

Dataflow analysis for interrupt-driven microcontroller software

Nathan Coopridner
School of Computing, University of Utah
coop@cs.utah.edu

October 12, 2007

1 Introduction

The proliferation of embedded devices has spurred the sales of microcontroller units (MCUs). 6.8 billion MCUs shipped in 2004 [3] and the Semiconductor Industry Association predicts MCU sales to grow by 6.1% from 2006 to 2009 [41]. This expectation for MCU sales growth exists in part because MCUs permit the addition of software control to constrained embedded devices.

Power, size, features, and price all constrain MCU-based systems. This is especially true of wireless system deployments in hard-to-retrieve locations. Wireless sensor networks (WSNs) represent a burgeoning research area of constrained MCU-based systems. A wireless sensor network consists of many nodes, called motes, in a network. The components in a mote vary, but typically include one or more sensors, a radio communications device, a MCU, and a battery power source. Development of WSN applications requires carefully considering the system's constraints:

- **Power** - Motes use some form of limited power source, most often batteries. Replacing the batteries of a mote is impractical, if not impossible, after deployment. Motes carefully minimize power usage by sleeping most of the time and only waking up to occasionally perform the necessary tasks. Decreasing the amount of time the mote needs to be awake increases the life of the mote
- **Size** - Because motes will go into extreme environments, they must be small so they will fit. Existing WSN deployment locations include on zebras [21], in wildlife preserves [26], on glaciers [28], next to volcanoes [47], and surrounding urban spaces [42]. These remote locations require a small footprint, which limits the number of peripheral devices available for a mote. Additional resources cannot simply be strapped on to the main device
- **Features** - WSNs must handle a large number of activities which change from application to application. Most networks require some sort of routing, but some may also require security, dynamic updating, data handling, or other features as well. Developers continually stretch constrained hardware to execute new features without upgrading the hardware itself.
- **Price** - Prolific and remote deployments encourage the use of cheap motes. The WSN community concedes that "disposable" motes may still be decades away for non-military applications, but that is still a driving design goal. Even in the absence of disposable motes, large WSNs require cheap motes in order to be practical.

These constraints are tightly interrelated. For example, the need for small motes prohibits using larger batteries, and the need for cheap motes prevents using a state-of-the-art MCU with more features. These constraints also push WSNs to use tiny MCUs. WSN developers must always remember these constraints and react accordingly.

Current practice for dealing with these constraints is to design efficient systems, program conservatively, and then manually tweak to make it fit. This process leaves much to be desired. Designing efficient systems is difficult. The hardware on which the system will run often changes, warranting a change to the system. Enforcing conservative programming is also difficult, especially since it is not always well defined. Anyone

who has had the experience of manually tweaking programs to meet these constraints will concede it is both difficult and error-prone. It is often the case that manual tweaking also makes code over-specialized and difficult to maintain.

The effectiveness of this process may be greatly improved by using an optimizer powered by dataflow analysis. Such an optimizer reduces the need of system designers and programmers to be clever because the optimizer will be clever for them. In fact, an aggressive dataflow analysis actually discourages cleverness because such behavior often confuses the analysis! An automatic optimizer improves over manual optimization because a more generic version of the code (pre-optimization) may be stored and maintained.

Problem A straightforward instantiation of traditional dataflow analysis techniques fails to provide adequate precision for aggressive optimization of interrupt-driven MCU systems. These systems contain features which either must be dealt with because they hinder the analysis, or should be leveraged because they represent an untapped potential for information. The pivotal features in interrupt-driven MCU systems are the interrupts and the microcontroller themselves.

An interrupt complicates the dataflow analysis for a given code segment because during the segment it may fire at any time, it may fire repeatedly, or it may never fire at all. To avoid unnecessary degradation of the results, an analysis must have a better coping strategy for multiple flows due to interrupts than just modeling all possible interleavings of the flows. On the other hand, interrupts also provide useful divisions between program elements. Some data and code may be only accessed inside a single interrupt, inside multiple interrupts, or outside of any interrupt. Not capitalizing on the isolation of accesses due to interrupts will unnecessarily reduce analysis precision.

Low-level systems programming on MCUs often involves inline assembly and directly accessing specific parts of the MCU. For example, the status register may be directly read, shifted and masked in order to determine the status of the interrupt bit. Naively and pessimistically analyzing all hardware accesses degrades the analysis of systems code very quickly. Leveraging the information is crucial for the success of the analysis.

Solution I am developing a framework to enable sound and accurate dataflow analysis for interrupt-driven microcontroller programs. This framework adapts existing abstract interpretation ideas to system-level C code. My framework integrates several synergistic analyses such as value-flow analysis, pointer analysis, and callgraph construction. Contributions of my work include using pluggable abstract interpretation domains, providing a novel model of interrupt-driven concurrency, allowing dataflow through volatile variables when safe to do so, and tracking interrupt firing dependencies. When compared to a highly optimizing C compiler, my framework improves traditional code optimizations such as conditional constant propagation, dead code elimination, redundant synchronization elimination, and inessential-safety-check removal. It also enables new transformations such as RAM compression, the sub-word packing of statically allocated global variables. These transformations help microcontroller programs meet stringent resource requirements.

In Section 2 I present background material upon which I build my framework. Section 3 outlines my research. Finally, I provide a plan for my work in Section 4.

2 Background

In this section I review established principles, techniques, and tools upon which I build my research.

2.1 Abstract interpretation

Abstract interpretation [11] is a static analysis technique used to determine the behavior of a program. During a dynamic execution of a program, the program will have a specific state at every moment. These states are referred to as *concrete states*. For non-trivial programs the number of these concrete states is practically infinite. This makes determining complete program behavior through exploring concrete states impossible. Abstract interpretation replaces the concrete states with *abstract states* in order to make the space manageable.

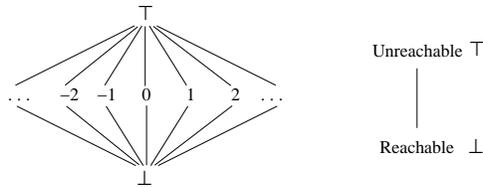


Figure 1: Conditional constant propagation is an analysis combining the constant propagation lattice (left) with the unreachable code elimination lattice (right)

An abstract state represents a set of concrete states and exists as a member of an *abstract domain*. An abstract domain is a partially ordered set of abstract states. The abstract domain forms a *lattice* with “unknown,” the set of all possible concrete values, being at the bottom (\perp) and “undefined,” the empty set, being at the top (\top). Figure 1 shows lattices for the constant propagation domain and unreachable code elimination domain.

An abstract interpretation may be used inside a *dataflow analysis*. Dataflow analysis works by symbolically executing the program using abstract states in place of concrete states. The semantics of the program are abstracted into *transfer functions* which operate on the abstract states. Transfer functions must soundly overapproximate all possible concrete operations which could occur given abstract state inputs. When control-flow revisits a program point, the incoming abstract states are *merged* together. Merging computes the least upper bound (LUB) of both states in their abstract domain lattice. The LUB will contain all the concrete states from both abstract states.

Aggressive and sound optimizers use fixed-point dataflow analysis. In this approach the analysis initializes the abstract state at all program points optimistically (and unsoundly) except for the entry points to the program. The analysis assigns pessimistic but precise as possible abstract states to the entry points. The dataflow computation then iterates until reaching a fixed-point, allowing the abstract state to flow from the entry points. This finds the greatest lower bound in the abstract domain lattice for each program point, or the most precise abstraction for the domain.

2.2 Microcontroller Software

Software for MCUs is somewhat different from general-purpose application code. These characteristics are largely programming-language independent, having more to do with requirements of the domain and properties of the platform. Below are some key features of microcontroller software; one represents a complication that I am forced to handle and three are restrictions that I can exploit.

- **Software is interrupt-driven.** Interrupts are the only form of concurrency on many MCU-based systems, where they serve as an efficient alternative to threads. Interrupt-driven software uses diverse synchronization idioms, some based on disabling interrupts and others based on volatile variables and developer knowledge of hardware-atomic memory operations. The implication for my work is that I need to provide a sound dataflow analysis of global variables even in the presence of unstructured, user-defined synchronization primitives.
- **Software reuses code.** Application components are pieced together from libraries. While these libraries are highly tuned, some undesired code still will get imported. This means there is a potential for optimization by getting rid of unused imported code and data from libraries. Another consequence of software reuse is that it can prevent complex optimizations. Manual tweaking of the code may make code reuse impractical or even impossible.
- **Memory allocation is static.** The only form of dynamic memory allocation on most MCU-based systems is the call stack. When RAM is very constrained, heaps are unacceptably unpredictable (e.g., fragmentation is difficult to reason about and allocations may fail). The implication is that my pointer analysis can be in terms of static memory objects and stack objects.

- **Memory mapped I/O.** Areas of the MCU addressable space are set aside for I/O instead of memory. Those assigned addresses then map to hardware registers. Memory mapped I/O avoids complexity introduced by port mapped I/O. This makes the MCUs cheaper and faster. The MCU ISA does not need to add additional instructions for dealing with I/O, and all memory addressing modes are available for I/O.

Analyzing microcontroller software is an area of active research. Several industrial applications of abstract interpretation exist, including PAG [27] and ASTRÉE [4]. PAG uses domain-specific languages to automatically generate a dataflow analysis, while ASTRÉE does not handle recursion but has been used to analyze Airbus control code. In addition to general-purpose tools, abstract interpretation has been used to determine the maximum run-time stack depth statically [5, 17, 37]. Several languages targeting microcontroller programs use static analysis for optimization. nesC [15] performs a limited dead code elimination pass. Virgil [45] separates *initialization time* from *run time* in order to improve its static analyses. TinyGALS [6] enforces type safety of component connections through the use of a static analysis. Previous work on dataflow analysis for concurrent software has focused on thread-based systems [14, 40]. My work on dataflow analysis for *interrupt-driven* concurrent systems will be the first of its kind that I am aware of.

2.3 Platforms

My framework targets C programs for the mica2 and telosb wireless sensor network platforms. I use CIL to parse and organize C programs written in TinyOS.

CIL [29] provides several useful features for static analysis. First, CIL tools reduce generic C into CIL’s intermediate language. This intermediate language simplifies the constructs my analysis needs to handle. Second, CIL provides a source code merger. This tool combines the code for a program from several files into a single file, making whole-program analysis feasible. Third, CIL handles all of ANSI C and GNU C. Fourth, CIL is mature and well supported. While it is an academic tool, it has been rigorously used and tested. CIL’s maintainers welcome and quickly react to bug reports. Lastly, CIL provides several frameworks and extensions for doing static analysis.

TinyOS [19] is a component-based operating system for wireless sensor network nodes. It uses a static resource allocation model and is written in a C dialect, called nesC [15], that is translated into C. TinyOS uses a specific interrupt-driven concurrency model. Most code runs in tasks which do not preempt one another. Interrupts may preempt tasks and interrupts. Pieces of code with interrupts disabled form non-preemptable atomic sections. Typically, TinyOS applications spend most of their time in some kind of *sleep* mode in order to conserve energy.

The mica2 and telosb motes are popular TinyOS platforms. The mica2 mote uses an Atmel AT-Mega128L MCU, which runs at a top speed of 16 MHz. The name for this chip comes from its flash memory size: 128 K. The flash memory is typically used as program memory and for applications it can be considered as ROM. The AT-Mega128L has 4K of SRAM for data memory. On the other hand, the telosb mote uses a Texas Instruments MSP430 MCU with 10 K of RAM. It has 48 K of flash for program memory and runs at a top speed of 16 MHz. Both of these chips have severe resource constraints.

3 Research plan

My research will present several solutions to challenges impeding the adoption of dataflow analysis techniques to MCU programs. I will present these solutions in terms of an analysis framework. My framework will build on established abstract interpretation techniques but will also exhibit four novel contributions. These contributions are pluggable abstract domains, dataflow precision increase for interrupt-driven code, triggering information, and RAM compression.

3.1 Pluggable abstract domains

Motivation An abstract interpretation moves program computation from a concrete domain into an arbitrary abstract domain. Choosing an appropriate abstract domain is an important part of designing an effective abstract interpretation. In general, analyzing in terms of faster abstract domains provides less

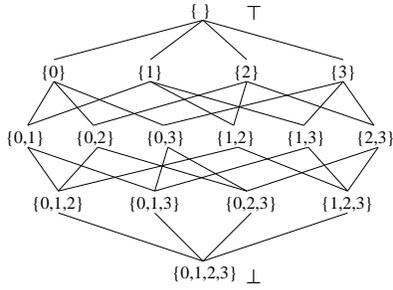


Figure 2: An example lattice for the value-set domain. The concrete domain in this example is limited to the integers from zero to three

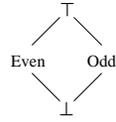


Figure 3: The parity lattice

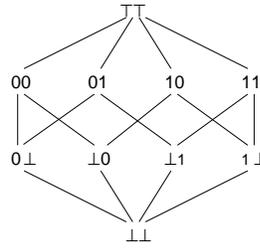


Figure 4: The two-bit bitwise lattice

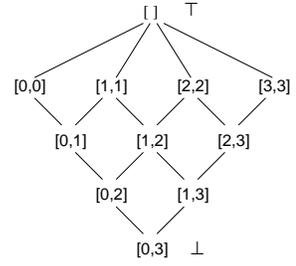


Figure 5: The two-bit unsigned interval lattice

precision while analyzing in terms of more precise domains results in slower analyses. The ideal domain for an analysis runs as quickly as possible while still obtaining just enough information as required for the verification or optimization.

One problem with comparing abstract domains in search of this ideal is that changing a domain often means changing the analysis. This is tedious and it also changes cross-domain comparisons into cross-analysis comparisons. However, most of the pieces of an abstract interpretation are independent of the domain. I clarify and formulate the interface between the domain and the rest of my analysis.

The choice of an abstract domain is also complicated by the difficulty of judging precision. The effectiveness of using a particular abstract domain in the past does not necessarily correlate to the effectiveness of using it in the future. How the analysis will be used and the make up of the program both factor into future performance expectations for an abstract domain. I formulate an independent metric for judging the precision of an abstract domain.

Proposed solution I define a *pluggable abstract domain interface* for my value-flow analysis. Providing an implementation for this interface produces a pluggable abstract domain. Pluggable abstract domains permit the same analysis to execute with different abstract domains. This keeps the rest of the analysis constant and enables cross-domain comparison.

The majority of the pluggable abstract domain interface specifies transfer functions. Transfer functions abstract program behavior into the appropriate abstract domain. Transfer functions must be defined for most of C’s operators. Necessary definitions include the mathematical, logical, and bitwise operators, plus casts. Transfer functions take one or more abstract domain values as inputs along with the type of the operation (i.e. `char`, `short`, `int`). They return logical values for logical operators and abstract values for other operators.

Besides transfer functions, the rest of the pluggable abstract domain interface specifies helper functions. These include concretization (move from the abstract domain to the concrete domain), abstraction (move from the concrete domain to the abstract domain), join (only keep values covered by both abstract values), meet (keep all abstract values covered by either abstract value), widen (if abstract value represents too many concrete values then move it to \perp), and compare (are the abstract values the same). These helper functions deal with control flow, debugging, and analysis issues.

The abstract domain interface consists of 42 functions. Thirteen are abstract interpretation helper functions and 29 are transfer functions. Further development may adapt the interface in order to meet unforeseen needs. For example, already included in those totals are backwards operations added after the initial interface definition. The Ocaml module interface specifying the current abstract domain interface can be found in Appendix A.

I use the pluggable abstract domain interface to implement five abstract domains: parity, constant, value-set, interval, and bitwise. The lattices for these domains are shown in figures 1 and 2-5. In the parity domain a value is either \top , \perp , **even** or **odd**. The constant domain is well understood. In the value-set domain a value is either \perp or a set of values. The interval domain uses intervals of the form $[x, y]$ for values. In the

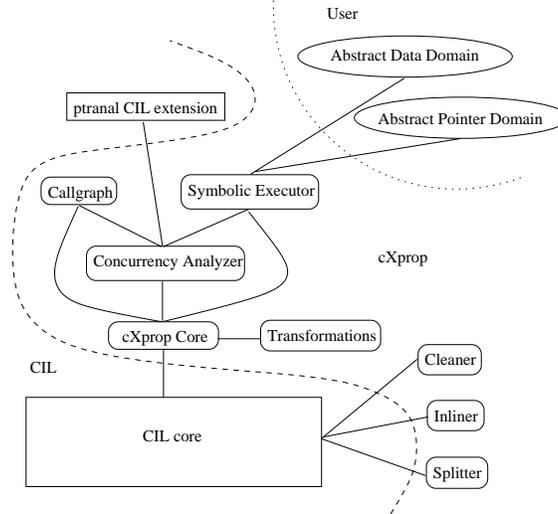


Figure 6: cXprop, its internal structure, and its relationship with CIL and abstract domains

bitwise domain values are vectors of three valued bits: 0, 1, \perp . For the bitwise and interval domains I use many automatically generated transfer functions created by Regehr and Duongsaa [35].

I define a metric for inter-domain comparison of precision called *information known*. My framework defines precision in terms of candidate elimination; the fewer values an abstract value represents, the more precise the analysis. For example, if x holds the value-set 0, 42, 65 using the value-set domain and $[0, 0]$ using the interval domain, then the interval domain is more precise because the value-set abstract value represents two additional values beyond 0. Information known quantifies precision in terms of the domain representation and the bitwidth of the variable. This metric can be collected at each program point or for the program as a whole.

Research contributions

- Isolating the abstract domain interface for value-flow analysis
- Describing precision in terms of the **information known metric**

Evaluation The pluggable abstract domain interface and the information known metric will be judged by how much they help with choosing an abstract domain to use. The first step is to produce several abstract domains. Qualitative feedback will be collected with regard to producing abstract domains using the interface. Once the domains are created, I will collect information known statistics for each of the domains on a collection of benchmarks. These benchmarks will come from SPEC 2000 [18], MiBench [16], TinyOS [19], and TinyOS 2 [24]. I will evaluate the information known metric itself by also comparing the CP and DCE results of each domain and seeing how the transformation results and the information known metric relate.

Relation to previous work PAG [27] automatically generates a dataflow analyzer based on the user defining the domain lattice, the transfer functions, a language-describing grammar, and a fixpoint solution method. My framework keeps the language-describing grammar (CIL) and the fixpoint solution method both constant while allowing flexibility in the domain lattice and transfer functions through the abstract domain interface. The unique design point provided by my framework allows for better exploration and creation of value-flow domains and greater specialization of of the analysis for C.

Dwyer et al. provide composable dataflow analysis pieces in a library [13]. This library provides interfaces for composing transfer functions, lattices, flow graphs, and a fixpoint solver. This is a more abstract and theoretical tool, whereas my framework pragmatically focuses on analyzing microcontroller software written in C.

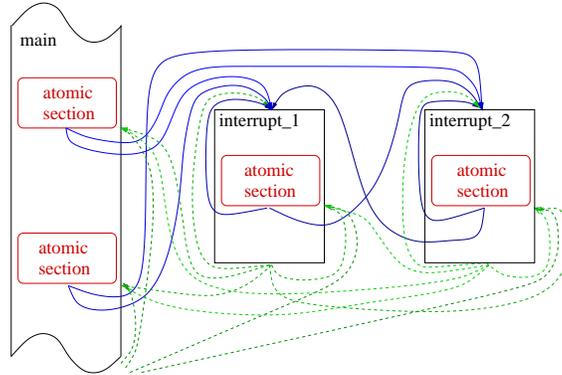


Figure 7: To analyze interrupt-driven software, conditional X propagation adds implicit flow edges to each interrupt handler, and back, at the end of every `atomic section`

The Bitwise compiler [43] inspired the development of my *information bits* evaluation metric. That work compares static analysis results with dynamically gathered information. Work has been done for measuring the precision of transfer functions [31], but this did not seem amenable to cross-domain comparisons.

Current status Usit Duongsaa’s masters thesis [12] compares his interval and bitwise transfer functions using my pluggable abstract domain interface and information known precision metric. The results were also presented in a conference paper [35].

My research prototype for my framework is called *cXprop*. The name stands for conditional X propagation, where X is the pluggable abstract domain. I published my initial work with *cXprop* in a couple of different venues [8, 9]. A journal article capturing novel contributions of *cXprop* and current performance numbers is forthcoming.

3.2 Dataflow with interrupts

Motivation Most real MCU code uses some form of concurrency and most concurrency on MCUs is interrupt-driven. Dataflow in the presence of interrupt-driven concurrency becomes a first-order concern for MCU program analysis. Fortunately, interrupt-driven concurrency typically uses simple locking mechanisms, short interrupt handlers, and specific preemption behavior. I present a novel model of interrupt-driven concurrency in order to increase precision for MCU code for global variables. Improving the precision of global variables will also improve the precision of local variables which depend on them.

My framework will leverage features of the concurrency model used in TinyOS. Many embedded applications use this model, but TinyOS provides a clean implementation to examine. TinyOS calls code which runs exclusively in tasks *synchronous* while code which may run in an interrupt is called *asynchronous*. Synchronous code is atomic with respect to other synchronous code, while asynchronous code is only atomic inside of atomic sections. Variables which are only accessed in synchronous code are *unshared*, meaning only one flow of execution will access them at a time. Variables which may be accessed in asynchronous code are *shared*, meaning more than one concurrent flow may access the variable at the same time. My novel concurrency model will allow for the dataflow tracking of shared variables.

Proposed solution My proposed novel concurrency model leverages properties of atomic sections and interrupts in conjunction with a race detector. Atomic sections, implemented by turning off interrupts, will form a basic unit in my model. When the program enters an atomic section it will then exhibit sequential behavior until it exits. Upon exit, my framework will simulate firing all the interrupts. This creates extra flow edges which connect all the atomic sections. The framework will also simulate firing all interrupts at the end of each interrupt in order to capture all possible interrupt firing orders. The model, with the atomic section basic units and extra flow edges, is shown in Figure 7.

My framework’s method for dealing with concurrency provides an adequate number of additional flow edges to soundly approximate behavior of the system. The dataflow analysis simulates all possible interleavings of atomic sections. This is the most precise an analysis can do without additional information about interrupt firing dependencies (see Section 3.3). Atomic sections may not be broken up by definition, which means that global variables always protected by atomic sections will be soundly approximated under this model. My framework will use a race detector in conjunction with the concurrency model to track dataflow through global variables not always protected by atomic sections.

The race detector identifies racing variables which cannot be tracked under this model. A race is when a variable may be written by one flow while simultaneously being accessed by another. This erratic flow behavior forces my framework to conservatively assume racing global variables are \perp outside of atomic sections. Global variables which are not protected by atomic sections and also not racing may still be tracked in my concurrency model. These global variables will either only be accessed in one flow (essentially sequential code), or never written in a concurrent situation (only written during atomic sections in `main` or not at all). My concurrency model accurately captures both of these behaviors.

Research contributions

- **Novel concurrency model**
- Analyzing **global variables** in the presence of concurrency

Evaluation I will evaluate how my framework handles interrupts in terms of precision and speed while not sacrificing soundness. I will execute my framework prototype on benchmarks with these features active and inactive. When the features are inactive, the framework will use the next-best solution: failing to model global variables.

Relation to previous work Previous work on dataflow analysis for concurrent software has focused on thread-based systems [14, 40]. My framework is novel because it will address and leverage features of interrupt-driven systems.

Current status I implemented the novel concurrency model early in the development of my prototype [8, 9]. It greatly increased analysis precision. I added race detection to cXprop as part of the work for SafeTinyOS [34, 33, 7].

3.3 Sequencing information

Motivation Implicit dependencies make it difficult for an analysis to follow the control-flow. Determining the control-flow successors of a given statement often requires only local knowledge to obtain adequate precision. Local information is insufficient in other cases, such as with interrupts and task dispatches. These cases require implicit non-local information because they depend on global structures, underlying hardware, or environmental events. Without using this implicit information, the control-flow decision appears random and the analysis sacrifices precision to conservatively overestimate the number of possible successors.

I propose that capturing these implicit dependencies in a static analysis will significantly add precision, leading to better optimization and verification. These dependencies are common when analyzing interrupt driven systems code for WSNs.

Proposed solution I will introduce trigger recognition into my framework and use triggering information for tasks and interrupts to create a sequencing graph. The created sequencing graph will be supplemental to the control-flow graph (CFG), increasing the overall precision of the analysis by removing false control-flow edges. The improvement will be measured by comparing optimizations on TinyOS 2 code using the sequencing graph with those using traditional CFG generation.

A trigger represents an implicit control-flow dependency. The triggers of interest to my analysis are pieces of code which exclusively cause a task or interrupt to run. Posting a task means the task is scheduled to run and the task will not run unless it is posted. Turning on the timer eventually leads to timer interrupts

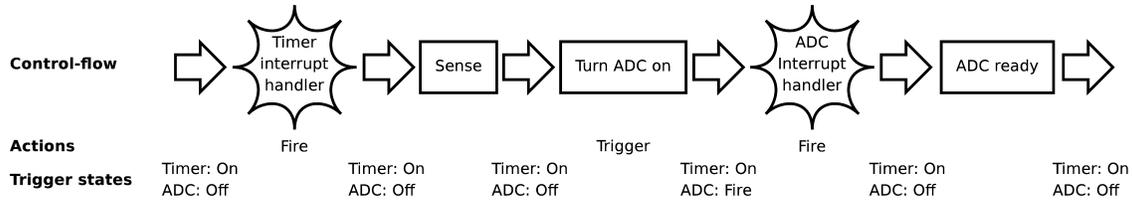


Figure 8: Example of triggering. Timer interrupt remains “on” after firing, while ADC interrupt only fires once per trigger.

and timer interrupts to not occur until the timer is activated. Activating the analogue to digital converter (ADC) eventually results in a data ready interrupts, but the interrupt will not be sent by an inactive ADC. Many triggers such as these exist, but are not obvious to an analysis from the code.

My analysis will make triggers explicit using system specific information or programmer annotations. The scheduling of a task follows a certain language idiom in TinyOS: a call to the scheduler with the task identifier as a parameter. My analysis will identify task triggers by searching for this idiom in the code. Some interrupt triggers will be identified by additional modeling of the hardware. Certain bits of memory-mapped hardware registers turn on certain features, identifying a trigger. Simple annotations will identify additional triggers.

Scheduler information and interrupt priorities may be used to remove infeasible triggers. Since a particular task may appear only once in the scheduling queue for TinyOS, when that task dispatches it does not remain triggered. Similarly, if a low priority task or interrupt executes, none of the higher-priority tasks or interrupt are triggered.

Figure 8 illustrates two interrupts and how my analysis will handle their triggers. The hardware timer has been turned on prior to the start of the example, so the timer interrupt trigger is *on*. This means that it can fire more than once per trigger, so even after the interrupt the trigger remains on. The ADC interrupt trigger indicates the interrupt will *fire*, meaning after one execution it returns to an untriggered state, or *off*.

One behavior of triggers shown in Figure 8 is that they form a sequence of execution. The timer interrupt is triggered, which then triggers the ADC interrupt. All of the possible trigger sequences for a program form a sequencing graph. `main`, tasks, and interrupts form the nodes and triggering relationships form the edges.

Information from the sequencing graph will augment the computed CFG. For example, after an atomic section an analysis must assume that all available interrupts can fire. The sequencing graph will eliminate interrupt firings which have not been triggered and are therefore unavailable. The sequencing graph will introduce non-local information about implicit control-flow dependencies. This information will then reduce the set of potential control-flow successors, which will increase the precision of the analysis.

Research contributions

- Identify **trigger** relationships in MCU code. A trigger indicates an implicit control-flow dependency.
- Leverage scheduler information and interrupt priorities to eliminate infeasible active triggers
- Use triggering information to determine the **sequencing** of a program. Sequencing refers to chains of triggers.
- Combine sequencing graph with the CFG to improve analysis precision

Evaluation I will evaluate the effect of sequencing by comparing the performance of my prototype with and without sequencing information. I will use benchmarks from the TinyOS 2 [24] release applications. My prototype should eliminate more dead code and propagate more constants when sequencing information is used. I will also compare my prototype with sequencing information to `gcc`.

Relation to previous work Several projects have used ideas similar to triggers and sequencing, but in different ways than I propose. Ramaprasad and Mueller [32] examine preemption points to increase the

Legend

analyze?	analyze: the framework will analyze this data type to some degree
	*: the framework assumes this data type does not occur
local/global	local: local variables
	global: everything else
(un)shared	unshared: data only accessed in synchronous code
	shared: everything else
racing?	racing: the data satisfies the criteria for a race condition
volatile?	volatile: the data is marked with volatile in the source
RAM/device memory	RAM: global or local variables
	device memory: memory-mapped control registers

analyze?	local/global	(un)shared	racing?	volatile?	RAM/device memory
analyze	local	unshared	-	-	RAM
analyze	local	unshared	-	volatile	RAM
*	local	shared	-	-	RAM
*	local	shared	-	volatile	RAM
analyze	global	unshared	-	-	RAM
*	global	unshared	-	-	device memory
analyze	global	unshared	-	volatile	RAM
analyze	global	unshared	-	volatile	device memory
analyze	global	shared	-	-	RAM
*	global	shared	-	-	device memory
analyze	global	shared	-	volatile	RAM
analyze	global	shared	-	volatile	device memory
-	global	shared	racing	-	RAM
-	global	shared	racing	-	device memory
-	global	shared	racing	volatile	RAM
-	global	shared	racing	volatile	device memory

Figure 9: Data classifications. Logically impossible rows are not shown, such as unshared racing and local device memory

precision of WCET analysis. The Ptolemy project [22] studies the analysis of concurrent embedded systems, among other things as well. The tagged signal model [23], causality [50], and ordering [49] are all related to triggers and sequencing. Li [25] presents a method for dynamic monitoring the sequencing relationships between components. Some of the ideas from Task/Scheduler Logic (TSL) [39, 38, 36] are complimentary to sequencing. Applying ideas from that work may improve the accuracy of the sequencing graph.

Current status I have already generated sequencing graphs for some TinyOS programs and have begun working on the implementation for cXprop.

3.4 Volatile variables

Motivation The `volatile` type qualifier in C indicates that a data value may change in “ways not under control of the implementation” [20]. In practice, developers use the `volatile` keyword in order to prevent compiler optimizations which may break the program. A racing variable or memory-mapped register access may be marked as volatile in order to prevent the compiler from caching accesses to the data. Volatile data is left out of compiler optimizations by definition, so traditional dataflow analyses conservatively assume it has \perp for an abstract value.

Volatile data forms central pieces of MCU programs. Developers mark access to hardware registers with `volatile`. The hardware registers affect dataflow and even control-flow (see Section 3.3) in the program. Shared global variables may be marked as volatile as a precaution in case they race. Not modeling volatile data results in imprecision which permeates the analysis. My framework will leverage system-specific information in conjunction with my analysis to reduce this imprecision by allowing volatile variables to be modeled.

My framework will divide data into classifications as shown in Figure 9. Each classification row has certain properties which my framework will leverage in order to increase precision. The rows which do

Device	ROM	RAM	Ratio	Price
ATtiny24	2 KB	128 B	16:1	\$0.70
ATtiny45	4 KB	256 B	16:1	\$0.97
ATmega48	4 KB	512 B	8:1	\$1.50
ATmega8	8 KB	1024 B	8:1	\$2.06
ATmega32	32 KB	2048 B	8:1	\$4.75
ATmega128	128 KB	4096 B	32:1	\$8.75
ATmega256	256 KB	8192 B	32:1	\$10.66

Figure 10: Characteristics and prices of some members of Atmel’s AVR family, a popular line of 8-bit MCUs. Prices are from <http://digikey.com> and other sources, for quantities of 100 or more.

not represent volatile data are either analyzed with traditional dataflow analysis, analyzed using ideas I previously proposed, assumed to not occur, or not analyzed (always \perp). For example, my novel concurrency model enables the analyzing of global, shared RAM. Volatile data covers the remaining classifications which my framework will analyze but which require additional work.

The behavior of volatile data is opaque at the level of the C source code, but introducing lower-level details about the MCU allows for the analysis of these classifications. Although some hardware registers must be viewed as truly volatile, some registers or even some bits of registers behave in analyzable ways. For example, the interrupt bit of the status register can be modeled by looking for assembly instructions which change it and direct changes to it through the address to which it is memory mapped. The interrupt bit will not change except in those ways. My analysis will analyze volatile data by capturing this type of system specific information. Internal abstract variables will keep track of analyzed register information.

My analysis’ race detector also helps when analyzing volatile data. Developers sometimes use the `volatile` keyword to “protect” racing variables, and sometimes they do this to variables which do not actually race. For these false-positives, the `volatile` keyword will be ignored. The race detector will also be updated to seek for racing volatile hardware registers in order to prevent incorrect modeling.

Research contributions

- Using MCU specific information and race detection to allow **dataflow through volatiles**

Evaluation Modeling volatile data will increase the precision of my analysis, and I will verify this by comparing the analysis with and without analyzing volatile data. Extra care will be taken to test the applications after analysis and transformation with volatile information.

Relation to previous work Sound dataflow through volatile variables is novel. *Nobody has been crazy enough to intentionally try it before.*

Current status I tracked the volatile interrupt bit of the status register in order to produce my concurrency model [9]. I have leveraged the race detector to enable dataflow through volatiles in order to enable better RAM compression [10]. Specifically, the task queue in TinyOS could be compressed once dataflow occurred through non-racing and non-hardware volatiles.

3.5 RAM compression

Motivation RAM usage is a first-order concern for developers of MCU software. Figure 10 shows the comparison of RAM to ROM for some small MCUs. RAM is expensive in terms of price and is also expensive in terms of energy use. These expenses normally keep available RAM at a minimal amount. I introduce a novel RAM compression technique which will allow for cheaper ROM to replace the more expensive RAM.

Proposed solution My framework enables a novel RAM compression transformation. My RAM compression technique reduces the bits used by a global variable based on how many different values the variable must represent. The dataflow analysis in my framework computes a conservative estimate of the set of representable values for each global variable. When the number of bits required to distinguish between elements of the set is less than the number of elements allocated for the variable, savings will result from RAM compression. For example, if a two-byte pointer can only ever point to four different objects then it actually only needs two bits in order to distinguish between the four objects. This example results in a 14 bit savings of RAM. The possible pointer values are stored in ROM and the two bits are used to distinguish between the pointer values. Compression and decompression functions convert the two bit values into their address counterparts.

This RAM compression transformation represents a tradeoff of ROM or CPU cycles for more RAM. The tables of values take up space in ROM and the compression and decompression functions take up CPU cycles. These tradeoffs may be leveraged to decide when to apply RAM compression because RAM compression occurs on a per global variable basis. Variables ready for compression may be ordered by increasing negative effect per the amount of RAM saved. Gradually compressing variables along this ordering is like turning a *RAM compression knob*. As the knob turns, my framework maximizes the RAM saved while minimizing the penalty. I will explore orderings where the penalty of interest extra CPU cycles or spent ROM, but some combination of these and/or other orderings are possible.

Research contributions

- Offline **RAM compression** transformation
- RAM compression **knob** for adjustment

Evaluation I will test RAM compression by comparing RAM compressed benchmarks to benchmarks compiled with standard `gcc`. My benchmark suite will be a collection of applications for Atmel AVR 8-bit microcontrollers. Benchmarks will come from KESO [44] compiled robot control programs, the Paparazzi project [30], and TinyOS [19]. I will measure ROM and RAM usage in addition to measuring the duty cycle of the TinyOS applications using Avrora [46]. The benchmarks should exhibit a significant decrease in RAM usage when being compiled using RAM compression. Potential increases in CPU cycles or ROM should be minimized using the RAM compression knob.

Relation to previous work Ananian and Rinard [1] use a static bitwidth analysis for the field packing of Java objects. Virgil [45] uses a number of offline RAM optimizations including reachable member analysis, reference compression, and moving constant data into ROM. RAM compression differs from these works in a couple of ways. RAM compression targets whole MCU applications and not just benchmarks. It also uses a very tight bit compression mechanism. Finally, RAM compression uses tradeoffs and optimizations to deal with the realities of code running on MCUs.

CRAMES [48] and MEMMU [2] use standard compression techniques to achieve online compression for embedded systems. My RAM compression technique is completely complimentary to these tools, since I use an offline approach.

Current status I added the RAM compression transformation onto `cXprop` and called the resulting tool `CComp` [10].

4 Plan of work

<i>Semester</i>	<i>Milestone</i>
Fall 2004	Read papers, tutorials, summaries and lectures on static analysis, abstract interpretation and dataflow. Obtained working versions of TinyOS for future evaluations.
Spring 2005	Learned OCaml by taking CS 7963. Investigated and learned about CIL. In particular, evaluated CIL's dataflow analysis capabilities and separated the good from the bad.

Summer 2005	Leveraged CIL to produce results for Usit Duongsaa's thesis on automatically generated transfer functions. Introduced pluggable abstract domains. Implemented the ideas described in Section 3.1.
Fall 2005	Continued to develop and enhance cXprop. Implemented concurrency models. Performed some of the work described in Section 3.2.
Spring 2006	Improved cXprop as a whole program optimizer. Implemented backwards transfer functions and introduced a stronger alias analysis. Wrote a peephole optimization cleaner and got an inliner.
Summer 2006	Continued to improve cXprop as a whole-program optimizer. Added a race detector. Began to investigate triggering and ordering. Found useful applications for cXprop as a whole-program optimizer. Investigated ideas described in Section 3.3 and worked on ideas from Section 3.2.
Fall 2006	Implemented RAM compression transformation as outlined in Section 3.5. Made necessary improvements and additions to cXprop, including modeling volatile variables, adding a struct splitter, implementing collapsed arrays. These improvements are described in 3.4, 3.2, and 3.5.
Spring 2007	Additional improvements to cXprop as a whole-program optimizer. Integrated callgraph creation as part of the analysis.
Summer 2007	Write dissertation proposal. Take written qualifier.
Fall 2007	Dissertation proposal. Start writing dissertation. Do initial work for triggers and sequencing described in Section 3.3. Expand the modeling of volatile variables described in Section 3.4
Spring 2008	Implement trigger recognition. Create sequencing and ordering analyses using information from triggers. Evaluate effectiveness of the technique on benchmarks. This work is described in Section 3.3. Continue working on dissertation.
Summer 2008	Finish generating dissertation. Dissertation defense. Edit dissertation. Submit completed dissertation.

5 Summary

I have begun work on **cXprop**, a prototype implementation of the dataflow framework described in this proposal. cXprop generalizes and extends conditional constant propagation into *conditional X propagation*. It targets low-level interrupt-driven MCU code. In addition to incorporating traditional dataflow analysis techniques, my framework has introduced and will continue to introduce several novel ideas. My framework already provides pluggable abstract domains, a novel model for interrupt driven concurrency, ways for dealing with context insensitivity, aggressive dataflow through volatile variables, and a new RAM compression transformation. Support for sequencing is currently under development.

On benchmarks **cXprop** has been shown at this point to reduce code size by 20% and data size by 10%. My new RAM compression transformation may leverage dataflow information for an addition 12% in data size savings, making total RAM savings currently at 22%. **cXprop** is open-source software and may be downloaded from <http://www.cs.utah.edu/~coop/research/cxprop>.

References

- [1] C. Scott Ananian and Martin Rinard. Data size optimizations for Java programs. In *Proc. of the 2003 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 59–68, San Diego, CA, June 2003.
- [2] Lan S. Bai, Lei Yang, and Robert P. Dick. Automated compile-time and run-time techniques to increase usable memory in MMU-less embedded systems. In *Proc. of the Intl. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, Seoul, Korea, October 2006.

- [3] Max Baron and Cheryl Cadden. Strong growth to continue for MCU market, 2005. <http://www.instat.com/press.asp?ID=1445&sku=IN0502457SI>.
- [4] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207, San Diego, California, USA, June 7–14 2003. ACM Press.
- [5] Dennis Brylow, Niels Damgaard, and Jens Palsberg. Static checking of interrupt-driven software. In *Proc. of the 23rd Intl. Conf. on Software Engineering (ICSE)*, pages 47–56, Toronto, Canada, May 2001.
- [6] Elaine Cheong, Judy Liebman, Jie Liu, and Feng Zhao. Tinygals: a programming model for event-driven embedded systems. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 698–704, New York, NY, USA, 2003. ACM Press.
- [7] Nathan Coopriider, Will Archer, Eric Eide, David Gay, and John Regehr. Efficient Memory Safety for TinyOS. In *Proc. of the 5th ACM Conf. on Embedded Networked Sensor Systems (SenSys 2007)*, Sydney, Australia, November 2007.
- [8] Nathan Coopriider and John Regehr. cXprop: Postpass optimization for tinyos applications. In *TinyOS Technology Exchange (TTX)*, Stanford, CA, February 2006.
- [9] Nathan Coopriider and John Regehr. Pluggable abstract domains for analyzing embedded software. In *Proc. of the 2006 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 44–53, Ottawa, Canada, June 2006.
- [10] Nathan Coopriider and John Regehr. Offline compression for on-chip RAM. In *Proc. of the ACM SIGPLAN 2007 Conf. on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2007.
- [11] 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.
- [12] Usit Duongsaa. Automatic generation of abstract transfer functions. Master's thesis, University of Utah, July 2005.
- [13] 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.
- [14] 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.
- [15] 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.
- [16] 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>.
- [17] Reinhold Heckmann and Christian Ferdinand. Stack usage analysis and software visualization for embedded processors. In *Proc. of the Embedded Intelligence Congress*, Nuremberg, Germany, February 2002.
- [18] John L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7), July 2000.

- [19] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 93–104, Cambridge, MA, November 2000.
- [20] Samuel P. Harbison III and Guy L. Steele Jr. *C A Reference Manual*. Prentice Hall, Uper Saddle River, NJ 07458, fifth edition, 2002.
- [21] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shiuan Peh, and Daniel Rubenstein. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebrantet. volume 37, pages 96–107, New York, NY, USA, 2002. ACM Press.
- [22] Edward A. Lee. Overview of the Ptolemy project. Technical Memorandum UCB/ERL M01/11, University of California at Berkeley, March 2001.
- [23] Edward A. Lee and Alberto Sangiovanni-Vincentelli. The tagged signal model - a preliminary version of a denotational framework for comparing models of computation. Technical report, UC Berkeley, 1996.
- [24] 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 Networks Group, Technische Universität Berlin, November 2005.
- [25] Jun Li. Monitoring and characterization of component-based systems with global causality capture. In *Proc. of the 23rd International Conference on Distributed Computing Systems (ICDCS)*, pages 422–431, 2003.
- [26] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, David Culler, and John Anderson. Wireless sensor networks for habitat monitoring. In *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA '02)*, Atlanta, GA, September 2002.
- [27] Florian Martin. PAG—An efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.
- [28] K. Martinez, R. Ong, and J. Hart. Glacsweb: a sensor network for hostile environments. In *First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks (SECON)*, pages 81–87, October 2004.
- [29] 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.
- [30] The Paparazzi project, 2006. <http://www.nongnu.org/paparazzi>.
- [31] 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.
- [32] Harini Ramaprasad and Frank Mueller. Tightening the bounds on feasible preemption points. In *Proc. of the 27th IEEE International Real-Time Systems Symposium (RTSS)*, pages 212–224, 2006.
- [33] John Regehr, Nathan Cooperider, Will Archer, and Eric Eide. Efficient type and memory safety for tiny embedded systems. In *Proc. of the 3rd Workshop on Linguistic Support for Modern Operating Systems (PLOS)*, San Jose, CA, October 2006.
- [34] John Regehr, Nathan Cooperider, Will Archer, and Eric Eide. Memory safety and untrusted extensions for TinyOS. Technical Report UUCS-06-007, University of Utah, June 2006.

- [35] 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.
- [36] John Regehr and Alastair Reid. Lock inference for systems software. In *Proc. of the Second AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 1–6, Boston, MA, March 2003.
- [37] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. In *Proc. of the 3rd Intl. Conf. on Embedded Software (EMSOFT)*, pages 306–322, Philadelphia, PA, October 2003.
- [38] John Regehr, Alastair Reid, Kirk Webb, Michael Parker, and Jay Lepreau. Evolving real-time systems using hierarchical scheduling and concurrency analysis. In *Proc. of the 24th IEEE Real-Time Systems Symposium (RTSS '03)*, pages 25–36, Cancun, Mexico, December 2003.
- [39] Alastair Reid and John Regehr. Task/scheduler logic: Reasoning about concurrency in component-based systems software. Technical report, September 2002. <http://www.cs.utah.edu/~regehr/papers/ts1/ts1-pdf.pdf>.
- [40] Martin C. Rinard. Analysis of multithreaded programs. In *Proc. of the 8th Static Analysis Symposium*, Paris, France, July 2001.
- [41] Semiconductor Industry Association (SIA). SIA forecast: Microchip industry will reach \$321 billion in 2009. Press release, November 2006. http://www.sia-online.org/pre_release.cfm?ID=420.
- [42] Gyula Simon, Miklós Maróti, Ákos Lédeczi, Görgy Balogh, Branislav Kusy, András Nádas, Gábor Pap, János Sallai, and Ken Frampton. Sensor network-based countersniper system. In *Proc. of the Conf. on Embedded Networked Sensor Systems (SenSys)*, pages 1–12, Baltimore, MD, November 2004.
- [43] 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.
- [44] Michael Stalkerich, Christian Wawersich, Wolfgang Schröder-Preikschat, Andreas Gal, and Michael Franz. An OSEK/VDX API for Java. In *Proc. of the 3rd Workshop on Programming Languages and Operating Systems (PLOS)*, San Jose, CA, October 2006.
- [45] Ben L. Titzer. Virgil: Objects on the head of a pin. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Portland, OR, October 2006.
- [46] 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.
- [47] Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proc. of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.
- [48] Lei Yang, Robert P. Dick, Haris Lekatsas, and Srimat Chakradhar. On-line memory compression for embedded systems. *ACM Trans. Embedded Computing Systems (TECS)*, 2006.
- [49] Ye Zhou. *Interface Theories for Causality Analysis in Actor Networks*. PhD thesis, EECS Department, University of California, Berkeley, May 2007.
- [50] Ye Zhou and Edward A. Lee. A causality interface for deadlock analysis in dataflow. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 44–52, New York, NY, USA, 2006. ACM Press.

A Value-flow abstract domain interface

```
(*****  
 * A generic signature for abstract data and operations  
 *****)  
  
(* NOTE: We adopt the convention from the dataflow literature where  
   top is the initial optimistic machine state while bottom is where  
   we know nothing. I.e. conc(top) is the empty set and conc(bottom)  
   is universal set of values. *)  
  
open DomainUtils  
  
(* abstract data value *)  
type t  
  
(* the bottom of the lattice *)  
val bottom : t  
  
(* lifts a cil expression into the abstract domain *)  
val conc_to_abs : Cil.exp -> t  
  
(* concretize an abstract value if the size of the concretization set is 1 *)  
val abs_to_conc : t -> Cil.typ -> Cil.exp  
  
val abs_to_triBool : t -> triBool  
  
(* domain throws this when the framework calls abs_to_conc on an abstract  
   value whose concretization set has cardinality != 1 *)  
exception Not_a_singleton  
  
(* domain throws this when a function is called that isn't yet implemented *)  
exception Not_implemented  
  
(* domain throws this when CIL does something that the domain believes  
   to be nonsensical, such as calling a transfer function with  
   arguments that are two different types *)  
exception Cil_inconsistent  
  
(* we are not sure what happened but something is very wrong *)  
exception Bug  
  
(* For some reason, the abstract domain ended up with Top (probably join) *)  
exception Top  
  
(* join: return most-general abstract value that is greater than both args *)  
val join : t -> t -> t  
  
(* we require abstract values to be ordered to we can make efficient sets of them *)  
val compare : t -> t -> int  
  
(* meet: return most-precise abstract value that is less than both arguments *)  
val meet : t -> t -> t  
  
(* accelerate termination by moving a value towards bottom *)  
val widen : t -> t  
  
(* create a C expression that can be used to dynamically test for sound abstraction *)
```

```

val cil_assertion : Cil.lval -> t -> Cil.exp

(* pretty-print abstract value *)
val to_string : t -> string

(* returns true if this type is not handled by the domain, false otherwise *)
val bad_type : Cil.typ -> bool

(* Useful for getting statistics on a domain *)
val info_bits_known : t -> int -> float

(* Useful for getting statistics on a domain *)
val bits_used : t -> int -> bool -> int

(* Used for bit compression. *)
val bit_compression : t -> Cil.exp list

(* UnOps *****)
val cast : t -> Cil.typ -> Cil.typ -> t

(* unary minus *)
val neg : t -> Cil.typ -> t

(* bitwise complement (~) *)
val bnot : t -> Cil.typ -> t

(* logical not (!) *)
val lnot : t -> Cil.typ -> triBool

(* BinOps *****)
(* arithmetic + *)
val plusa : t -> t -> Cil.typ -> t

(* arithmetic -*)
val minusa : t -> t -> Cil.typ -> t

(* * *)
val mult : t -> t -> Cil.typ -> t

(* / *)
val div : t -> t -> Cil.typ -> t

(* % *)
val amod : t -> t -> Cil.typ -> t

(* shift left *)
val shiftlt : t -> t -> Cil.typ -> t

(* shift right *)
val shiftrt : t -> t -> Cil.typ -> t

(* bitwise and *)
val band : t -> t -> Cil.typ -> t

(* exclusive-or *)
val bxor : t -> t -> Cil.typ -> t

```

```

(* inclusive-or *)
val bor : t -> t -> Cil.typ -> t

(* Comparisons *****)

(* < *)
val lt : t -> t -> Cil.typ -> triBool

(* > *)
val gt : t -> t -> Cil.typ -> triBool

(* <= *)
val le : t -> t -> Cil.typ -> triBool

(* >= *)
val ge : t -> t -> Cil.typ -> triBool

(* == *)
val eq : t -> t -> Cil.typ -> triBool

(* != *)
val ne : t -> t -> Cil.typ -> triBool

(* logical and. Unlike other expressions this one does not always
   evaluate both operands. If you want to use these, you must set
   Cil.useLogicalOperators. *)
val and : t -> t -> Cil.typ -> triBool

(* logical or. Unlike other expressions this one does not always
   evaluate both operands. If you want to use these, you must set
   Cil.useLogicalOperators. *)
val or : t -> t -> Cil.typ -> triBool

(*****)

(* looks at Cil.exp that evaluates to bool, then gets as much
   information as possible from it based on this domain *)

(* the (Cil.exp -> t) function is useful because the abs domain
   doesn't have anyway of evaluating an expression based on a state *)

(* < *)
val backwards_lt : t -> t -> Cil.typ -> (Cil.exp -> t) -> (triBool * (t * t) * (t * t))

(* > *)
val backwards_gt : t -> t -> Cil.typ -> (Cil.exp -> t) -> (triBool * (t * t) * (t * t))

(* <= *)
val backwards_le : t -> t -> Cil.typ -> (Cil.exp -> t) -> (triBool * (t * t) * (t * t))

(* >= *)
val backwards_ge : t -> t -> Cil.typ -> (Cil.exp -> t) -> (triBool * (t * t) * (t * t))

(* == *)
val backwards_eq : t -> t -> Cil.typ -> (Cil.exp -> t) -> (triBool * (t * t) * (t * t))

(* != *)
val backwards_ne : t -> t -> Cil.typ -> (Cil.exp -> t) -> (triBool * (t * t) * (t * t))

```

```
(* != *)  
val backwards_mod : t -> t -> Cil.typ -> (Cil.exp -> t) -> (triBool * (t * t) * (t * t))
```