CHANNEL AND ACTIVE COMPONENT ABSTRACTIONS FOR WSN PROGRAMMING

A Language Model with Operating System Support

Paul Harvey[1], Alan Dearle[2], Jonathan Lewis[2] and Joseph Sventek[1]

[1]*School of Computing Science, University of Glasgow, Glasgow, Scotland, U.K.*

[2]*School of Computer Science, University of St Andrews, St Andrews, Scotland, U.K.*

Keywords: Wireless Sensor Network, Insense, InceOS, nesC, TinyOS, Operating System, Programming Language.

Abstract: To support the programming of Wireless Sensor Networks, a number of unconventional programming models have evolved, in particular the event-based model. These models are non-intuitive to programmers due to the introduction of unnecessary, non-intrinsic complexity. Component-based languages like Insense can eliminate much of this unnecessary complexity via the use of active components and synchronous channels. However, simply layering an Insense implementation over an existing event-based system, like TinyOS, while proving efficacy, is insufficiently space and time efficient for production use. The design and implementation of a new language-specific OS, InceOS, enables both space and time efficient programming of sensor networks using component-based languages like Insense.

1 INTRODUCTION

Wireless sensor networks enable a wide variety of activities to be performed autonomously, and are currently being used in many diverse areas including measurements of mountain permafrost (Hasler et al., 2008) and grapevines (Burrell et al., 2004). Such networks take highly constrained hardware devices (motes) and connect them via short-range radios to form useful monitoring tools, protection systems, and research systems. In programming such devices, a number of unconventional programming models have evolved. For example, both TinyOS (Hill et al., 2000) and Contiki (Dunkels et al., 2004) use an event-driven programming model, although realised through different abstractions. We assert that these models are non-intuitive to programmers due to the introduction of unnecessary non-intrinsic complexity. In particular, the introduction of the TinyOS split-phase execution model is a barrier to understanding, writing and reasoning about WSN programs. The same argument could also be levelled at programming with TinyOS threads and Contiki proto-threads. However, due to space limitations, we focus on the event-driven TinyOS abstractions in this paper.

In this paper we describe a new language-specific operating system, InceOS, which was created to support the domain-specific language (DSL) Insense (Dearle et al., 2008). Unlike the previous Insense implementation on Contiki, InceOS is tuned to the needs of Insense, enabling non-trivial Insense application to fit onto real mote hardware and execute in a timely manner. We hypothesise that the component-based model and abstractions presented by Insense, and implemented by InceOS, support a simpler programming model for resource-constrained embedded systems, removing much of the complexity of an event-based system, while still affording the user the power to express complex applications. We demonstrate that applications written using Insense, and running on InceOS both fit onto and run efficiently on real mote hardware with better performance when compared to TinyOS. This is shown via a comparison of Insense and nesC applications that cover different aspects of embedded programming.

The paper is structured as follows: Section 2 gives an overview of event-driven programming and TinyOS, Sections 3 and 4 discuss Insense and InceOS, Section 5 presents the results of the comparison between the languages and systems, Section 6 highlights directions for future work, and Section 7 provides concluding remarks.

Harvey P., Dearle A., Lewis J. and Sventek J..
CHANNEL AND ACTIVE COMPONENT ABSTRACTIONS FOR WSN PROGRAMMING - A Language Model with Operating System Support.
DOI: 10.5220/0003832500350044
In *Proceedings of the 1st International Conference on Sensor Networks* (SENSORNETS-2012), pages 35-44
ISBN: 978-989-8565-01-3
Copyright © 2012 SCITEPRESS (Science and Technology Publications, Lda.)

2 BACKGROUND

In order to place Insense and InceOS in context, we describe the event-driven model and how it relates to the computational model of TinyOS[1]. Event-driven systems respond to events. These events can be generated by the hardware, for example by interrupts, or by software. An event triggers an associated event handler, which handles the event and results in some computation being initiated that may, in turn, generate further events.

In event-driven systems, there is no single locus of execution; rather, there are a number of them each triggered by an event. Such systems have become popular for embedded systems since they do not require the same memory and processing overheads as threads (e.g., for stacks and context switching), yet provide a concurrent computational model. Another advantage is that concurrency control is simplified since, in many (single-CPU) systems, multiple event handlers do not run simultaneously (Hill et al., 2000).

2.1 TinyOS

TinyOS is a component-based, event-driven system in which events are signalled by both hardware and software, and event handlers (callbacks) are expressed as user-written functions that are invoked in response to events. To support this event model, TinyOS applications are composed of *components* that are wired together; a configuration is shown in Figure 1. Each component contains state, effectively executes autonomously, and presents and uses (procedural) *interfaces*. Interfaces are used to specify the events and commands that a component should support.

```
configuration SenseAppC  {}
implementation {
components SenseC, MainC, LedsC,
       new TimerMilliC(),
       new DemoSensorC() as Sensor;
SenseC.Boot -> MainC;
SenseC.Leds -> LedsC;
SenseC.Timer -> TimerMilliC;
SenseC.Read -> Sensor;
}
```

Figure 1: nesC SenseAppC configuration.

Each component contains *command handlers* (for handling down-calls) and *event handlers* (for handling up-calls). Commands are non-blocking requests made to lower level components; a command will usually deposit parameters into the component's local state and post a task for later execution; it may also invoke a lower level command. Event handlers are intended to be small blocks of code that either invoke other event handlers, initiate split phase operations, call commands, or post *tasks*. Tasks, unlike events, are schedulable, non-preemptable entities that must run to completion.

```
#include "Timer.h"

module SenseC
{
  uses {
    interface Boot;
    interface Leds;
    interface Timer<TMilli>;
    interface Read<uint16_t>;
  }
}
implementation
{
  // sampling frequency in binary milliseconds
  #define SAMPLING_FREQUENCY 100

  event void Boot.booted() {
  call Timer.startPeriodic(SAMPLING_FREQUENCY);
  }

  event void Timer.fired() {
    call Read.read();
  }

  event void Read.readDone(error_t result,
                           uint16_t data) {
    if (result == SUCCESS){
      if (data & 0x0004)
        call Leds.led2On();
      else
        call Leds.led2Off();
      if (data & 0x0002)
        call Leds.led1On();
      else
        call Leds.led1Off();
      if (data & 0x0001)
        call Leds.led0On();
      else
        call Leds.led0Off();
    }
  }
}
```

Figure 2: nesC SenseC component.

TinyOS does not perform blocking operations; instead it uses split-phase operations for long running executions. This methodology typically involves initiating operations via a `startOperation()` command, and completing an operation with an `operationDone()` event handler. TinyOS and its applications are written in the nesC language (Gay et al.,

[1]Other event-based [e.g. Contiki] and component-based [e.g. Lorien (Porter and Coulson, 2009)] systems exist; space limitations prevent their inclusion in the discussion.

2003), a component-based language that extends C. The code fragment shown in Figure 1 describes the (static) configuration of the Sense program (Hauer, 2006) which displays the battery status of a mote on its LEDs. The `configuration` describes both the components to be used and their interconnectivity.

The only user-supplied nesC component for this program is shown in Figure 2. Execution starts when an event is handled by `Boot.booted()`; it initiates a periodic timer by calling `Timer.startPeriodic()` before yielding control. At some later point in time, a timer event is signalled to `Timer.fired()`; it initiates a read of the battery sensor by calling `Read.read()` before yielding control again. Finally, when the Read component signals a `Read.readDone()` event, it is handled and the LEDs switched accordingly.

We assert that the split-phase programming model introduces unnecessary cognitive load on the readers and writers of such programs. Note that even this, relatively short, piece of code contains 2.5 split-phase operations, obfuscating the flow of control.

SCIENCE AND TECHNOLOGY PUBLICATIONS

3 INSENSE

Insense is a component-based language for WSN programming that was previously implemented over Contiki. The computational model presented by Insense is simple: applications are constructed as compositions of active *components* communicating over strongly typed, directional, synchronous *channels*. This model borrows from both the Actor (Hewitt et al., 1973) and π-calculus (Milner et al., 1992) language families. This computational model simplifies three areas of programming: *synchronisation*, *memory management*, and *event-driven programming*.

3.1 Components

Insense components exhibit *shared nothing* semantics: there are no global variables and a component's state may only be accessed from within the component. Insense components may be dynamically instantiated using one of the components' *constructors* in a manner familiar to Java or C++ programmers.

Each Insense component comprises three elements: a *behaviour*, one or more *constructors* and an arbitrary number of *interfaces*. It may also contain *component local state* and *functions*. A component's *behaviour* is akin to a void procedure that is repeatedly executed upon component instantiation, until explicitly terminated. Each behaviour runs as an individual pre-emptible locus of control (similar to a thread) and may access the component instance's state. Since there is only one locus of control per component which can never leave the component in which it was created, threads and component instances are in 1:1 correspondence. This, coupled with all component state being private, means that there is no need for explicit synchronisation or other concurrency control primitives in the language.

3.2 Interfaces

Components present interfaces which comprise a collection of typed, named, directional channels. Synchronous sending of messages via *channels* is the only inter-component communication mechanism provided in the language. Figure 3 below shows an Insense component definition that presents an interface containing three channels named: *ticker*, *battReq* and *battVal*. Two of these channels are input channels meaning that values can be read from them, one is an output channel to which data may be written. The specification of a channel in a component interface implicitly makes a declaration, introducing the channel name into the scope of the component body; thus the names *ticker*, *battReq* and *battVal* are available for use within the component.

3.3 Channels

Channels are typed and directional. In addition to being able to declare channels in component interfaces, they may also be dynamically declared. Channels are first class entities in Insense and may be passed down appropriately typed channels. This technique may be used to dynamically wire-up Insense programs. As stated above, *send* and *receive* are synchronous operations in Insense and implement a rendezvous in which either the sender or the receiver will block until the other is ready. The program in Figure 3 receives ticks from a *ticker* channel and makes requests for data from the battery component.

Channels are connected using explicit *connect* statements in the language. If a component attempts to communicate over an unbound channel it will block until a connection is established and data is available. This mechanism is extremely powerful and facilitates dynamic composition and evolution of components in a live system. In the program in Figure 3, a newly created instance of the *sense* component is explicitly connected to the (predefined) *sensors* component. A *disconnect* operator is also provided although not shown in this example. In addition to the send and receive operations shown here, Insense also provides a powerful guarded, non-deterministic select statement permitting data to be read from an arbitrary number

of channels. Unlike nesC, connect statements may be dynamic, permitting flexible, reconfigurable architectures. Insense channels may be used to wire up components in arbitrary topologies. Thus an output channel can be connected to multiple input channels or vice versa. Whilst this makes channel implementation more complex, it increases the expressive capabilities of the language and permits complex architectural patterns to be expressed. It also introduces a second source of non-determinism into the language.

3.4 Memory Management

Like Java, Insense provides the `new` keyword for dynamic memory allocation of arrays, structs, channels, and components; reference counted garbage collection is used to restore unused memory to the heap, thus removing the burden of memory management from the programmer. An upper bound on the (relatively-small) stack space required for each component instance is calculated at compile time. This is possible due to restrictions on recursive invocation of functions and a simple computational model.

3.5 Event-driven Programming

The channel mechanism provides an excellent vehicle for event-driven programming with events delivered via channels either from hardware devices or other components. If required to do so, components can be made to block awaiting delivery of events via a `receive` statement. Typically a single component will be connected to an event channel. However, multiple handlers can also be connected to a single event channel providing the capability to process events in parallel. This model is more expressive than those of TinyOS, is no less efficient, and simpler to understand.

3.6 SenseC in Insense

The code in Figure 3, shows a program equivalent to the TinyOS *SenseC* application encoded in Insense. The code defines and instantiates a single component type called *sense* which presents an interface *ISense* containing three channels as described above. The component repeats its behaviour clause forever, blocking every repetition until a tick is received from the *ticker* channel. Once a tick is received, the battery levels are requested by sending a message on the *battReq* channel before blocking awaiting a reply. The final four statements represent the "main" for the application. They create the component, bind its channels to the sensors, and instantiate the *ticker* to deliver timer messages every 0.098 seconds; the value of SAMPLING_FREQUENCY in Figure 2 corresponds to a timer event every 0.098 seconds due to TinyOSs use of 1024 ticks per second. This program is shorter, simpler and, easier to understand than the nesC equivalent.

```
type ISense is
  interface (in bool ticker; out bool battReq;
          in integer battVal)

component sense presents ISense {
    constructor() {}

    behaviour {
        receive tick from ticker
        send true on battReq
        receive reading from battVal

        if (reading & 4) == 4
            then setRedLedPower(true)
            else setRedLedPower(false)
        if (reading & 2) == 2
            then setBlueLedPower(true)
            else setBlueLedPower(false)
        if (reading & 1) == 1
            then setGreenLedPower(true)
            else setGreenLedPower(false)
    }
}

s =  new sense()
connect s.battReq to sensors.batteryRequest
connect s.battVal to sensors.batteryOut
setTimer(s.ticker , 0.098 , true)
```

Figure 3: Entire Insense SenseC program.

4 InceOS

InceOS is a new operating system that has been specifically created to execute applications written in Insense. The OS is pre-emptive and supports the concurrent execution of multiple Insense components. The language concepts of *components* and *channels* are directly implemented, ensuring that the language semantics are captured and enforced in the OS. InceOS itself is written in C, and exports an API which is targeted by the Insense compiler.

InceOS provides Insense applications with both system calls and system components. System calls are provided to manage the fundamental language abstractions: channels and components, and to perform appropriate checks such as array bounds checks. These system calls are used by the C code generated by the Insense compiler. The system calls mainly encompass the creation, destruction, and ma-

nipulation of components and channels; for example, `channel_send()` and `channel_receive()` abstract over the communication and concurrency control aspects of inter-component communication. There is no need for the run-time to ensure type safety of such operations as this is ensured by the compiler. These system calls provide a platform-neutral API for the Insense compiler, enabling the language to be ported to multiple sensor hardware platforms.

The OS also provides a number of system components which provide an abstraction over access to the platform specific hardware and services, for example, the battery sensor in Figure 3. These components are *well-known*, in that they are known to both the programmer and Insense compiler. An application interacts with these system components via channels. These system components include the timer, radio, buttons, and sensors.

4.1 Scheduling and Threads

Due to the close coupling of channels and components, the InceOS scheduler does not need to maintain references to components which are blocked and unschedulable. A blocked component can only become eligible to run through the rendezvous of channel actions; the most recent rendezvous action unblocks the component and places it onto the run queue of the scheduler. This is possible as each channel is aware of the components to which it is connected; thus when a channel action occurs, only the relevant components are examined. Consequently, the scheduler is simplified, as it has less state to maintain. TinyOS requires a programmer to explicitly post a task to the scheduler before it can be executed. While this is not necessarily an onerous requirement on the programmer in TinyOS, the removal of this responsibility by InceOS simplifies the process.

InceOS provides a round robin scheduler with a quantum of 30 ms and two priority levels: interrupt and normal. Normal priority is for any component, system or user, which becomes runnable as a result of a channel operation. Interrupt priority is only used for system components which are made runnable by hardware interrupts. For both priority levels, the same run queue is used; normal priority components are added at the tail of the run queue, and interrupt priority components are placed at the head. Should multiple components be scheduled at interrupt priority, they are added in chronological order. To simplify the interaction with the hardware, a system component simply waits on a channel in the same manner as the `sense` component in Figure 3 waits on a `tick` from the `ticker` channel. The following exemplifies this for the system-level Timer component which interfaces with both the timer hardware and user-level components. The hardware timer generates interrupts causing an interrupt handler to be invoked. The handler sends a message to the system-level Timer component on its interrupt channel. The Timer component repeatedly waits for messages from an interrupt channel prior to sending messages to those user-level components for which requested timers are due. At the end of its behaviour clause, the Timer component resets the hardware timer for the next outstanding user-level timer that is due to expire prior to waiting for the next interrupt to occur. In this way, the system components are able to interact with both user components and hardware via the channel mechanism.

As with TinyOS, InceOS enters a low power state when there is no more work to be done - i.e., the scheduler's run queue is empty. The system reawakens when a hardware interrupt causes a system component to become schedulable again. User components are awakened when messages are received on the channels on which they are blocked as described above.

Each user component in Insense is pre-emptible, ensuring a fair share of the CPU to all eligible user components. This requires that each component has an associated thread and stack. The main opposition to threads and stacks on motes is that they consume relatively large amounts of limited RAM. Insense and InceOS minimise this impact by determining the stack size required for each component's behaviour at compile time and dynamically allocating this at runtime, as described in section 3.4. For example, the compiler has calculated a stack size of 22 bytes for the component in Figure 3. This provides space for local variables, as well as variables required by the OS. A further 120 bytes are added to provide space for system actions and interrupts, giving a maximum stack depth of 142 bytes. The compiler also calculated a size of 10 bytes for the component's heap object. This is used to hold references to a component's channels and instance state, as well as the fields used for garbage collection and component manipulation. These values are passed to the OS when creating a component.

4.2 Radio Comms

Presently, InceOS has broadcast and unicast capabilities. It does not support tree based routing protocols, or IP, unlike TinyOS. However, these can be implemented within a programmer's application. At present, user components communicate with a *radio* component provided by the system. This component provides two input channels, *broadcast* and *unicast*,

and a single output channel, *received*. A user component will connect to either the *broadcast* channel to send broadcast messages, or the *unicast* channel to send unicast messages. A component wishing to receive either type of message connects to the *received* channel.

5 EVALUATION

In order to demonstrate that applications written using Insense both fit onto and run efficiently on real mote hardware, a number of comparisons were made between Insense and nesC, and InceOS and TinyOS. Several WSN applications were written in nesC and Insense and executed on their respective operating systems. All of the following results were obtained from Tmote Sky[2] motes which have 10 KB of RAM and 48 KB of flash available. The motes were tested under full power.

5.1 Applications

The chosen test applications are primarily from the TinyOS source tree, augmented with some that were explicitly constructed to highlight syntactic and computational performance differences. The source code for all of these applications is available online[3].

- *BlinkA* is a simple application that periodically blinks the three different LED's of the Tmote Sky at different rates. *BlinkB* performs the same operation using the TinyOS thread library (*TOSThreads* (Klues et al., 2009)).
- *TestSineSensor* periodically samples a sensor, after which it forwards the obtained value over the serial link. Under TinyOS, it is implemented using TOSThreads.
- *RadioStress* uses three threads to send messages to another mote where three threads are listening for messages from their counterparts. Under TinyOS, it is implemented using TOSThreads.
- *RadioCountToLeds* involves two motes, one maintains a counter which is transmitted over the radio to the other mote which displays the lower three bits of the transmitted value on its LED's.
- *RadioSenseToLeds* is a similar application, except that it collects and sends sensor data as opposed to a software counter.
- *Sense* is similar to RadioSenseToLeds, but it only uses one mote and does not send sensor values over the radio, as shown in Figures 2 and 3.
- *TestRoundRobinArbiter* is an example of an access control mechanism where three resource users request access from a central controller, which grants access to each in turn.
- *Fourier* performs a Fourier transform on an array of 40 integers repeatedly.
- *Grid* is based on the notion of using ad-hoc grids in sensor networks to mitigate the power consumed by excessive radio transmission of data (Rondini and Hailes, 2007). This application consists of two types of node: leaders, who make requests, and slaves, who service requests. Initially the leader broadcasts a request asking for any free slave. Once a slave replies to this request, the leader collects enough sensor data to fill an array of size 10 and transmits it to the slave that acknowledged it. The slave performs a Fourier transform on the received data, calculates the maximum value and returns this to the leader.

The Grid and Fourier applications were not provided by TinyOS. The combination of these applications cover intense computation, radio transmission, interactions between components on a single node and combinations thereof.

5.2 Concurrency and Serialisation

As each component within Insense executes an isolated, independent thread, there is no need for the developer to use any special thread libraries, functions or routines while writing an application, in contrast to TinyOS. Instead they can simply concentrate on writing the code that embodies the activity of a component within its behaviour. Additionally, as InceOS is pre-emptive, it will ensure that all user components that are eligible to be run have an equal opportunity to do so. This ensures that no one component monopolises the CPU, and does not require the programmer to explicitly yield control.

A context switch in InceOS takes 15.3μs (380 assembly instructions). Context switching is often cited as a drawback of using threads in embedded systems, however these values show that the time required to go from executing in one component to another is small and bounded.

One of the most common problems associated with concurrency is the incorrect use of locking mechanisms to guarantee serialised access to shared variables, leading to race conditions and deadlock. Insense removes race conditions by making the instance

[2]http://www.sentilla.com/moteiv-transition.html, Accessed on 25/09/2011.

[3]http://blogs.cs.st-andrews.ac.uk/insense/insense-on-inceos-examples/

Table 1: Comparison of Insense and nesC code.

	Lines of Code		Components		Wiring Statements		Interfaces	
Application	Insense	nesC	Insense	nesC	Insense	nesC	Insense	nesC
BlinkA	28	40	2	6	3	5	1	5
BlinkB	30	54	4	7	1	6	1	6
TestSineSensor	13	45	2	7	2	8	1	7
RadioStress	50	94	5	13	9	12	1	12
RadioCountToLeds	55	103	5	7	4	7	2	7
RadioSenseToLeds	57	101	6	8	6	8	2	8
Sense	29	43	3	5	3	4	1	4
TestRoundRobinArbiter	49	180	5	11	10	15	2	24
Fourier	19	30	1	2	0	1	1	1
Grid	97	177	5	8	7	8	1	8

state of components private as part of the language. When data must be shared between components, it is done via the channel mechanism. The channel mechanism must be efficient for production use. In InceOS, typical transmission time on a channel when both components are ready is $215 \pm 0.08 \mu s$.

5.3 Code Composition

Both Insense and nesC applications are composed of components, interfaces, and wiring statements. Accordingly, these features were used as metrics to judge code complexity. Table 1 shows the results. The table shows the application, number of lines of code used, number of components either written or referenced, number of wiring/connect statements, and the number of interfaces used. Obviously each of these features could be manipulated -e.g., every Insense application could be written in a single component. To prevent this, each Insense application uses a component representing each activity of the program and the system components.

The table shows that the applications can be written in Insense with fewer elements from each category, excluding the Fourier application where both languages require one interface. Although it does not necessarily follow that fewer is better, the previous discussion of the simpler composition of Insense and these results show that it is possible to write functionally-equivalent programs in Insense with many fewer lines of code.

5.4 Size

Figure 4 shows the amount of flash which is consumed on the Tmote Sky by TinyOS and InceOS when compiled with an application. Figure 5 shows the amount of space consumed by the data and bss sections of the compiled InceOS and application, and Figure 6 shows the equivalent for TinyOS. It can be seen that there is variation between the applications on TinyOS compared to the relatively static figures for InceOS. InceOS consumes more flash than TinyOS partially due to the optimisations of the nesC compiler, but mostly due to the extra support mechanisms found in InceOS. These include the runtime features discussed previously.

InceOS dynamically allocates the structures used to represent channels and components at runtime. Dynamic allocation leads to the small values seen in Figure 5, and the lack of conditional compilation causes them to be uniform across the different applications. The use of dynamic allocation and stacks exacts a runtime cost in RAM. The cost for Insense and InceOS is highlighted via two examples. The first uses a (null) component with no channels and no code in the behaviour section. This null component requires 188 bytes. The 188 bytes is the size required to represent the component, its channels, its stack, and any dynamically allocated structures. After the system and null components are initialised, with both components and channels being allocated, there are 4363 bytes of RAM available. The second example uses the component from Figure 3. Here the *sense* component requires 364 bytes, leaving 4187 bytes of RAM available. Taking the sense component as an example of an average component, there is enough space on the Tmote Sky to create 11 such components. As discussed in section 4.1, 120 bytes are added to the compiler computed stack size for a component to accommodate system calls and interrupts. We intend to introduce a kernel stack for all system operations, thus reducing the 120 byte overhead to 26 bytes, the maximum stack space required for an interrupt. For the sense example above, this would permit the creation of 17 Sense component instances, each consuming 272 bytes of RAM.

InceOS consumes more flash and RAM than TinyOS, however there is still adequate space available on the motes for even larger and more complex applications; the largest and most complex application in this evaluation, grid, leaves just under 24 KB

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

of flash and nearly 3 KB of RAM available, 50% and 31% of the total space available, respectively.

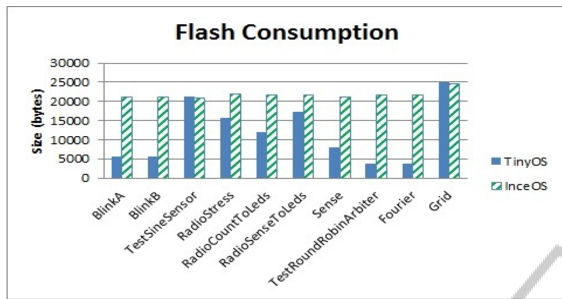

Figure 4: Flash consumed by applications and specified OS.

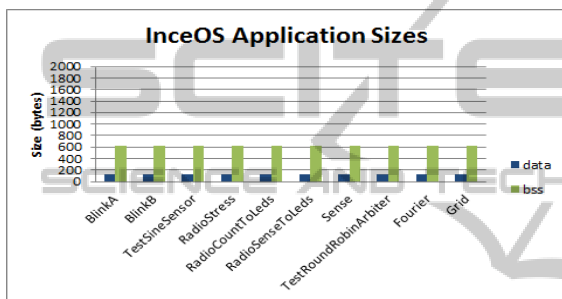

Figure 5: Space consumed by InceOS applications.

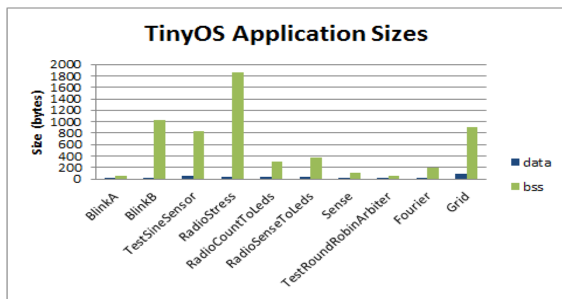

Figure 6: Space consumed by TinyOS applications.

5.5 Performance

To ascertain if support for the channel and threading mechanisms exact some cost in performance, we measured the performance of a representative cross section of the applications on both TinyOS and InceOS. In the following graphs, each data point is the average of 100 iterations of the application. For example, each point in Figure 7 is the average time required to increment a software counter and broadcast this value in a packet over the radio 100 times. The error bars on each point represent the standard deviation, however most are not visible as the results are often consistent within the measurement accuracy. Both TinyOS and InceOS are using a csma/ca protocol for radio transmission - i.e., before attempting to send, the radio hardware is queried to detect the presence of other radio transmissions; if radio signals are detected, the transmission is delayed, otherwise the packets are sent.

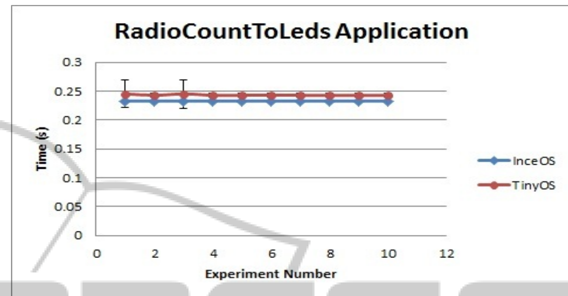

Figure 7: Comparison of RadioCountToLeds application.

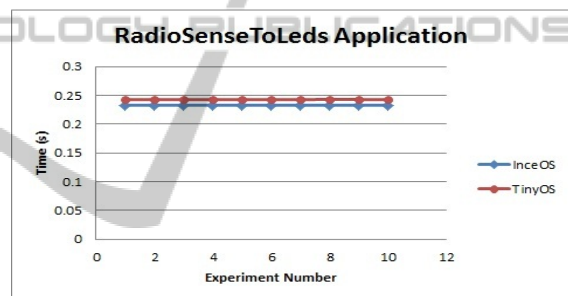

Figure 8: Comparison of RadioSenseToLeds application.

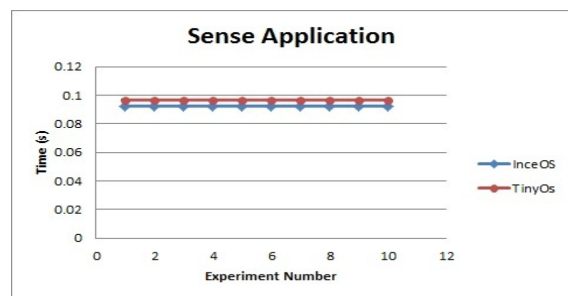

Figure 9: Comparison of the Sense application.

In both Figures 7 and 8, the measurements reflect the sender's action of collecting the data to be sent and sending it. Both figures show a similar performance increase of approximately 4 ms for InceOS as compared to TinyOS. This can be attributed to the fact that the Insense behaviour clause is repeatedly executed, rather than in TinyOS where events must be generated before the application can continue to

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

its next iteration. Figure 9 shows the comparison of the Sense application from Figures 2 and 3. We can see that InceOS performs a further 2 ms better than it did in Figures 7 and 8. This is because unlike RadioCount/SenseToLeds, the InceOS Radio component is not being used (or being scheduled) thus giving more time for the Sense component to execute.

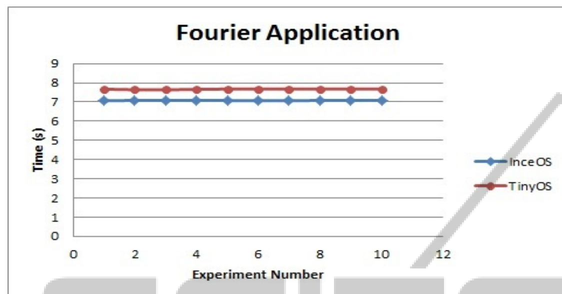

Figure 10: Comparison of the Fourier application.

Figure 10 shows an example of intense computation; the measurements reflect the time required to complete a fast Fourier transform on a forty element array and then calculate the maximum. Again InceOS outperforms TinyOS. This is due to the use of tasks in TinyOS to process the Fourier computation. When the task is finished, it must be reposted, requiring an invocation of the scheduler, whereas the behaviour clause in Insense is naturally repeated, not requiring any intervention by the scheduler, or extra code from the developer. This highlights that a purely event-driven model is not well-suited to straight computation (Dunkels et al., 2004).

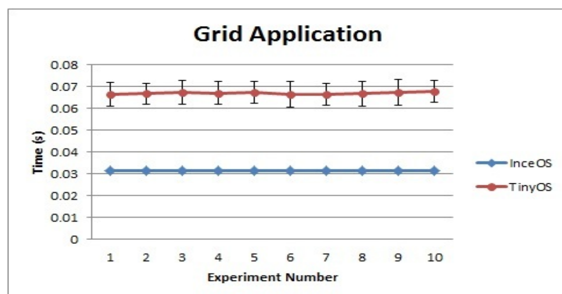

Figure 11: Comparison of the Grid application.

The results for the grid application are shown in Figure 11. Here the time for a complete iteration of the application is taken: request a slave, give it work and collect the reply. We see that InceOS performs substantially better than TinyOS. The 35 ms difference is caused by a simpler flow through the logic of the Insense application and the relatively small number of actions required to access the sensor and radio components. This is in contrast to the disjoint flow necessitated by control switching between the event handlers of the application in nesC, as well as the posting of a task to compute the Fourier transform.

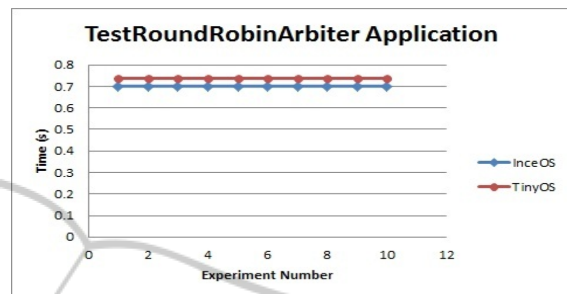

Figure 12: Comparison of the TestRoundRobinArbiter application.

The TestRoundRobinArbiter application results are displayed in Figure 12, again showing an InceOS performance gain when compared to TinyOS. Here the 36 ms gain is because the Insense channel abstraction is used in InceOS to arbitrate access to the shared resource, whereas additional functionality is required for such arbitration using TinyOS. This particular application is well-suited to the Insense blocking channel interaction which naturally handles arbitration.

5.6 Summary

The initial hypothesis was that sensor network applications written using a programming language with more appropriate abstractions, such as active components communicating over synchronous channels (as provided by Insense/InceOS), fit onto and run efficiently on real mote hardware. By comparing Insense implementations against equivalent nesC implementations on the same hardware platform, we have demonstrated the following:

- It is possible to write functionally-equivalent programs in Insense with many fewer lines of code;
- InceOS applications consistently outperform TinyOS for all applications in our test suite;
- Insense/InceOS systems consume more RAM and Flash memory to provide this performance, but a substantial fraction of each resource still remains for use by applications.

6 FUTURE WORK

Given the strong encapsulation exhibited by Insense components, as well as the low coupling between them, an obvious next step is to include dynamic reprogramming via the radio in Insense and an implementation in InceOS. This will enable dynamic, over-the-air composition of components at runtime.

Currently the radio interaction is exposed to the programmer and is the only means of inter-node communication. Although the actual hardware interaction is abstracted, the programmer is aware of the radio transfer via the Radio component. Future work will include the ability to introspect and react to the nodes network environment, thus making it possible to discover a node's neighbours and be alerted to changes.

New language-based systems can only become prevalent if sufficient program development, testing, verification, deployment, and runtime monitoring capabilities are also provided. We are building such a development environment for Insense/InceOS-based systems.

SCIENCE AND TECHNOLOGY PUBLICATIONS

7 CONCLUSIONS

The language-specific operating system, InceOS, enables programs written in Insense to exhibit the space and time efficiency needed for production use in sensor networks. A comparison of Insense/InceOS and nesC/TinyOS code for a range of applications shows that it is possible to write functionally-equivalent programs in fewer lines of Insense and that InceOS applications consistently outperform their TinyOS counterparts. Thus, use of the component-based abstractions provided by Insense coupled with the efficient support for these abstractions in InceOS facilitates the development of WSN applications that exhibit state-of-the-art performance while reducing programming complexity.

More generally, we conclude that provision of a language-specific operating system is an effective mechanism for making programs written in higher-level languages competitive with equivalent programs written in lower-level languages supported by more general-purpose operating systems.

REFERENCES

Burrell, J., Brooke, T., and Beckwith, R. (2004). Vineyard computing: Sensor networks in agricultural production. *IEEE Pervasive Computing*, 3:38–45.

Dearle, A., Balasubramaniam, D., Lewis, J., and Morrison, R. (2008). A component-based model and language for wireless sensor network applications. In *COMPSAC '08: Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference*, pages 1303–1308, Washington, DC, USA. IEEE Computer Society.

Dunkels, A., Gronvall, B., and Voigt, T. (2004). Contiki - a lightweight and flexible operating system for tiny networked sensors. In *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, Washington, DC, USA. IEEE Computer Society.

Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. (2003). The *nesC* language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, volume 38, pages 1–11, New York, NY, USA. ACM.

Hasler, A., Talzi, I., Tschudin, C., and Gruber, S. (2008). Wireless sensor networks in permafrost research - concept, requirements, implementation and challenges. In *Proc. 9th Intl Conf. on Permafrost (NICOP 2008*.

Hauer, J. (2006). nesc sense application repository. Web Site. http://code.google.com/p/tinyos-main/source/browse/trunk/apps/Sense/?r=2898 , Accessed on 25/09/2011.

Hewitt, C., Bishop, P., and Steiger, R. (1973). A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. (2000). System architecture directions for networked sensors. *SIGOPS Oper. Syst. Rev.*, 34(5):93–104.

Klues, K., Liang, C.-J. M., Paek, J., Musăloiu-E, R., Levis, P., Terzis, A., and Govindan, R. (2009). Tosthreads: thread-safe and non-invasive preemption in tinyos. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 127–140, New York, NY, USA. ACM.

Milner, R., Parrow, J., and Walker, D. (1992). A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40.

Porter, B. and Coulson, G. (2009). Lorien: a pure dynamic component-based operating system for wireless sensor networks. In *MidSens '09: Proceedings of the 4th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks*, pages 7–12, New York, NY, USA. ACM.

Rondini, E. and Hailes, S. (2007). Distributed computation in wireless ad hoc grids with bandwidth control. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, SenSys '07, pages 437–438, New York, NY, USA. ACM.