Atomicity and Visibility in Tiny Embedded Systems

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

David Gay
Intel Research Berkeley
david.e.gay@intel.com

Abstract

Visibility is a property of a programming language's memory model that determines when values stored by one concurrent computation become visible to other computations. Our work exploits the insight that in nesC, a C-like language with explicit atomicity, the traditional way of ensuring timely visibility—volatile variables—can be entirely avoided. This is advantageous because the volatile qualifier is a notorious source of programming errors and misunderstandings. Furthermore, the volatile qualifier hurts performance by inhibiting many more optimizations than are necessary to ensure visibility. In this paper we extend the semantics of nesC's atomic statements to include a visibility guarantee, we show two ways that these semantics can be implemented, and we also show that our better implementation has no drawbacks in terms of resource usage.

1. Introduction

The nesC [5] language is a C dialect designed for programming embedded wireless sensor network nodes. It supports safe concurrent programming using an `atomic` construct. This construct guarantees that a statement "is executed 'as-if' no other computation occurred simultaneously." These semantics do not address *visibility*, which Alexandrescu et al. [1] define as addressing the question: "Under what conditions will the effects of a write action by one thread be seen by a read from another thread?" In situations where a computation spins waiting for the result of another computation, visibility errors lead to deadlocks. In other situations data corruption is possible.

Atomicity on uniprocessor embedded systems is easily implemented by disabling interrupts. This implementation is extremely lightweight compared to software transactional memory, but it lacks a visibility guarantee. Since small uniprocessor embedded systems do not have caches, compiler optimizations are the only issue that must be considered when making visibility guarantees.

Until now, nesC has left it to programmers to ensure visibility by judicious use of C's `volatile` qualifier. In this paper we investigate the idea of strengthening nesC's memory model such that the semantics of `atomic` are augmented with a visibility guarantee. This change results in a novel design point for lightweight atomicity in a C-like language that reduces programmers' effort and opportunities to make subtle errors. We present, and evaluate the performance and code size impact of, two implementations of the new semantics.

PLOS'06, October 22, 2006, Santa Jose, California, U.S.A.

ACM 1-59593-577-0/10/2006.

```
bool flag = false;

// interrupt handler
void __vector_5 (void)
{
  atomic flag = true;
}

void wait_for_interrupt(void)
{
  bool done = false;
  do {
    atomic if (!flag) done = true;
  } while (!done);
}
```

Figure 1. nesC code containing a visibility error

2. Problems with `volatile`

In C and nesC, a volatile variable is one where every source-level read or write is guaranteed to correspond to a load from or store to a physical memory location. Furthermore, the compiler must not reorder these loads and stores. In effect, volatile variables are exempt from most compiler optimizations. Volatiles are used to implement communication between concurrent flows and also to ensure that hardware registers are accessed properly.

Figure 1 shows nesC code that is unlikely to work. When this code is compiled for two popular sensor network platforms, Mica2 and TelosB, `wait_for_interrupt` spins on a register instead of on the actual flag variable. The problem would not have occurred if the developer had declared `flag` as volatile.

Our experience is that developers do not do a good job of declaring variables as volatile. This problem is common enough to appear at the top of the "frequently asked question" list for the AVR port of the GNU C Library [2]. Similarly, a simple inspection of the code base of TinyOS 1.x [6], an operating system for wireless sensor network nodes written in nesC, shows that volatile is used to declare 18 variables in 157k lines of code. Clearly, most shared data is not declared volatile. The data in Table 1, described in more detail in Section 5, support this claim. Part of the problem is that there are a variety of situations in which code may work even when a shared variable is not declared volatile. For example, a register may not be available in which to cache a shared variable. Similarly, the scope in which a shared variable is allocated to a register may end in time for a value to be pushed to RAM before it is needed. Our sense is that the latter case is most common in TinyOS applications, where an important synchronization idiom is for an interrupt handler to *post* a task that runs at a later time. The end of the interrupt handler serves as an implicit memory barrier, forcing

the results of its computations to be flushed to RAM. However, it is clear that in many cases, implicit visibility is fragile, and code making improper use of volatile can be broken by a change in compiler version, compiler flags, or phase of the moon.

Another problem with volatile variables is that they are overkill [9]: they affect all accesses to a variable, whereas visibility only requires that the final access in an atomic section be forced to RAM. Since they are such a blunt tool for inhibiting compiler optimizations, declaring too many variables as volatile hurts performance. Developers of resource constrained systems tend to use a trial-and-error approach to declaring volatiles: initially few variables are volatile, and then more volatile qualifiers are added in order to fix bugs that are observed during testing.

3. Visibility Semantics for nesC

nesC encourages the use of a static programming model with no dynamic memory allocation. Thus, in the rest of this paper we will discuss concurrency, atomicity and visibility in terms of variables rather than more general memory objects.

nesC's concurrency model divides global variables into three categories: synchronous, asynchronous, and racing. Synchronous variables are not modified by concurrent code. Asynchronous variables may be modified by concurrent code, but only inside atomic sections. Racing variables are those that may be concurrently modified from outside of an atomic section. Of these categories, we are interested only in asynchronous variables. Synchronous variables require no visibility guarantee, and racing variables have platform-dependent behavior.

We propose updating the semantics of nesC's `atomic` construct, so it guarantees that a statement

> ...is executed 'as-if' no other computation occurred simultaneously, and furthermore any values stored to asynchronous variables from inside an atomic statement are visible inside all subsequent atomic statements. No guarantee is made about racing variables.

Thus, as long as there are no racing variables, the programmer does not need to figure out where `volatile` annotations should be used.

4. Implementing Visibility

The nesC compiler produces a single C file as output, which is then passed to a regular C compiler. Atomic statements are compiled by calling target-specific intrinsic functions at the start and end of each atomic section. We extended this compiler with two implementations for our visibility semantics; both are straightforward.

The first is to add the volatile annotations that programmers tend to forget: we mark all asynchronous variables as volatile in the generated C code, and any pointers used to access them as pointer-to-volatile. This is clearly correct: all loads and stores to these variables are forced to go to RAM.

The second implementation is to ensure that the compiler does not perform the optimizations that break visibility: we strengthen the nesC intrinsic functions that start and end atomic sections (which until now simply disabled and enabled interrupts) with compiler-level memory barriers.[1] These barriers stop the compiler from retaining copies of variables in registers across a barrier: values are flushed to RAM before the barrier and reloaded afterwards.

[1] While memory barriers cannot be portably specified (the common idiom of calling an external function would not seem sufficient to prevent optimization of static global variables) they are supported by many C compilers. For instance, in gcc (used as a backend for all current nesC targets), a memory barrier can be written as

```
asm volatile("" : : : "memory");
```

application (loc)	sync	async	race	vol
`blinktask` (1871)	18	4	0	3
`osc` (5587)	34	25	7	3
`genericbase` (6941)	51	57	1	4
`rfmtoleds` (7401)	57	46	7	4
`cnttoledsandrfm` (7810)	68	50	8	4
`micahwverify` (8131)	68	50	8	4
`sensetorfm` (8259)	67	54	8	4
`testtimestamping` (8308)	60	57	22	5
`surge` (10657)	117	53	10	4
`ident` (11146)	101	50	8	4
`hfs` (12284)	118	51	17	4

Table 1. Kinds of variables in the TinyOS applications that we use to evaluate our work. The volatility of a variable is independent of whether it is synchronous, asynchronous, or racing.

Again we believe this implementation to be correct, even though the analogous use of memory barriers around mutex lock and unlock operations is not sufficient, as pointed out by Boehm [3]. We discuss this issue further under related work. Our second transformation enables a minor performance optimization where the volatile qualifier can be removed from asynchronous variables that have already been declared as volatile.

5. Evaluation

This section evaluates the effect of the visibility implementations on code size and duty cycle. Duty cycle is the percentage of application running time during which the CPU is active (as opposed to sleeping in order to save power). We measure duty cycles using Avrora [8], a cycle-accurate sensor network simulator.

The applications under consideration are a representative collection of example applications from the TinyOS 1.x CVS tree (available from `sourceforge.net`). Table 1 shows, for each application and the TinyOS components that it uses, the size (in lines) and number of synchronous, asynchronous, and racing variables. Note that these categories are mutually exclusive. The target platform is the Mica2 from Crossbow [4], a sensor network node based on Atmel's ATmega128, an 8-bit RISC processor. Our toolchain uses nesC 1.2.7a and gcc 3.4.3.

Figure 2 shows the impact of our transformations on object code size. The baseline for code size is the executable generated by the unmodified nesC toolchain. Making asynchronous variables volatile bloats code size by about 6%, on average. While this is not prohibitive it could be a problem for large applications. On the other hand, adding memory barriers to nesC's atomic intrinsics actually reduces code size slightly, on average. The cause for this effect is not totally clear.

Figure 3 shows the impact of our transformations on application duty cycles. Again the baseline is the duty cycle of the executable generated by the unmodified nesC toolchain. Adding the visibility guarantee has no clear effect in either direction on application performance, and the differences are at most a few percent.

6. Related Work

Boehm [3] discusses why C cannot reliably be used for threaded (concurrent) programming without compiler knowledge of threads. In particular, he shows that the common approach of ensuring that mutex lock and unlock operations contain compiler-level memory barriers is not sufficient to ensure correctness. Our modifications to nesC do not suffer from the three problems he identified. The requirement in nesC that all shared variables be updated within

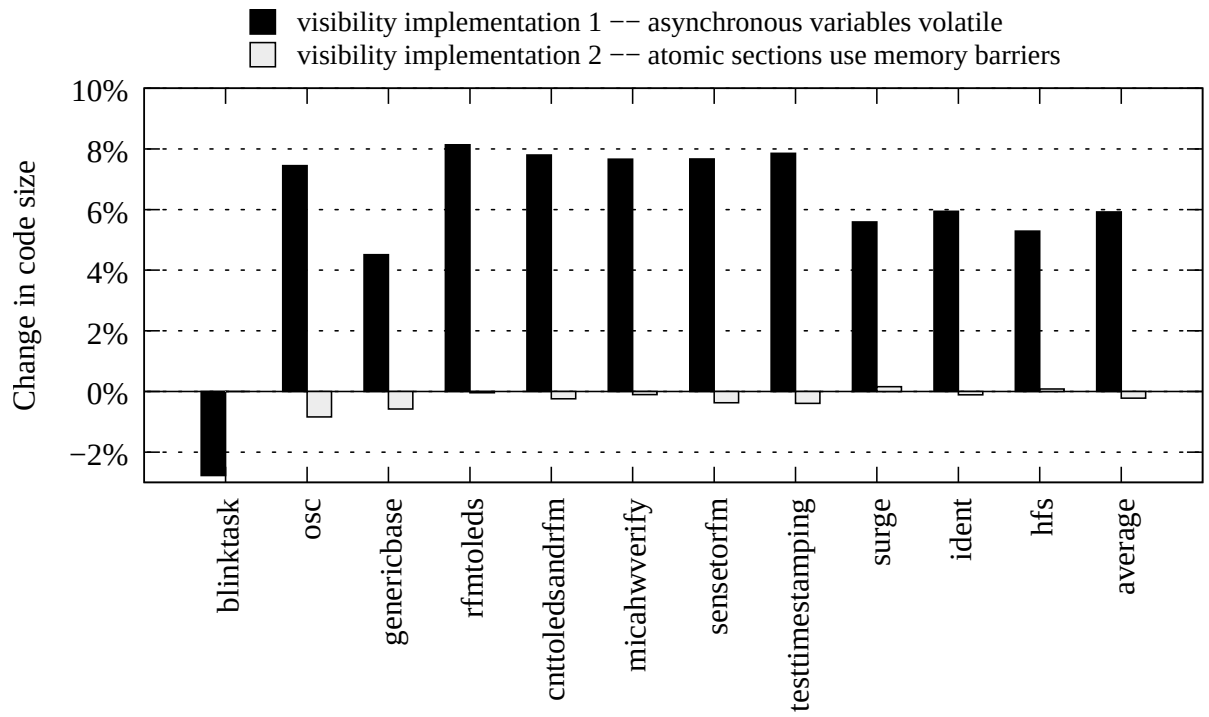

Figure 2. Change in code size relative to a baseline which does not provide visibility guarantees

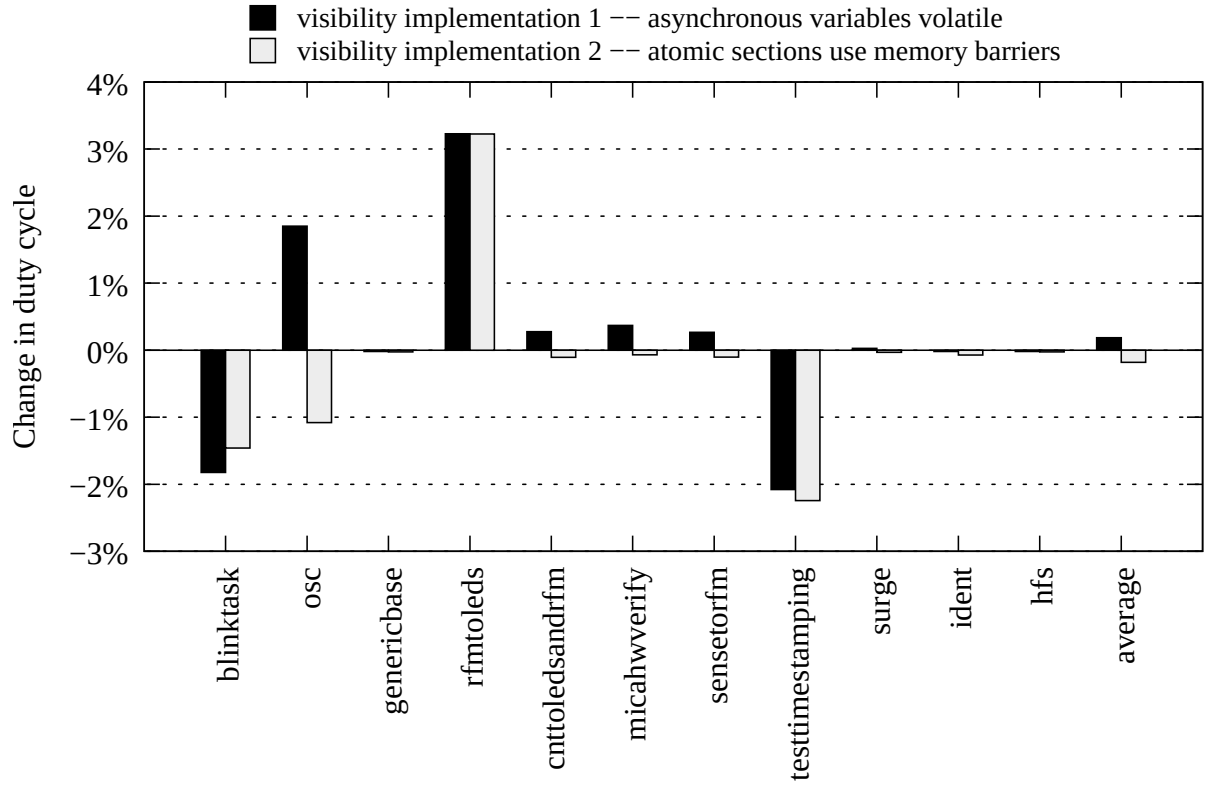

Figure 3. Change in duty cycle relative to a baseline which does not provide visibility guarantees

atomic sections avoids the "concurrent modification" (locks are only required if there are races, but compiler transformations may introduce races) and "rewriting of adjacent data" (e.g., accesses to different but adjacent bitfields in different threads) problems.[2] The fact that nesC's atomic sections are syntactic statements avoids the problem of optimizations around conditionally executed locking statements.

Sutter [7], in a work in progress, proposes a memory model based on the concept of "interlocked" memory locations and a set of eight rules that compilers must follow. An Interlocked variable is similar to a volatile variable, but is explicitly designed to support inter-thread communication. Interlocked reads and writes have "acquire" and "release" semantics on a thread's other writes, and may be optimized away in restricted circumstances. The rules are designed to ensure that causality is always respected, and that correctly locked programs behave as in a sequentially-consistent model. This proposal is more general but lower-level than our atomic section based approach: it supports more general inter-thread communication patterns (e.g., communication via a single interlocked variable), but is significantly more complex to understand and use. It would be a good target for alternative implementations of nesC's atomic sections, as revised in this paper.

7. Conclusion

Visibility of data modifications is an important part of correct concurrent programming. We observe that programmers of C-like languages typically fail to ensure visibility through the use of the `volatile` type qualifier; thus visibility should be ensured at the language level. We show that in nesC, a language targeted at resource-constrained wireless sensor network nodes, visibility can be guaranteed by a small change to the compiler, and without any significant performance impact.

References

[1] Andrei Alexandrescu, Hans Boehm, Kevlin Henney, Doug Lea, and Bill Pugh. Memory model for multithreaded C++, September 2004. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1680.pdf`.

[2] AVR C Runtime Library Project. AVR libc Frequently Asked Questions, 2006. `http://www.nongnu.org/avr-libc/user-manual/FAQ.html`.

[3] Hans-Juergen Boehm. Threads cannot be implemented as a library. In *Proc. of the ACM SIGPLAN 2005 Conf. on Programming Language Design and Implementation (PLDI)*, pages 261–268, Chicago, IL, June 2005.

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

[5] 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.

[6] 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.

[7] Herb Sutter. Prism: A principle-based sequential memory model for microsoft native code platforms, July 2006. Draft, available from `http://www.gotw.ca/memmodel/Prism%20-%20draft%200.8.pdf`.

[8] 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.

[9] Linus Torvalds. Anti-volatile diatribe, July 2006. Linux-Kernel Mailing List posting, `http://lkml.org/lkml/2006/7/6/159`.

[2] We do assume that updates to two variables adjacent in memory cannot cause data races. This is true of all platforms in widespread use that we know of. It wasn't true for, e.g., early versions of the Alpha microprocessor.