handler, and back, at the end of every nesC `atomic` section

3.3 Concurrency

Concurrent execution introduces many implicit control flow edges into a program, making precise dataflow analysis more difficult. cXprop makes two basic assumptions. First, concurrent flows (threads or interrupts) must not access variables on each others' stacks. Second, all entry points (other than `main()`) must be provided to cXprop on the command line.

cXprop classifies all global data as either shared or unshared. Unshared data, which can be shown to be accessed by just one concurrent flow, can be modeled in the standard, sequential manner. Shared data, which may be accessed by more than one concurrent flow, is more complicated. Data accessed by multiple TinyOS tasks is not shared because tasks do not execute concurrently. Rather, data is considered shared when it is accessed by multiple interrupt handlers, or by at least one interrupt handler and at least one task.

cXprop supports three concurrency models. First, cXprop can simply avoid modeling any shared data: it is treated as unknown at all program points. Second, cXprop can track the status of the processor's interrupt mask in order to compute the program points at which interrupts can fire. Regions of the program that run without interference from interrupts are effectively sequential and within them the standard sequential dataflow analysis can be applied. cXprop's implementation of this concurrency model is tailored to two underlying hardware platforms: the AVR-based Mica2 motes from Crossbow [6] and the MSP430-based TelosB motes from Moteiv [20]. It could easily be generalized to handle code for additional platforms.

The third concurrency model supported by cXprop is perhaps the most interesting and useful. It is based on the following observations:

- Code executing within a TinyOS atomic section always runs to completion.
- Code executing outside of TinyOS atomic sections is guaranteed not to touch any shared data except (1) variables marked with the nesC `norace` attribute and (2) variables for which the nesC compiler reports a possible race condition [11].

We exploit the TinyOS/nesC concurrency model in order to obtain more accurate dataflow results. First, global shared variables that are declared `norace` or for which nesC reports a race cannot be modeled outside of atomic sections; cXprop forces them to $\perp$ upon exit from an atomic section. Second, cXprop 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. Figure 3 illustrates this. This forces the analysis to consider all possible interleavings of atomic sections and interrupts. Although this concurrency model is specific to nesC/TinyOS, it could be generalized to work for generic embedded software if we implemented our own race condition detector.

3.4 Soundness and completeness

Precise and sound dataflow analysis of C is nearly impossible. To see this, consider that any store into an array where the index is indeterminate could overwrite any memory cell in the current address space. After each such store a sound analyzer must drop all dataflow facts. In fact, since indirect stores can overwrite saved return addresses on the call stack, it is impossible to even build a sound control flow graph for most C programs. cXprop deals with C's soundness problems in the standard way: by making additional assumptions about program behavior.

Memory model The C language does not make any guarantees about the relative locations of unrelated variables in memory. cXprop, along with all other C analysis tools that we know of, will return invalid results for programs that access out-of-bounds memory. Since we are targeting heap-free embedded systems, cXprop does not attempt to model the heap.

External calls Functions that are not defined inside the scope of the analysis, such as library functions, must be treated pessimistically. For example, cXprop forces all variables on the stack that have had their addresses taken to $\perp$ on analyzing an external call. To increase precision, we permit the user to specify a list of functions that overwrite only their arguments.

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 $\perp$.

Floating point The embedded applications that we target primarily use integers, and hence we do not model floating point operations. In fact, no current TinyOS platform has a hardware floating-point unit. However, given a collection of abstract transfer functions supporting floating point values, we could easily add support for these to our abstract domain interface.

Inline assembly We handle `gcc`'s inline assembly extension. For example, consider the following code:

```
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 cXprop is "outputs"—a list of C variables to which the compiler should arrange for the results of the assembly code to be written. cXprop handles this by forcing these variables to $\perp$. In this example `t0` would be killed.

Order of evaluation The C standard permits compiler writers to choose the order of evaluation of arguments to function calls. Similarly, subexpressions of a larger expression can be evaluated in any order. A sound analysis of side-effecting subexpressions or function arguments would take into account all possible orders of evaluation. Because this is both pessimistic and computationally expensive, we instead take the approach of emulating `gcc`'s order of evaluation. Of course, well-behaved C programs do not depend on a particular order of evaluation.

University of Utah Institutional Repository
Author Manuscript

UU IR Author Manuscript

UU IR Author Manuscript

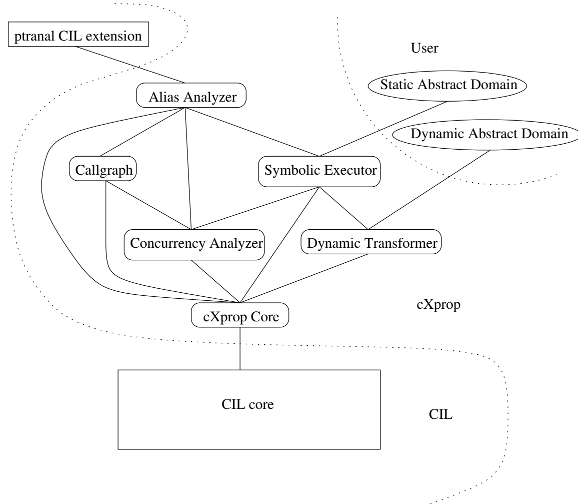

Figure 4. cXprop, its internal structure, and its relationship with CIL and abstract domains

Order of evaluation of function arguments is only a problem when the original program order must be preserved. Since cXprop is a source-to-source transformer, CIL chooses an order of evaluation by introducing temporary variables. The resulting code will be correctly compiled by any C compiler because it no longer gives the compiler the freedom to choose an order of evaluation. The order of evaluation of subexpressions is more difficult to control since it would require the introduction of a very large number of temporary variables. Neither CIL nor cXprop does this.

4. Implementation

cXprop is an abstract interpreter. Figure 4 describes the structure of cXprop and its relationship with CIL and with a given abstract domain. 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,
- transforms the analyzed program, and
- pretty-prints the transformed program and some statistics.

The cXprop core uses information from five other modules. An alias module uses CIL's pointer analysis extension to compute function pointer destinations. cXprop's callgraph module computes a whole-program callgraph, including edges corresponding to function pointers, and gathers other function-level information. Our three concurrency models operate in the concurrency analyzer and interact with the cXprop core through a narrow interface. This interface will allow us to investigate more complex and precise analyses as they are developed. The dynamic transformer uses domain information from an abstract domain specification to model variables dynamically. 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 our abstract domain interface. The symbolic executor also keeps track of points-to sets.

4.1 Program transformations

cXprop can perform three different program transformations. The first replaces constants and eliminates dead code. In the second, `assert` statements are added that 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:

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

Our experience has been that developing correct abstract domains is difficult without this kind of aggressive safety checking.

In the final transformation the original program is changed to gather dynamic dataflow 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 uncomputable "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, we are guaranteed that the following lattice inequalities hold:

$$x_d \sqsupseteq x_i \sqsupseteq x_s$$

We exploited these relations to validate cXprop.

4.2 Validation

We frequently stress-tested cXprop using a random program generator [27]. Our tool successfully analyzes and transforms the vast majority of these randomly generated programs. cXprop does change the behavior of some randomly generated programs, but only those that rely 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 was greatly simplified by Delta [30], an implementation of the Delta debugging algorithm [31]. This tool mechanically reduces the size of a program while preserving a user-defined "interestingness" criterion. We defined a program to be interesting when it compiles correctly and also causes cXprop operating in assert mode to reach an assertion failure.

5. Abstract Domains

At compile time, cXprop is configured with an abstract domain. This section describes the interface for pluggable domains and the five domains that we have implemented.

5.1 The abstract domain interface

Domains are written in OCaml [2] and can be written with little or no 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.

Our experience is that implementing domains through this interface is difficult only when the domain itself it difficult to think about. One of the authors implemented a collection of transfer functions for the parity domain in under an hour, at which point the new domain could be used to analyze programs such as those in the SPEC2000 suite.

INSTITUTIONAL REPOSITORY
THE UNIVERSITY OF UTAH
University of Utah Institutional Repository
Author Manuscript

UU IR Author Manuscript

UU IR Author Manuscript

```
let mult d1 d2 tp =
  match d1, d2 with
    Constant(z), _
  | _, Constant(z)
      when (isZero z) -> Constant (zero)
  | Constant(z), other
  | other, Constant(z)
      when (isInteger z = Some Int64.one) -> other
  | Bottom, _
  | _ , Bottom -> Bottom
  | Constant(e1), Constant(e2) ->
      conc_to_abs (BinOp(Mult,e1,e2,tp))
```

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

The design of our domain interface, however, restricts the kinds of domains that can be plugged in: 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, we do not support relational abstract domains such as octagons [19].

5.2 Constants

The lattice for constant propagation is shown on the left side of Figure 1. 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.

5.3 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 5.

Given CIL's constant folder, a brute-force implementation of value-set is not too difficult. We constructed 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` performs a two-dimensional fold on the input sets, using the concrete operation as the folding function. The results of the concrete operations are unioned together into a set. If the result set's cardinality is larger than the maximum, $\perp$ is returned, otherwise the set is returned. The quadratic nature of this implementation implies that large value-sets will result in very slow analysis. In practice we have observed reasonable analysis speed with value-set sizes up to about 32.

5.4 Parity

The parity domain is easy to understand and not powerful enough to be useful other than as an example. The parity lattice is shown in Figure 6. One of the authors implemented the transfer functions for this domain in under an hour. Figure 10 gives the addition transfer function.

5.5 Bitwise

In the bitwise domain, shown in Figure 7, 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. Our previous experience with the bitwise domain had convinced us that hand-written transfer functions are a mistake: they are tedious and error-prone to implement unless precision is punted on. Consequently the bitwise transfer functions used by cXprop were automatically derived using a tool chain that we recently developed [23]. 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.

```
let plusa (d1:t) (d2:t) tp =
  match d1, d2 with
    Bottom, _
  | _ , Bottom -> Bottom
  | Even, Even
  | Odd, Odd -> Even
  | Even, Odd
  | Odd, Even -> Odd
```

Figure 10. Addition in the parity domain

5.6 Interval

The interval domain, shown in Figure 8, is a commonly used value propagation domain. Our interval transfer functions are automatically derived by the same tool that produces the bitwise transfer functions. They are also maximally precise, and, unlike many interval implementations, they are correct in the presence of integer overflow. We currently support 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 datatype, the fixpoint computation can take a long time. To accelerate termination, we widen intervals above a user-defined width to bottom.

6. Evaluation

We use the following 20 benchmarks to evaluate cXprop.

Miscellaneous applications:

- `rc4` — cryptography
- `dhrystone` — a standard timing benchmark
- `yacr2` — yet another channel router

SPEC 2000 [14] applications:

- `gzip` — compression
- `mcf` — combinatorial optimization
- `bzip2` — compression

MiBench applications [13]:

- `basicmath` — mathematical calculations
- `patricia` — a Patricia trie
- `FFT` — Fast Fourier Transform

TinyOS 1.x applications:

- `blink` — 1Hz timer on red LED
- `cnttoledsandrfm` — 4Hz timer on LEDs and network
- `hfs` — high frequency sampling
- `rfmtoleds` — network data to LEDs
- `surge` — multi-hop ad hoc routing

University of Utah Institutional Repository
Author Manuscript

UU IR Author Manuscript

UU IR Author Manuscript

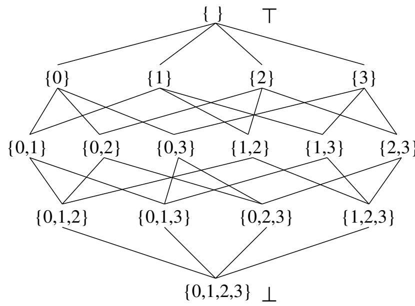

Figure 5. An example lattice for the value-set domain. The concrete domain in this example is limited to the integers from zero to three

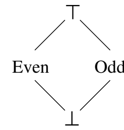

Figure 6. The parity lattice

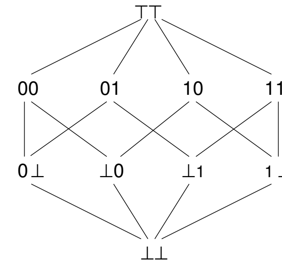

Figure 7. The two-bit bitwise lattice

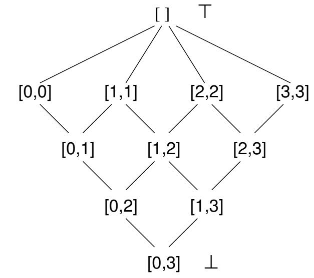

Figure 8. The two-bit unsigned interval lattice

- `taskapp` — Tiny Application Sensor Kit [1]
- `acoustic` — acoustic localization [15]
- `agilla` — mobile agent middleware [10]
- `ecc` — elliptic curve cryptography [17]

TinyOS 2.x [16] alpha-quality applications:

- `null` — empty skeleton application
- `basestation` — bridge between serial and radio

We used a maximum set size of 16 for the value-set domain and a maximum interval size of 50 for the interval domain. These limitations are arbitrary and may be increased or decreased to match performance criteria for an analysis. We also limited the constant and value-set domains to gather only statistics for integral types in order to compare with our implementation of the interval and bitwise domains. The existing transfer functions we used for the interval and bitwise domains did not handle anything other than integral types. This is not a fundamental limitation of the domains, but merely a consequence of our implementation. Implementing transfer functions in these domains is tedious and error-prone, so using an existing correct implementation was reasonable.

In the rest of this section, graphs present the benchmarks in order of increasing average analysis runtime for all five domains. We omit results for the parity domain as it never returns useful results.

6.1 A domain-independent precision metric

Pluggable domains make it tricky to measure cXprop's analysis precision in a generic way. We 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 `int` 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 `int`, 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*. We define the information known about a variable as follows:

$$\text{\# of information bits} \stackrel{\text{def}}{=} \log_2 j - \log_2 k$$

where
j = total # of representable values
k = # of possible values

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 $\perp$, 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 we extend this metric to cover entire programs. The metric that we have chosen is inspired by SSA [7]; it examines every static assignment to a variable. We define the *total information known* about a program to be the sum of the information known about each variable after an assignment, divided by the sum of the information representable by each variable that is assigned. We illustrate total information known using the following code:

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

Function `foo` contains three assignments, at lines two, four and six. Suppose that `x` is $\perp$ coming into the function. The assignment at line two also gives $\perp$, contributing zero bits of information. The assignments at lines four and six are constant, contributing 32 bits of information each. The total information known for this function is then $64/96$ information bits, or 67%.

6.2 Precision evaluation

Figure 11 compares the total information known for various benchmarks when analyzed using different abstract domains. Several interesting features stand out in the graph. First, none of the domains is uniformly better than the others. For `hfs` and `taskapp` the bitwise domain obtains the most information, while for `blink` and `acoustic` the value-set domain leads to more information. Sec-

University of Utah Institutional Repository
Author Manuscript

UU IR Author Manuscript

UU IR Author Manuscript

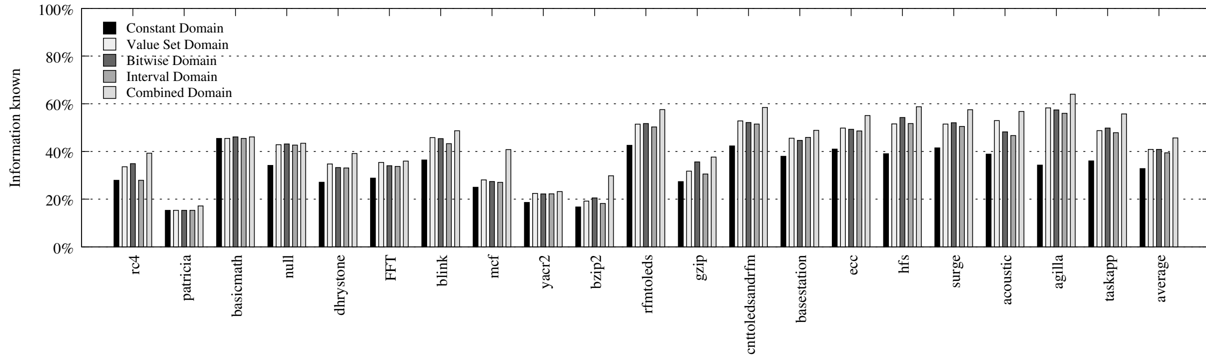

Figure 11. Different abstract domains compute different information about the test programs

ond, the domains do not perform uniformly across all programs. Although the "average" bar shows that the interval domain collects less information for the test suite as a whole, on two of the twenty benchmarks it is the best domain. It is important to have pluggable domains so that the domain most suited for a particular application may be used.

One question that we had was: When different domains perform comparably, are they "learning" the same information about a program? To answer this question we implemented a "combined domain" that is always at least as precise as any of the bitwise, interval, and value set domains. The combined domain basically intersects the concretization sets of each of the three abstract values that cXprop computes for a variable at a program point. When a bar for the combined domain in Figure 11 is higher than the other bars, it means that the three domains are learning different facts about the program.

Of course, even an ideal dataflow analyzer cannot know 100% of the information for a typical program. Since the upper bound on analysis precision is not computable, we performed a dynamic dataflow analysis by instrumenting our benchmark programs. The true upper bound of analysis precision must then be somewhere between the static and dynamic measurements. This data is presented in Figure 12. The information known dynamically is typically a combination of multiple runs of the benchmarks with different inputs.

There are several reasons for the gap between the static and the dynamic information. The biggest source of information loss is cXprop's context insensitivity. We are currently investigating various potential solutions to this problem. The next largest source of imprecision is the inherent limitations of our abstract domains and imprecision in hand-written transfer functions. Our imprecise modeling of arrays also causes some precision loss.

We do not have dynamic program information for TinyOS applications because the overhead of the dynamic dataflow instrumentation is prohibitive on the resource-constrained sensor network nodes. We do have some preliminary results—only for global variables—for TinyOS applications that we generated by extending the Avrora [26] cycle-accurate simulator. We do not present results from these, but we have used them to test that cXprop is sound when used to analyze TinyOS applications.

6.3 Analysis time

Some cXprop analysis times are presented in Figure 13. The choice of abstract domain influences analysis time in two ways. The value-set, bitwise, and interval lattices are all much taller than the constant or parity domains. It therefore takes longer for values to descend the lattice to their fixpoints. The more complicated domains also have more complicated transfer functions, which are slower.

benchmark (loc)	constant	value-set	bitwise	interval
`blink` (1852)	0.29	0.78	0.75	0.82
`rfmtoleds` (7785)	6.88	46.9	54.9	95.2
`surge` (10653)	18.1	138	172	299
`acoustic` (13804)	32.5	193	253	360
`agilla` (34414)	786	2990	2770	5570

Figure 13. Time in seconds for analysis to complete for some benchmarks

Because of these dependencies, the time it takes cXprop to run varies from almost nothing for `null` to over an hour and a half for `taskapp`. Both `taskapp` and `agilla` have significantly longer analysis times than the rest of our benchmarks. cXprop's performance is limited by our choice of a dense dataflow representation, by the fact that we model global variables (which necessitates pushing around very large machine states for some programs), by our concurrency model for TinyOS (which adds many implicit flow edges), and by our decision to model struct fields. Also, our implementation is a research prototype; its performance has not yet been tuned.

6.4 Reducing code size of TinyOS applications

Figure 14 presents the results of using cXprop's value-set domain and atomicity-aware concurrency model to eliminate dead code in some TinyOS applications. The average code size reduction for the value-set domain is 9.2%. We 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 pass is 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 for interprocedural analysis. Second, minor code transformations implemented by CIL, such as the introduction of temporary variables, handicap our efforts by increasing the size of the generated code by several percent. We have implemented a number of peephole optimizations in CIL to undo these transformations when CIL pretty-prints a C file. These partially offset CIL's code size penalty.

6.5 Finding limited-bitwidth data in TinyOS applications

Some experiments that we performed using a simulator for TinyOS nodes [26] indicated that on average, only 1–2 bits of each byte allocated by a TinyOS program are actually used. For example,

University of Utah Institutional Repository
Author Manuscript
INSTITUTIONAL REPOSITORY
THE UNIVERSITY OF UTAH
UU IR Author Manuscript
UU IR Author Manuscript

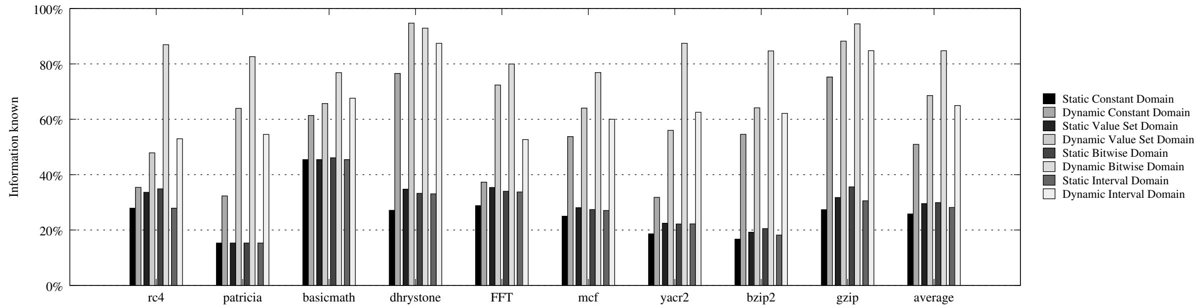

Figure 12. Comparison of static and dynamic information known

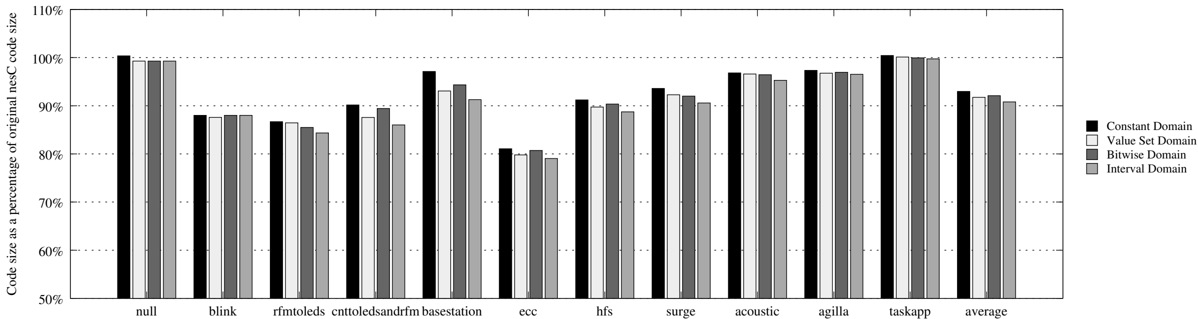

Figure 14. Code size of TinyOS applications after using cXprop to perform interprocedural dead code elimination

many bytes are used to store flag variables, state variables, or limited-range counters. We wanted to see if we could statically identify this limited-bitwidth global data. A possible use for this information would be to perform RAM compression.

Our results are shown in Figure 15. cXprop discovers that an average of 8.2% of bits allocated to global variables are unnecessary. Unfortunately, much of the poorly-used RAM in TinyOS applications is difficult or impossible to identify statically. For example, many applications allocate too many buffers for network data, put data only into the first part of allocated packet buffers, or receive limited-bitwidth data over the network. These unused bytes can only be detected if the behavior of the entire sensor network is considered—this is well beyond the scope of our work.

7. Related Work

A vast amount of research on dataflow analysis exists; we therefore discuss and cite only the most related work here. Generalized constant propagation [28] extends constant propagation to handle intervals. Our "conditional X propagation" can be considered to be a generalization of generalized constant propagation.

PAG [18] automatically generates a dataflow analyzer based on inputs written in domain-specific languages describing the domain lattice, the transfer functions, a language-describing grammar, and a fixpoint solution method. cXprop is far less configurable, providing only pluggable abstract domains. However, as far as we know, cXprop already supports a richer variety of value-propagation domains and concurrency models than does PAG. Also, while the PAG research evaluated a number of different fixpoint algorithms, we 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 [8]. This library uses well-defined interfaces to allow the user to compose transfer functions, lattices, flow graphs, and a fixpoint solver. Dwyer and Clarke present a more abstract and theoretical tool, whereas cXprop pragmatically focuses on analyzing embedded software written in C.

Our strategy for analyzing global variables in the presence of interrupts and nesC `atomic` sections is novel. Previous work on dataflow analysis for concurrent software has focused on thread-based systems [9, 24].

Although our "information bits" evaluation metric for cXprop is novel, we were inspired by the evaluation of the Bitwise compiler [25], which compares static interval domain results with dynamically gathered dataflow information. Methods also exist for measuring the precision of transfer functions [22], but this did not seem to be amenable to the cross-domain comparisons that we wanted to make.

8. Future Work

Improving analysis speed Our current version of cXprop uses a dense representation for variable assignment. Moving to SSA or FSSA should significantly reduce the time it takes for the analysis to reach a fixpoint. This is consistent with Wegman et al.'s results [29] and FSSA research [3]. Also, cXprop is currently untuned; we expect that significant speedups could be obtained with a little hard work and a profiler. Finally, our most precise concurrency model simulates the firing of all interrupts at the end of every atomic section. By using the results of an alias analysis to compute the set of interrupts that could possibly interfere with the global variables manipulated by a given atomic section, we expect to be able to avoid modeling the firing of some interrupts.

Improving precision The combined abstract domain that we described in Section 6.2 collected results from various domains

University of Utah Institutional Repository
Author Manuscript

UU IR Author Manuscript

UU IR Author Manuscript

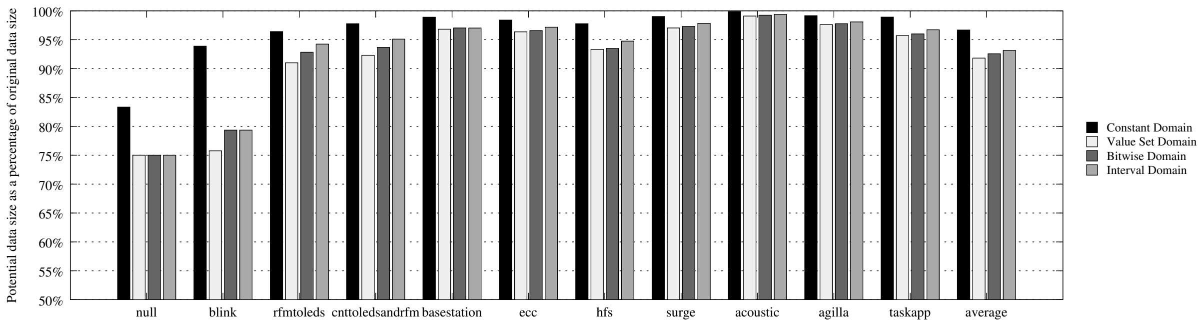

Figure 15. Percentage of bits allocated to global variables that cXprop cannot show to be unnecessary

only after analysis in each domain had reached a fixpoint. On the other hand, domain products such as the reduced product [4] and Granger's product [12]. exploit synergies between multiple domains during the course of the analysis, potentially increasing precision.

cXprop currently models arrays in a rudimentary way. Furthermore, there is room for improvement in our treatment of variables killed when a function is called. Finally, we expect that further improvements to our concurrency models will be able to increase analysis precision.

Improving expressiveness While our abstract domain interface supports a wide variety of domains, it currently does not support backwards analyses or relational domains such as octagons [19]. We plan to generalize our interface to support these.

Finding more nails to hit cXprop is a general-purpose value propagation engine that has more applications to improving embedded software than we have yet explored. For example, we believe that dataflow analysis can be used to infer causality and sequencing relationships between interrupt handlers. This information will support reducing the size of the state space that program-checking tools need to explore when verifying safety and liveness properties of interrupt-driven embedded applications.

9. Conclusions

cXprop is a tool for performing analysis and transformation of C programs that provides a standard, convenient interface for plugging in user-defined abstract value propagation domains. Pluggable domains are useful for two reasons. First, they permit a suitable domain to be used to analyze a particular program. Second, they facilitate experimentation with new abstract domains without the significant overhead of creating an analyzer for C. We have implemented and evaluated five abstract domains: parity, constants, intervals, bitwise, and value-sets.

We applied cXprop to TinyOS applications for the Mica2 platform. To improve analysis precision, we created a novel concurrency model that exploits nesC's `atomic` sections and race detection in order to efficiently analyze the values of global variables in the presence of interrupts. Our tool reduces application code size by 9.2% on average. It is also capable of finding limited-bitwidth global variables that make poor use of RAM.

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

Acknowledgments

We thank the CIL developers for their help. We also thank Eric Eide, Alastair Reid, and Ben Titzer for their helpful comments on drafts of this paper. This work is supported by National Science Foundation CAREER Award CNS-0448047.

References

[1] Phil Buonadonna, Joseph Hellerstein, Wei Hong, David Gay, and Samuel Madden. TASK: Sensor network in a box. In *Proc. of the European Workshop on Wireless Sensor Networks*, Istanbul, Turkey, 2005.

[2] Caml language Web site. `http://caml.inria.fr/`.

[3] Jong-Deok Choi, R. Cytron, and J. Ferrante. On the efficient engineering of ambitious program analysis. *IEEE Trans. Softw. Eng.*, 20(2):105–114, 1994.

[4] Michael Codish, Anne Mulkers, Maurice Bruynooghe, Maria Garcia de la Banda, and Manuel Hermenegildo. Improving abstract interpretations by combining domains. *ACM Transactions on Programming Languages and Systems*, 17(1):28–44, 1995.

[5] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th Symp. on Principles of Programming Languages (POPL)*, pages 238–252, Los Angeles, CA, January 1977.

[6] Crossbow Technology, Inc. `http://xbow.com`.

[7] Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[8] Matthew B. Dwyer and Lori A. Clarke. A flexible architecture for building data flow analyzers. In *Proc. of the 18th Intl. Conf. on Software Engineering (ICSE)*, pages 554–564, Berlin, Germany, March 1996.

[9] Matthew B. Dwyer, Lori A. Clarke, Jamieson M. Cobleigh, and Gleb Naumovich. Flow analysis for verifying properties of concurrent software systems. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*, pages 359–430, October 2004.

[10] Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. Mobile agent middleware for sensor networks: An application case study. In *Proc. of the 4th Intl. Conf. on Information Processing in Sensor Networks (IPSN 05)*, pages 382–387, Los Angeles, CA, April 2005.

[11] David Gay, Phil Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, pages 1–11, San Diego, CA, June 2003.

University of Utah Institutional Repository
Author Manuscript

[12] Philippe Granger. Improving the results of static analyses of programs by locally decreasing iterations. In *Proc. of the Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 68–79, New Delhi, India, December 1992.

[13] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of Workshop on Workload Characterization*, pages 3–14, Austin, TX, December 2001. `http://www.eecs.umich.edu/mibench`.

[14] John L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7), July 2000.

[15] Ákos Lédeczi, András Nádas, Péter Völgyesi, György Balogh, Branislav Kusy, János Sallai, Gábor Pap, Sebestyén Dóra, Károly Molnár, Miklós Maróti, and Gyula Simon. Countersniper system for urban warfare. *ACM Trans. Sen. Netw.*, 1(2):153–177, November 2005.

[16] Philip Levis, David Gay, Vlado Handziski, Jan-Hinrich Hauer, Ben Greenstein, Martin Turon, Jonathan Hui, Kevin Klues, Cory Sharp, Robert Szewczyk, Joe Polastre, Philip Buonadonna, Lama Nachman, Gilman Tolle, David Culler, and Adam Wolisz. T2: A Second Generation OS For Embedded Sensor Networks. Technical Report TKN-05-007, Telecommunication Network Group, Technische Universität Berlin, November 2005.

[17] David Malan, Matt Welsh, and Michael Smith. A Public-Key Infrastructure for Key Distribution in TinyOS Based on Elliptic Curve Cryptography. In *Proc. of the Intl. Conf. on Sensor and Ad hoc Communications and Networks (SECON)*, Santa Clara, CA, October 2004.

[18] Florian Martin. PAG—An efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[19] Antoine Miné. The Octagon abstract domain. In *Proc. of the 8th Working Conf. on Reverse Engineering (WCRE)*, Stuttgart, Germany, October 2001.

[20] Moteiv Corporation. `http://www.moteiv.com`.

[21] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. of the Intl. Conf. on Compiler Construction (CC)*, pages 213–228, Grenoble, France, April 2002.

[22] Alessandra Di Pierro and Herbert Wiklicky. Measuring the precision of abstract interpretations. In *Proc. of the Intl. Workshop on Logic Based Program Synthesis and Transformation (LOPSTR)*, pages 147–164, London, UK, July 2001. Springer-Verlag.

[23] John Regehr and Usit Duongsaa. Deriving abstract transfer functions for analyzing embedded software. In *Proc. of the 2005 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Ottawa, Canada, June 2006.

[24] Martin C. Rinard. Analysis of multithreaded programs. In *Proc. of the 8th Static Analysis Symposium*, Paris, France, July 2001.

[25] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, pages 108–120, Vancouver, Canada, June 2000.

[26] Ben L. Titzer, Daniel Lee, and Jens Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Proc. of the 4th Intl. Conf. on Information Processing in Sensor Networks (IPSN)*, Los Angeles, CA, April 2005.

[27] Bryan Turner. RandomProgramGenerator, 2005. `http://www.fractalscape.org/RandomProgramGenerator`.

[28] Clark Verbrugge, Phong Co, and Laurie Hendren. Generalized constant propagation a study in C. In *Proc. of the Intl. Conf. on Compiler Construction (CC)*, Linkoping, Sweden, April 1996.

[29] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):181–210, April 1991.

[30] Daniel S. Wilkerson. Delta, 2003. `http://delta.tigris.org/`.

[31] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.

UU IR Author Manuscript
UU IR Author Manuscript