The nesC Language: A Holistic Approach to Networked Embedded Systems

http://nescc.sourceforge.net

ABSTRACT

We present nesC, a programming language for networked embedded systems that represent a new design space for application developers. An example of a networked embedded system is a sensor network, which consists of (potentially) thousands of tiny, low-power “motes,” each of which execute concurrent, reactive programs that must operate with severe memory and power constraints.

nesC’s contribution is to support the special needs of this domain by exposing a programming model that incorporates event-driven execution, a flexible concurrency model, and component-oriented application design. Restrictions on the programming model allow the nesC compiler to perform whole-program analyses, including data-race detection (which improves reliability) and aggressive function inlining (which reduces resource consumption).

nesC has been used to implement TinyOS, a small operating system for sensor networks, as well as several significant sensor applications. nesC and TinyOS have been adopted by a large number of sensor network research groups, and our experience and evaluation of the language shows that it is effective at supporting the complex, concurrent programming style demanded by this new class of deeply networked systems.

1. INTRODUCTION

Advances in networking and integration have enabled small, flexible, low-cost nodes that interact with their environment through sensors, actuators and communication. Single-chip systems are now emerging that integrate a low-power CPU and memory, radio or optical communication, and substantial MEMs-based on-chip sensors; these nodes are colloquially referred to as “motes” or “smart dust” [49]. Target costs (for single-chip motes) are less than 10 cents per unit, which enables networks with potentially tens of thousands of motes. Target power consumption means that motes can last years with low-bandwidth communication, or even be battery-free when fueled by ambient power (e.g., heat from the environment).

In this paper, we present nesC, a systems programming language for networked embedded systems such as motes. nesC supports a programming model that integrates reactivity to the environment, concurrency, and communication. By performing whole-program optimizations and compile-time data race detection, nesC simplifies application development, reduces code size, and eliminates many sources of potential bugs.

A key focus of nesC is holistic system design. Mote applications are deeply tied to hardware, and each mote runs a single application at a time. This approach yields three important properties. First, all resources are known statically. Second, rather than employing a general-purpose OS, applications are built from a suite of reusable system components coupled with application-specific code. Third, the hardware/software boundary varies depending on the application and hardware platform; it is important to design for flexible decomposition.

There are a number of unique challenges that nesC must address:

**Driven by interaction with environment:** Unlike traditional computers, motes are used for data collection and control of the local environment, rather than general-purpose computation. This focus leads to two observations. First, motes are fundamentally event-driven, reacting to changes in the environment (message arrival, sensor acquisition) rather than driven by interactive or batch processing. Second, event arrival and data processing are concurrent activities, demanding an approach to concurrency management that addresses potential bugs such as race conditions.

**Limited resources:** Motes have very limited physical resources, due to the goals of small size, low cost, and low power consumption. We do not expect new technology to remove these limitations: the benefits of Moore’s Law will be applied to reduce size and cost, rather than increase capability. Although our current motes are measured in square centimeters, a version has been built that measures less than 5 mm².

**Reliability:** Although we expect individual motes to fail due to hardware issues, we must enable very long-lived applications. For example, environmental monitoring applications must collect data without human interaction for months at a time. An important goal is to reduce run-time errors, since there is no real recovery mechanism in the field except for automatic reboot.

**Soft real-time requirements:** Although there are some tasks that are time critical, such as radio management or sensor polling, we do not focus on hard real-time guarantees. Our experience so far indicates that timing constraints are easily met by having complete control over the application and OS, and limiting utilization. One of the few timing-critical aspects in sensor networks is radio communication; however, given the fundamental unreliability of the radio link, it is not necessary to meet hard deadlines in this domain.

Although nesC is a synthesis of many existing language concepts
targeted at the above problems, it provides three broad contributions. First, nesC defines a component model that supports event-driven systems: the model provides bidirectional interfaces to simplify event flow, supports a flexible hardware/software boundary, and admits efficient implementation that avoids virtual functions and dynamic component creation. Second, nesC defines a simple but expressive concurrency model coupled with extensive compile-time analysis: the nesC compiler detects most data races at compile time. This combination allows applications to exhibit highly concurrent behavior with very limited resources. Third, nesC provides a unique balance between accurate program analysis to improve reliability and reduce code size, and expressive power for building real applications. In addition to static data race detection, the nesC compiler performs static component instantiation, whole-program inlining, and dead-code elimination. We prohibit many features that hinder static analysis, including function pointers and dynamic memory allocation, but are capable of supporting complex applications and a substantial user community.

nesC is used as the programming language for TinyOS [16], a small operating system for sensor network applications that is in use by more than 100 research groups worldwide. Several significant sensor network applications have been implemented in nesC, including TinyDB [29], a sensor network query processing engine, and Maté [28], a small virtual machine that allows rapid reprogramming of sensor networks. Section 2 presents background material on sensor networks and introduces Surge, a sample nesC application used as a running example throughout the paper. Section 3 presents the nesC design; Section 4 summarizes our experience with nesC and evaluates the effectiveness of data-race detection and inlining. We conclude with a survey of related work (Section 5) and a discussion of nesC and its future directions (Section 6).

2. BACKGROUND

Wireless sensor networks are composed of large numbers of tiny resource-limited devices (motes). A first application of these networks is data collection in uncontrolled environments, such as nature reserves [30] or seismically threatened structures [25]. Four key features have emerged in networks we have deployed: interaction with the local environment through sensors, communication via a wireless network, lifetime requirements of months to a year, and physical inaccessibility.

Table 1 presents several generations of motes designed at UC Berkeley. Although very resource constrained, motes must be very reactive and participate in complex distributed algorithms, such as data aggregation [19, 29] or spatial localization [50]. This combination of requirements makes traditional operating systems and programming models inappropriate for sensor networks. Mote hardware evolves rapidly: Table 1 covers five platforms in three years, with different sensors and varying levels of hardware support for operations such as radio-based messaging. A sensor network operating system and programming language must make it easy for applications to adapt to these changes.

2.1 TinyOS

TinyOS [16] is an operating system specifically designed for network embedded systems. TinyOS has a programming model tailored for event-driven applications as well as a very small footprint (the core OS requires 400 bytes of code and data memory, combined). Two of our motivations in designing nesC were to support and evolve TinyOS’s programming model and to reimplement TinyOS in the new language. TinyOS has several important features that influenced nesC’s design: a component-based architecture, a simple event-based concurrency model, and split-phase operations.

Component-based architecture: TinyOS provides a set of reusable system components. An application connects components using a wiring specification that is independent of component implementations; each application customizes the set of components it uses. Although most OS components are software modules, some are thin wrappers around hardware; the distinction is invisible to the developer. Decomposing different OS services into separate components allows unused services to be excluded from the application. We present nesC’s support for components in Sections 3.1 and 3.2.

Tasks and event-based concurrency: There are two sources of concurrency in TinyOS: tasks and events. Tasks are a deferred computation mechanism. They run to completion and do not preempt each other. Components can post tasks; the post operation immediately returns, deferring the computation until the scheduler executes the task later. Components can use tasks when timing requirements are not strict; this includes nearly all operations except low-level communication. To ensure low task execution latency, individual tasks must be short; lengthy operations should be spread across multiple tasks. The lifetime requirements of sensor networks prohibit heavy computation, keeping the system reactive.

Events also run to completion, but may preempt the execution of a task or another event. Events signify either completion of a split-phase operation (discussed below) or an event from the environment (e.g. message reception or time passing). TinyOS execution is ultimately driven by events representing hardware interrupts. We discuss the refinement and inclusion of TinyOS’s concurrency model into nesC in Sections 3.3 and 3.4.

Split-phase operations: Because tasks execute non-preemptively, TinyOS has no blocking operations. All long-latency operations are split-phase: operation request and completion are separate functions. Commands are typically requested to execute an operation. If the operation is split-phase, the command returns immediately and completion will be signaled with an event; non-split-phase operations (e.g. toggle an LED) do not have completion events. A typical example of a split-phase operation is a packet send: a component may invoke the send command to initiate the transmission of a radio message, and the communication component signals the sendDone event when transmission has completed. Each component implements one half of the split-phase operation and calls the other; the wiring connects both commands and events across component boundaries. We discuss how nesC captures split-phase operations in Section 3.1.

Resource contention is typically handled through explicit rejection of concurrent requests. In the example above, if the communication component cannot handle multiple concurrent send operations, it signals an error when a concurrent send is attempted.
Alternately, the communication component could queue the request for future processing.

The simple concurrency model of TinyOS allows high concurrency with low overhead, in contrast to a thread-based concurrency model in which thread stacks consume precious memory while blocking on a contended service. However, as in any concurrent system, concurrency and non-determinism can be the source of complex bugs, including deadlock (e.g., the Mars Rover [22]) and data races (e.g., the Therac-25 [27]). One of the primary benefits of nesC is helping the programmer use concurrency safely by ensuring the absence of (most) data races. This is discussed in Section 3.3.

2.2 Surge: A Sensor Network Application

A common application of sensor networks is to sample a sensor periodically (e.g., light or temperature) and report readings to a base station, which is typically a node with a wired network connection and power source. As a running example throughout this paper, we present Surge, a simple application that performs periodic sensor sampling and uses ad-hoc multi-hop routing over the wireless network to deliver samples to the base station. Surge is intentionally simple and does not perform advanced functions such as in-network data aggregation [19, 29].

Surge motes organize themselves into a spanning tree rooted at the base station. Each mote maintains the address of its parent and its depth in the tree, advertising its depth in each radio message (either sensor sample or forwarded message) that it transmits. A node selects an initial parent by listening to messages and choosing the node with the smallest depth; to seed the creation of the spanning tree, the base station periodically broadcasts beacon messages with depth 0. Nodes estimate parent link quality; when the link quality falls below some threshold, nodes select a new parent from their neighbor set based on link-quality estimates and depth.

Once a second, each mote samples its light sensor and sends the sample to its parent. Parents acknowledge received packets. Surge uses the acknowledgments to provide a reliable transport layer; parent link quality is calculated as the fraction of transmitted messages that are acknowledged. When a node receives a message from another node, it forwards the message to its parent. Sensor samples are collected at the base station where they can be analyzed or visualized.

Figure 1 shows the components of Surge and the interfaces by which they are wired. This example shows several important advantages of components. The Surge application only needs to include the parts of TinyOS that it needs, i.e., system boot code (Main), the timer (Timer), a sensor (Photo), access to the LEDs (Leds), and multi-hop message routing (Multihop). Secondly, the application code – the Surge component – explicitly specifies its environmental dependencies in terms of interfaces. Surge requires a timer (Timer interface), sensor (ADC interface), LEDs (Leds interface), and communication (Send interface). The code is therefore independent of the particular sensor hardware used. For example, ADC could easily be wired to a temperature sensor rather than a light sensor, as long as the two provide the same interface.

3. nesC DESIGN

In this section we discuss the primary concepts in nesC’s design. First, nesC applications are built out of components with well-defined, bidirectional interfaces. Second, nesC defines a concurrency model, based on tasks and events, and detects data races at compile time.

A few basic principles underlie nesC’s design:

nesC is an extension of C: C [23] produces efficient code for all the target microcontrollers that are likely to be used in sensor networks. C provides all the low-level features necessary for accessing hardware, and interaction with existing C code is simplified. Last but not least, many programmers are familiar with C.

C does have significant disadvantages: it provides little help in writing safe code or in structuring applications. nesC addresses safety through reduced expressive power and structure through components. None of the new features in nesC are tied to C: the same ideas could be added to other imperative programming languages such as Modula-2 [52].

Whole-program analysis: nesC programs are subject to whole program analysis (for safety) and optimization (for performance). Therefore we do not consider separate compilation in nesC’s design. The limited program size on motes makes this approach tractable.

nesC is a “static language”: There is no dynamic memory allocation and the call-graph is fully known at compile-time. These restrictions make whole program analysis and optimization significantly simpler and more accurate. They sound more onerous than they are in practice: nesC’s component model and parameterized interfaces eliminate many needs for dynamic memory allocation and dynamic dispatch. We have, so far, implemented one optimization and one analysis: a simple whole-program inliner and a data-race detector. Details are given in Section 4.

nesC supports and reflects TinyOS’s design: nesC is based on the concept of components, and directly supports TinyOS’s event-based concurrency model. Additionally, nesC explicitly addresses the issue of concurrent access to shared data (Section 3.3). In practice, nesC resolved many ambiguities in the TinyOS concepts of components and concurrency, and TinyOS evolved to the nesC versions as it was reimplemented.

3.1 Component Specification

nesC applications are built by writing and assembling components. A component provides and uses interfaces. These interfaces are the only point of access to the component. An interface generally models some service (e.g., sending a message) and is specified by an interface type. Figure 2 shows the Timer component, part of the TinyOS timer service, that provides the StdControl and Timer interfaces and uses a Clock interface (all shown in Fig-
Components are also a clean way to abstract the boundary between hardware and software. For instance, on one sensor board, the temperature sensor (accessed via a component named Temp) is mostly in hardware; Temp is a thin layer of software accessing on-chip hardware registers. On another it is accessed over an I²C bus; Temp is implemented as a number of interacting components including a generic I²C access component. A subtle but important point is that bidirectional interfaces make it very easy to support hardware interrupts. In contrast, one-way interfaces based on procedure calls force hardware polling or having two separate interfaces for hardware operations and the corresponding interrupts.

### 3.2 Component Implementation

There are two types of components in nesC: modules and configurations. Modules provide application code, implementing one or more interfaces. Configurations are used to wire other components together, connecting interfaces used by components to interfaces provided by others. Every nesC application is described by a top-level configuration that wires together the components used. The body of a module is written in C-like code, with straightforward extensions. A command or event f in an interface i is named \textit{i.f}. A command \textit{call} is like a regular function call prefixed with the keyword \textit{call}, similarly an event \textit{signal} is like a function call prefixed by \textit{signal}. The definition of a commands or event named \textit{i.f} is prefixed with \textit{command} or \textit{event}. We require these annotations to improve code clarity. Figure 4 is a simplified excerpt from SurgeM, which is part of the Surge application. It defines the \texttt{StdControl.init} command, called at boot-time, and two of the events handled by Surge: the firing of the timer (\texttt{Timer.fired}) and sensor data acquisition (\texttt{ADC.dataReady}). The code calls the \texttt{Timer.start} command to setup periodic timer events and the \texttt{ADC.getData} command to request a new sensor sample. Modules have private state, in this example the \texttt{sensorReading} variable.

**Figure 4:** Simplified excerpt from SurgeM.

```c
module SurgeM {
    provides interface StdControl;
    uses interface ADC;
    uses interface Timer;
    uses interface Send;
}

implementation {
    uint16_t sensorReading;

    command result_t StdControl.init() {
        return call Timer.start(TIMER_REPEAT, 1000);
    }

    event result_t Timer.fired() {
        call ADC.getData();
        return SUCCESS;
    }

    event result_t ADC.dataReady(uint16_t data) {
        sensorReading = data;
        ... send message with data in it ... return SUCCESS;
    }
}
```

**Figure 2:** Specification and graphical depiction of the TimerM component.

```
module TimerM {
    provides {
        interface StdControl;
        interface Timer;
    }
    uses interface Clock as Clk;
}... 
```

**Figure 3:** Some interface types.
Data races occur due to concurrent updates to shared state. In order to prevent them, a compiler must 1) understand the concurrency model, and 2) determine the target of every update. In this section we present the concurrency model and the key invariant that the compiler must enforce to avoid data races. We achieve tractable target analysis by reducing the expressive power of the language and performing alias analysis. In particular, nesC has no dynamic memory allocation and no function pointers.

In TinyOS, code runs either asynchronously in response to an interrupt, or in a synchronously scheduled task. To facilitate the detection of race conditions, we distinguish synchronous and asynchronous code:

**Asynchronous Code (AC):** code that is reachable from at least one interrupt handler.

**Synchronous Code (SC):** code that is only reachable from tasks.

The run-to-completion rule and sequential execution of tasks lead immediately to a key invariant:

**Invariant:** Synchronous Code is atomic with respect to other Synchronous Code.

By “atomic,” we mean that any shared state between the two will be updated atomically. This essentially provides atomicity by default for tasks. Code that includes split-phase operations, which by definition must include (at least) two tasks, is not atomic as a whole, although each half is atomic. We discuss larger units of atomicity in Section 6.

Although non-preemption allows us to avoid races among tasks, there are still potential races between SC and AC, as well as AC and AC. We claim that:

**Claim 1:** Any update to shared state from AC is a potential race condition.

**Claim 2:** Any update to shared state from SC that is also updated from AC is a potential race condition.

To reinstate atomicity in these cases, the programmer has two options: either to convert all of the sharing code to tasks (SC only), or to use atomic sections to update the shared state. An atomic section is a small code sequence that nesC ensures will run atomically. We present the syntax and implementation issues for atomic sections in Section 3.4. We require that any update to shared state that is a potential race condition based on the claims above, must occur within an atomic section. This gives us our basic invariant.
module SurgeM { ... } 
implementation {
  bool busy;
  norace uint16_t sensorReading;
  event result_t Timer.fired() {
    bool localBusy;
    atomic {
      localBusy = busy; 
      busy = TRUE;
    }
    if (!localBusy)
      call ADC.getData();
    return SUCCESS;
  }
  task void sendData() { // send sensorReading
    ADCPacket.data = sensorReading;
    ADC.getData();
    call Send.send(adcPacket, sizeof adcPacket.data);
    return SUCCESS;
  }
  event result_t ADC.dataReady(uint16_t data) {
    sensorReading = data;
    post sendData();
    return SUCCESS;
  }
  ... 
}

Figure 7: Concurrency and atomicity in SurgeM. Changes from Figure 4 are highlighted.

Race-Free Invariant: Any update to shared state is either not a potential race condition (SC only), or occurs within an atomic section.

In nesC, we enforce this invariant at compile time, which avoids most data races. Note that this invariant only guarantees that individual accesses are race-free; incorrect use of atomic sections can still lead to race conditions.

3.4 Concurrency in nesC

Concurrency is central to nesC components: events (and commands) may be signaled directly or indirectly by an interrupt, which makes them asynchronous code. To handle this concurrency, nesC provides two tools: atomic sections and tasks. Figure 7 illustrates their use, showing the core logic of the Surge application. Here, Timer.fired and ADC.dataReady are asynchronous code.

The Timer.fired event is signaled periodically. If Surge is not already busy, ADC.getData is called to get a new sensor value. As busy is accessed in asynchronous code, its use is protected by an atomic statement that performs a test-and-set on busy. When the sensor value is available, the ADC.dataReady event is signaled. Sending the message with the sensor reading is not a time-critical operation, and the TinyOS communication layer is not designed to be executed as asynchronous code. Therefore, SurgeM posts a task, sendData, which sends the sensor reading message. Posted tasks are executed by the TinyOS scheduler when the processor is idle.

We currently implement atomic by disabling and enabling interrupts, which has very low overhead (a few cycles). However, leaving interrupts disabled for a long period delays interrupt handling, which makes the system less responsive. To minimize this effect, atomic statements are not allowed to call commands or signals, which makes the system less responsive. To minimize this effect, atomic statements are not allowed to call commands or signals, which makes the system less responsive. To minimize this effect, atomic statements are not allowed to call commands or signals, which makes the system less responsive. To minimize this effect, atomic statements are not allowed to call commands or signals, which makes the system less responsive. To minimize this effect, atomic statements are not allowed to call commands or signals, which makes the system less responsive. To minimize this effect, atomic statements are not allowed to call commands or signals, which makes the system less responsive. To minimize this effect, atomic statements are not allowed to call commands or signals, which makes the system less responsive. To minimize this effect, atomic statements are not allowed to call commands or signals, which makes the system less responsive.

As discussed in Section 3.3, atomic prevents concurrent access to the shared data accessed in the statement. Given atomic sections, we define the key rule that enforces our race-free invariant:

If a variable \( x \) is accessed by AC, then any access of \( x \) outside of an atomic statement is a compile-time error.

The compiler reveals the specific conflict in addition to signaling the error. To remove the error, the programmer must either add an atomic section, or move the offending code into a task.

There are some cases in which there is a potential data race on a variable that the programmer knows is not an actual data race, for instance sensorReading in Figure 7. The declaration of sensorReading includes the norace qualifier to suppress errors about this particular false positive. This avoids uselessly protecting all accesses to sensorReading with atomic statements. Section 4.2 shows that we can effectively detect a large number of data races.

3.5 Parameterized Interfaces

Parameterized interfaces are nesC’s mechanism for introducing runtime command and event dispatch within a first order language. A component declares an interface with a parameter list which creates a separate interface for each tuple of parameter values. Parameterized interfaces are used to model Active Messages [47] in TinyOS: in Active Messages, packets contain a numeric identifier that specifies which event handler should be executed. Figure 8 shows a very simplified definition of the communication component (GenericComm). Wires to a parameterized interface must specify a specific interface with a compile-time constant: the multihop message routing uses Active Message 42 for all its communication, hence wires to GenericComm.Send[42] in Figure 8.

In a module, the implemented commands and events of a parameterized interface receive extra parameters specifying the selected interface (see Send.send in the ActiveMessages module of Figure 8) and select a specific interface when invoking a command or event in a parameterized interface (see sendComplete). This last construction translates into a runtime dispatch to one of the functions connected to Send.sendDone by the application’s configurations. However the set of possible dispatch targets is explicitly specified by the program’s configurations. An additional benefit is that no RAM is needed to store the active message dispatch table.

4. EVALUATION

In this section we evaluate nesC’s component model, concurrency model, and whole-program inlining with respect to a set of...
representative TinyOS applications, including Surge, TinyDB and Maté. We have implemented a compiler for nesC that generates a single C source file for a whole application, resolving all interface connections to direct function calls. In the results below, this C file is compiled with gcc 3.1.1 for the Atmel ATmega 103, an 8-bit RISC microprocessor used on Mica mote (Table 1).

### 4.1 Component Model

Anecdotally, nesC’s component model has been invaluable for event-driven sensor applications. The success of the component model is shown by the way in which components are used in the TinyOS code; applications are small, and make use of a large number of reusable components. Moreover, nesC’s component model makes it possible to pick and choose which parts of the OS are included with each application.

For the purposes of the discussion below, we divided the TinyOS source into application code and core OS code. We consider core OS code to be fundamental things such as the TinyOS scheduler and radio stack as well as commonly used support routines for timers, sensor management, etc. All other code is considered to be application specific. Additionally, we restricted our calculations to one hardware platform to avoid double-counting hardware support routines.

The core TinyOS source consists of 172 components, of which 108 are code modules and 64 are configurations. The number of modules per application in the TinyOS 1.x release ranges from 8 to 67, with an average of 24. Modules are generally quite small — on average only 120 lines of code. This small size indicates the expressive power of nesC’s components, as programmers have not needed to break the model by producing large monolithic components, which are more difficult to analyze and more error prone.

Figure 9 illustrates the effectiveness of the component model for our three sample applications. It shows the total number of modules and lines of code for each application, and the number of these modules and lines of code that come from OS components. We also include with the OS numbers the fraction of the OS each application ended up including. These numbers show that only a small portion of the OS is used by each application. nesC’s component model restricts the code included in each application to the bare minimum, hence reducing the application’s memory footprint.

Figure 10 shows the locations of data races in the TinyOS tree. Half of the races existed in system-level components used by many applications, while the other half were application specific. Modules MultihopM, eepromM, and TinyAlloc had a disproportionate number of races due to the amount of internal state they maintain through complex concurrent operations. IdentC tracks node interactions, records them in a hash, and periodically sends them to the basestation; it has complex concurrency, lots of state, and was written before most of the concurrency issues were well understood.

The finite-state-machine style of decomposition in TinyOS led to the most common form of bug, a non-atomic state transition. State transitions are typically implemented using a read-modify-write of the state variable, which must be atomic. A canonical example of this race is shown in Figure 12, along with the fix.

The original versions of the communication, TinyAlloc and EEPROM components contained large numbers of variable accesses in asynchronous code. Rather than using large atomic sections, which might decrease overall responsiveness, we promoted many of the offending functions to synchronous code, by posting a few additional tasks.

False positives fell into three major categories: state-based guards,
functions. This allows extensive cross-component optimizations, able code and module boundary crossings as well as to inline small nesC compiler uses the application call-graph to eliminate unreachable code and module boundary crossings for inline small nesC does not enforce this limitation.

### 4.3 Optimization

As with data-race detection, nesC exploits the restrictions of the component model to perform static analysis and optimization. The nesC compiler uses the application call-graph to eliminate unreachable code and module boundary crossings for inline small nesC does not enforce this limitation.

buffer swaps, and causal relationships. The first class, state-based guards, occurred when access to a module variable is serialized at run time by a state variable. The state transition example of Figure 12 illustrates this; in this function, the variable count is safe due to the monitor created by state.

TinyOS’s communication primitives use a buffer-swapping policy for memory management. When a network packet is received, the radio component passes a buffer to the application; the application returns a separate buffer to the component for the next receive event. This raises problems with alias analysis; although only one component has a reference to a given buffer at any time, components swap them back and forth. Although alias analysis would conclude that both components could concurrently read and write the same buffer, swapping ensures that only one component has a reference to it at any time. To resolve this issue, we annotated message buffers with the norace qualifier.

The last class of false positives, causal relationships, comes from the split-phase operation of TinyOS components. The command and event pair of a split-phase operation might share a variable, but their causal relationship means that the event will not fire while the command is executing. The event could fire during the command only if another TinyOS component accessed the underlying hardware; although this would violate the TinyOS programming model, nesC does not enforce this limitation.

These last two classes of false positives are actually high risk, and therefore worth verifying by hand. The absence of data races depends on strict use of the two idioms: it is easy to access message buffers concurrently by accident, and it is also possible to have a given split-phase operation running concurrently with itself (breaking the assumption that the two phases do not overlap).

We hope that we will be able to reduce the number of false positives by extending nesC with ideas from the static checking of software protocols in Vault [6].

Figure 12: Fixing a race condition in a state transition.

Figure 13: Effect of inlining on application footprint and performance. All sizes are in bytes. The CPU reduction column shows the reduction in CPU cycles spent executing tasks with optimizations enabled.

<table>
<thead>
<tr>
<th>App</th>
<th>Code size</th>
<th>Code reduction</th>
<th>Data size</th>
<th>CPU reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Surge</td>
<td>14794</td>
<td>16984</td>
<td>12%</td>
<td>1188</td>
</tr>
<tr>
<td>Maté</td>
<td>25040</td>
<td>27458</td>
<td>9%</td>
<td>1710</td>
</tr>
<tr>
<td>TinyDB</td>
<td>64910</td>
<td>71724</td>
<td>10%</td>
<td>2894</td>
</tr>
</tbody>
</table>


Including constant propagation and common subexpression elimination, to be performed by the back-end C compiler. These optimizations greatly reduce memory footprint and execution overheads, which are key savings for embedded systems. We evaluate the effectiveness of our inlining by first giving overall reductions in code size and CPU usage for our three representative applications, including a detailed component-by-component code-size breakdown for Surge. Then we show, by analysis of a single but important code path, that our inlining is effective at reducing the overhead of the nesC component system.

Figure 14 shows for each of our three applications the code size (in bytes) with and without these optimizations, the data sizes, and the decrease in execution cycles spent in tasks obtained by inlining. Because the networking stack dominates execution time in our applications, we show the reduction in CPU cycles for time spent executing tasks, nearly all of which are application-level code. nesC’s optimizations lead to a 15-34% reduction in execution time, and a 9-15% reduction in overall footprint.

Figure 14 gives a breakdown of the CPU cycle reduction for handling a timer event, which requires crossing 7 module boundaries. nesC’s optimizations save not only 57% of the boundary crossing overhead, but also 29% of the work required to handle the timer event, for a total savings of 38%. The increase in boundary crossing overhead for the interrupt occurs because the inlining requires the handler to save more registers; however, the total time spent in the handler goes down.

Figure 15 provides a detailed breakdown of the code and data size for each component in the Surge application. The TinyOS row represents the core TinyOS initialization code and task scheduler, which fits into 400 bytes. C Runtime represents necessary runtime routines, including floating-point libraries (currently used by multihop routing). Note that inlining increases code size slightly for only two components – TimerM and AbsoluteTimerM – but reduces or eliminates many others, leading to an overall reduction in footprint.

5. RELATED WORK

The module systems of languages such as Modula-2 (plus descendants) [12, 52, 53] and Ada [20] explicitly import and export interfaces. However these systems are less flexible than nesC, as there is no explicit binding of interfaces: an exported interface I is automatically linked to all importers of I. Standard ML's module system offers similar functionality to nesC, except that circular “component" (structure) assemblies cannot be expressed. The nesC module system is very close to Mesa’s [35], and (coincidently) uses essentially the same terminology: modules contain executable code, configurations connect components (configurations or modules) by binding their interfaces, whose interface types must match.

Giotto [13], Esterel [4], Lustre [11], Signal [3] and E-FRP [48] are languages that target embedded, hard real-time, control sys-
The nesC language is well-suited to the unique challenges of programmable embedded systems. There are no interface types (port connections match methods by name and signature), fan-out is limited to methods with no result, and dispatch of port methods is dynamic. C# [32] and BCOOPL [5] support events in classes and interfaces. However the two directions are asymmetric: registration of an event handler with an event producer is dynamic.

The languages above, and languages commonly used for programming embedded systems (such as C, Ada, Forth), do not offer the set of features desired in nesC: an interrupt-based concurrency model, low-level hardware access, component-based programming and static concurrency checking. A number of operating systems have explored the use of component architectures. The Flux OSKit [10] is a collection of components for building operating systems, but provided components target workstation-like machines. Flux was subsequently reimplemented with the help of Knit [43], a language for constructing and linking components implemented in C. Knit’s component model, based on that of units [9], is similar to nesC in that components provide and use interfaces and that new components can be assembled out of existing ones. Unlike nesC, Knit lacks bidirectional interfaces and data race detection. THINK [8] takes the Flux model one step further by allowing explicit modeling of calls between components. This allows clean linking of THINK components across protection domains or networks. THINK does not employ whole-program optimization and relies on dynamic dispatch. We do not believe this model, with its associated runtime cost, is appropriate for network embedded systems. Other component-oriented systems include Click [36], Scout [37], and the J-kernel [17]. These systems are more specialized than nesC and do not support whole-program optimization (apart from several optimizations in Click [24]) or bidirectional interfaces.

Traditional real-time and embedded operating systems, such as VxWorks [51], QNX [42], and WinCE [33], differ from TinyOS in a number of respects. These systems are generally much larger and provide greater functionality than TinyOS, and are intended for larger-scale embedded systems. We refer the reader to TinyOS [16] for a thorough discussion of the differences among these systems.

There have been a few attempts at static detection of race conditions. ESC [7] is in between a type checker and a program verifier; it has been used to verify user-supplied associations between locks and variables, and also to enforce ordering constraints on lock acquisition. Sun’s LockLint [46] statically checks for inconsistent use of locks or lock ordering. As expected, these tools have trouble with first-class functions and aliasing, and tend to report a subset of the errors with false positives as well. They also focus on locking rather than atomicity; we chose the latter to enable more freedom of implementation, which is particularly important for interrupts. The next section covers the use of monitors in Mesa.

There are also tools for dynamic detection of races. Eraser [44] detects unprotected shared variables using a modified binary. “On the fly” race detectors [31, 39] serialize all accesses to a variable to verify serializability. These approaches only catch errors that actually occur during the test run. In addition, dynamic approaches are less appealing for motes due to their number, resource limitations, and UI constraints. All of these race detection systems, including nesC, validate individual variable accesses. They cannot detect a read-modify-write through a temporary variable in which the read and write occur in distinct atomic sections.
Programming networked embedded systems. nesC was originally designed to express the concepts embodied in TinyOS, and by re-implementing the operating system in nesC, those concepts were refined. nesC’s component model supports holistic system design by making it easy to assemble applications that include the necessary OS support. The component model also allows alternate implementations and a flexible hardware/software boundary. nesC’s concurrency model allows us to write highly concurrent programs on a platform with very limited resources; the use of bidirectional interfaces and atomic statements permit a tight integration of concurrency and component-oriented design. Careful restrictions on the programming model, including the lack of dynamic allocation and explicit specification of an application’s call-graph, facilitate whole-program analyses and optimizations. Aggressive inlining reduces both memory footprint and CPU usage, and static data-race detection allows the developer to identify and fix concurrency bugs.

The nesC design opens up several key areas for future work. These broadly fall into the areas of concurrency support, enhancements to language features, and application to domains other than networked embedded systems.

Concurrency support

The nesC concurrency model provides short atomic actions, which can be used to build higher-level synchronization mechanisms such as semaphores, condition variables, atomic queues, and locks. Some of these mechanisms imply blocking, but there is nothing in the language per se that prevents support for blocking; we would need to prohibit blocking calls in atomic sections as well as treat blocking calls as yield points for task scheduling.

Our current implementation of atomic sections, which works well for embedded systems, is to disable interrupts. This is acceptable in part because we prevent blocking and limit the length of atomic sections. It also depends on the assumption of a uniprocessor and on the lack of virtual memory, since a page fault should not occur within an atomic section. However, these assumptions can be relaxed by considering alternate implementations of atomic sections, for example, using non-blocking synchronization primitives [15].

The use of monitors in Mesa [26] is the most directly comparable concurrency model to ours. In Mesa, the authors considered and rejected atomicity based on non-preemption for several reasons, including the desire to support multiprocessors and virtual memory. Also, non-preemption alone does not handle interrupts; we use atomic sections to handle asynchronous updates to shared state. Finally, Mesa was unable to prevent calls within atomic sections from yielding the processor. This is not an issue for nesC.

Language enhancements

There are a number of idiosyncrasies common in TinyOS that are not well expressed in nesC. Multi-client services with per-client state are not well-supported. For example, consider a general timer service where each client wishes to receive timer events at a different frequency. Abstract components can be used for this purpose, although they are currently limited in that the internal state of each instance is private to that instance. We currently use parameterized interfaces to implement such multi-client services, where the parameter corresponds to the “client number.” We are not wholly satisfied with this approach, and we plan to investigate a better mechanism in the future.

Split-phase operations provide high concurrency with low overhead, but are difficult to program; reintroducing the convenience of a threaded model would greatly simplify programming. By automatically transforming blocking operations into split-phase calls, we could preserve expressive lightweight concurrency without forcing the programmer to manually build continuations within components (as they do now). As it stands, many components are written as small finite state machines; atomic state transitions result in replicated control flow, separating state transitions from their corresponding actions. A future direction for nesC is to provide explicit support for FSM-style decomposition that simplifies component design and allows properties of FSM behavior to be statically verified.

Application to other platforms

We believe that nesC is not limited to the domain of embedded systems. nesC’s component-oriented structure, focus on concurrency, and bidirectional interfaces are valuable concepts in programming larger systems, such as enterprise-class applications and Internet services. To effectively support this broader class of applications, several extensions to nesC are needed. First, nesC’s compile-time analyses would need to be extended to handle dynamic memory and component allocation, as well as patterns such as message buffer swap. The static checking of software protocols in Vault [6] may provide an approach to solving these problems (and may also help in reducing false positives in data race detection). nesC’s concurrency model should be extended to admit multiprocessors, blocking operations, and a more general notion of threads, as discussed above. Such an approach would lead to a rich set of concurrency primitives specifically tailored for component-oriented programming of large-scale systems.

Acknowledgements

This work was supported, in part, by the Defense Department Advanced Research Projects Agency (grants F33615-01-C-1895 and N6601-99-2-8913), the National Science Foundation under Grant No. 0122599, California MICRO program, and Intel Corporation. Research infrastructure was provided by the National Science Foundation (grant EIA-9802069).

7. REFERENCES


