Reliable and Efficient Programming Abstractions for Wireless Sensor Networks

Nupur Kothari* Ramakrishna Gummadi *
University of Southern California
{nkothari, gummadi}@usc.edu

Todd Millstein
University of California, Los Angeles
todd@cs.ucla.edu

Ramesh Govindan
University of Southern California
ramesh@usc.edu

Abstract

It is currently difficult to build practical and reliable programming systems out of distributed and resource-constrained sensor devices. The state of the art in today's sensornet programming is centered around a component-based language called nesC. nesC is a *node-level* language—a program is written for an individual node in the network—and nesC programs use the services of an operating system called TinyOS. We are pursuing an approach to programming sensor networks that significantly raises the level of abstraction over this practice. The critical change is one of perspective: rather than writing programs from the point of view of an individual node, programmers implement a *central* program that conceptually has access to the entire network. This approach pushes to the compiler the task of producing node-level programs that implement the desired behavior.

We present the Pleiades programming language, its compiler, and its runtime. The Pleiades language extends the C language with constructs that allow programmers to name and access node-local state within the network and to specify simple forms of concurrent execution. The compiler and runtime system cooperate to implement Pleiades programs efficiently and reliably. First, the compiler employs a novel program analysis to translate Pleiades programs into message-efficient units of work implemented in nesC. The Pleiades runtime system orchestrates execution of these units, using TinyOS services, across a network of sensor nodes. Second, the compiler and runtime system employ novel locking, deadlock detection, and deadlock recovery algorithms that guarantee serializability in the face of concurrent execution. We illustrate the readability, reliability and efficiency benefits of the Pleiades language through detailed experiments, and demonstrate that the Pleiades implementation of a realistic application performs similar to a hand-coded nesC version that contains more than ten times as much code.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Distributed programming; D.3.2 [*Programming Languages*]: Language Classification—Specialized application languages

General Terms Performance, Design, Languages, Reliability, Experimentation

Keywords Wireless Sensor Networks, Macroprogramming, Energy Efficiency, Serializability, Deadlocks

1. Introduction

Wireless sensor networks consist of a system of distributed sensors embedded in the physical world. They are increasingly used in scientific and commercial applications [15, 28]. However, constructing practical and reliable wirelessly-networked systems out of them is still a significant challenge. This is because the programmer must cope with severe resource, bandwidth, and power constraints on the sensor nodes as well as the challenges of distributed systems, such as the need to maintain consistency and synchronization among numerous, asynchronous loosely coupled nodes.

Current practice in sensor network programming uses a a highly concurrent dialect of C called nesC [5], which is a *node-level* language — a nesC program is written for an individual node in the network. nesC statically detects potential race conditions and optimizes hardware resources using whole-program analysis. nesC programs use the services of the TinyOS operating system [11], which provides basic runtime support for statically linked programs. TinyOS exposes an event-driven execution and scheduling model and provides a library of reusable low-level components that encapsulate widely used functionality, such as timers and radios. TinyOS was designed for efficient execution on low-power, limited-memory sensor nodes called *motes*.

nesC and TinyOS provide abstractions and libraries that simplify node-level sensor-network application programming, but ensuring the efficiency and reliability of sensor network applications is still tedious and error prone (Section 2). For example, the programmer must manually decompose a high-level distributed algorithm into programs for each individual sensor node, must ensure that these programs efficiently communicate with one another, must implement any necessary data consistency and control-flow synchronization protocols among these node-level programs, and must explicitly manage resources at each node.

We are pursuing an alternative approach to programming sensor networks that significantly raises the level of abstraction over current practice. The critical change is one of perspective: rather than writing programs from the point of view of an individual node in the network, programmers implement a *central* program that conceptually has access to the entire network. This change allows a programmer to focus attention on the higher-level algorithmics of an application, and the compiler automatically generates the node-level programs that properly and efficiently implement the application on the network. In the literature, this style of programming sensor networks is known as *macroprogramming* [29].

---

* Primary authors.
`http://kairos.usc.edu`

This material is based in part upon work supported by the National Science Foundation under Grant Nos. 0520299, 0121778, 0427202 and 0545850. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

PLDI'07 June 11–13, 2007, San Diego, California, USA.

We have instantiated our macroprogramming approach in the context of a modest extension to C called Pleiades, which augments C with constructs for addressing the nodes in a network and accessing local state from individual nodes. These features allow programmers to naturally express the global intent of their sensor-network programs without worrying about the low-level details of inter-node communication and node-level resource management. By default, a Pleiades program is defined to have a sequential thread of control, which provides a simple semantics for programmers to understand and reason about. However, Pleiades includes a novel language construct for parallel iteration called `cfor`, which can be used, for example, to iterate concurrently over all the nodes in the network or all one-hop neighbors of a particular node.

The Pleiades compiler translates Pleiades programs into node-level nesC programs that can be directly linked with standard TinyOS components and the Pleiades runtime system and executed over a network of sensor motes. The key technical challenge for Pleiades is the need to automatically implement high-level centralized programs in an efficient and reliable manner on the nodes in the network. The Pleiades compiler and runtime system cooperate to meet this challenge in a practical manner (Section 3). This paper makes the following contributions:

- **Automatic program partitioning and migration for minimizing energy consumption.** Energy efficiency is of primary concern for sensor nodes because they are typically battery-powered. Wireless communication consumes significant battery energy, and so it is critical to minimize communication costs among nodes. Pleiades uses a novel combination of static and dynamic information in order to determine at which node to execute each statement of a Pleiades program. A compile-time analysis first partitions a program's statements into *nodecuts*, each representing a unit of work to be executed on a single node. The runtime system then uses knowledge of the actual nodes involved in a nodecut's computation to determine at which node it should be executed in order to minimize the communication overhead.
- **An easy-to-use and reliable concurrency primitive.** Concurrent execution is a natural component of sensor network applications, since each sensor node can execute code in parallel. However, with concurrency comes the potential for subtle errors in synchronization that can affect application reliability. To support concurrency while ensuring reliability, the Pleiades runtime system guarantees *serializability* for each `cfor`: the effect of a `cfor` loop always corresponds to some sequential execution of the loop. To achieve this semantics, the runtime system automatically synchronizes access to variables among `cfor` iterations via locks, alleviating the programmer of this burden. Locking has the potential to cause deadlocks, so the compiler and runtime system also support a novel distributed deadlock detection and recovery algorithm for `cfor`s.
- **A mote-based implementation and its evaluation.** We have implemented Pleiades on the widely used, but highly memory-constrained, mote platform. The motes we use have 10kB RAM for program variables and 48kB ROM for compiled code. Our implementation generates event-driven node-level nesC code that is conceptually similar to what a programmer would manually write today. We evaluate three applications belonging to three different classes (Section 4). We first compare the performance of a sophisticated pursuit-evasion game macroprogram with that of a hand-coded nesC version written by others [7]. We find that the Pleiades program is significantly more compact (the source code size less than 10% as large), well-structured, and easy to understand. At the same time, the Pleiades implementation has comparable performance with the native nesC implementation. We then evaluate a car parking application that requires a strict notion of consistency and show that the Pleiades implementation of the concurrent execution is reliable. We finally demonstrate the utility of control flow migration within a simple network information gathering example.

Researchers have previously explored abstractions for programming sensor networks in the aggregate [8, 25, 29], as well as intermediate program representations to support compilation of such programs [23]. However, to our knowledge, a self-contained macroprogramming system for motes—one that generates the complete code necessary for stand-alone execution—has not previously been explored or reported on. Pleiades is also related to research on parallel and distributed systems. Unlike traditional parallel systems and research on automatic parallelization, we are primarily interested in achieving high task-level parallelism rather than data parallelism, given the loosely coupled and asynchronous nature of sensor networks. Further, we target concurrency support toward minimizing energy consumption rather than latency, since sensor networks are primarily power constrained. Unlike traditional distributed systems, Pleiades features a centralized programming model and pushes the burden of concurrency control and synchronization to the compiler and runtime. A more detailed comparison with related work is presented in Section 5.

2. The Pleiades Language

2.1 Design Rationale

Pleiades is designed to provide a simple programming model that addresses the challenges and requirements of sensor network programming. Pleiades' sequential semantics makes programs easy to understand and is natural when programming sensor networks in a centralized fashion. Concurrency is introduced in a simple manner appropriate to the domain, via the `cfor` construct for node iteration. At the same time, the sequential semantics is still appropriate for the purpose of programmer understanding, because Pleiades ensures serializability of `cfor`s. This strong form of consistency and reliability is important for a growing class of sensor network applications, like car parking and the part of an application responsible for building a routing tree across the nodes. For these kinds of applications, we argue that Pleiades's sequential semantics is the right one. We have also used Pleiades for applications such as routing, localization, time synchronization and data collection, which require consistency for at least some program variables. To our knowledge, no other macroprogramming system guarantees even weak forms of consistency.

While Pleiades provides a sequential semantics, it nonetheless efficiently and naturally supports event-driven execution. Pleiades has special language support for sensors and timers that provides a synchronous abstraction for event-driven execution. The synchronous semantics is easy for programmers to understand and fits well with the sequential nature of a Pleiades program. Under the covers, the language constructs are compiled to efficient event-driven nesC code.

2.2 Parking Cars with Pleiades

We illustrate the language features of Pleiades and the benefits they provide over node-level nesC programs through a small but realistic example application. It involves low cost wireless sensors that are deployed on streets in a city to help drivers find a free space. (According to recent surveys [27], searching for a free parking spot already accounts for up to 45% of vehicular traffic in some metropolitan areas.) Each space on the street has an associated sensor node that maintains the space's status (free or occupied). The goal is to identify a sensor node with a free spot that is as close to the desired destination of the driver as possible. For ease of explanation, we define distance by hop count in the network, but

```
1: #include "pleiades.h"
2: boolean nodelocal isfree=TRUE;
3: nodeset nodelocal neighbors;
4: node nodelocal neighborIter;

5: void reserve(pos dst) {
6:  boolean reserved=FALSE;
7:  node nodeIter,reservedNode=NULL;
8:  node n=closest_node(dst);
9:  nodeset loose nToExamine=add_node(n, empty_nodeset());
10: nodeset loose nExamined=empty_nodeset();


11: if(isfree@n) {
12:  reserved=TRUE; reservedNode=n;
13:  isfree@n=FALSE;
14:  return;
15: }

16: while(!reserved && !empty(nToExamine)){
17:  cfor(nodeIter=get_first(nToExamine);nodeIter!=NULL;
          nodeIter = get_next(nToExamine)){
18:   neighbors@nodeIter=get_neighbors(nodeIter);
19:   for(neighborIter@nodeIter=get_first(neighbors@nodeIter);
          neighborIter@nodeIter!=NULL;
          neighborIter@nodeIter=get_next(neighbors@nodeIter)){
20:    if(!member(neighborIter@nodeIter,nExamined))
21:     add_node(neighborIter@nodeIter,nToExamine);
22:   }
23:   if(isfree@nodeIter){
24:    if(!reserved){
25:     reserved=TRUE; reservedNode=nodeIter;
26:     isfree@nodeIter=FALSE;
27:     break;
28:    }
29:   }
30:   remove_node(nodeIter,nToExamine);
31:   add_node(nodeIter,nExamined);
32:  }
33: }
34:}
```

Figure 1. A street-parking application in Pleiades.

it is straightforward to base this on physical distance. We consider an implementation of this application in Pleiades as well as two node-level versions written in nesC [5]. We show that the Pleiades version is simultaneously readable, reliable, and efficient. Each of the two nesC versions is more complex and provides reliability or efficiency, but not both simultaneously.

Figure 1 shows the key procedure that makes up a version of the street-parking application written in Pleiades. When a car arrives near the deployed area, a space near the driver's indicated destination is found and reserved for it by invoking `reserve`, passing the car's desired location. The `reserve` procedure finds the closest sensor node to the desired destination and checks if its space is free. If so, the space is reserved for the car. If not, the node's neighbors are recursively and concurrently checked.

The code in Figure 1 makes critical use of Pleiades's *centralized* view of a sensor network. We describe the associated language constructs in turn.

Node Naming. Pleiades provides a set of language constructs that allow programmers to easily access nodes and node-local state in a high-level, centralized, and topology-independent manner. The `node` type provides an abstraction of a single network node, and the `nodeset` type provides an iterator abstraction for an unordered collection of nodes. For example, variable `n` (line 8) in `reserve` holds the node that is closest to the desired position (the code for the `closest_node` function is not shown), and `nToExamine` (line 9) maintains the set of nodes that should be checked to see if the associated space is free.

The set of currently available nodes in the network is returned by invoking `get_network_nodes()`, which returns a `nodeset`. Pleiades also provides a `get_neighbors(n)` procedure that returns a `nodeset` containing `n`'s current one-hop radio neighbors. In Figure 1, the `reserve` procedure uses `get_neighbors` (line 18) to add an examined node's neighbors to the `nToExamine` set. The Pleiades runtime implements `get_neighbors` by maintaining a set of sensor nodes that are reachable through wireless broadcast.

Node-Local Variables. Pleiades extends standard C variable naming to address node-local state. This facility allows programmers to naturally express distributed computations and eliminates the need for programmers to manually implement inter-node data access and communication. Node-local variables are declared as ordinary C variables but include the attribute `nodelocal`, as shown for the `isfree` variable (line 2) in Figure 1. The attribute indicates that there is one version of the variable per node in the network.

A node-local variable is addressed inside a Pleiades program using a new expression `var@e`, where `var` is a `nodelocal` variable and `e` is an expression of type `node`. For example, the `reserve` procedure uses this syntax to check if each node in `nToExamine` is free (line 23). An expression of the form `var@e` can appear anywhere that a C l-value can appear; in particular, a node-local variable can be updated through assignment.

All variables not annotated as `nodelocal` are treated as ordinary C variables, whose scope and lifetime respect C's standard semantics. In Pleiades, we call these *central* variables, to distinguish them from node-local variables. In our example code, `reserved` is a central variable (line 6), which is therefore shared across all nodes in the network.

Concurrency. By default, a Pleiades program has a sequential execution semantics. However, Pleiades also provides a simple form of programmer-directed concurrency. The `cfor` loop is like an ordinary `for` loop but allows for concurrent execution of the loop's iterations. A `cfor` loop can iterate over any `nodeset`, and the loop body will be executed concurrently for each node in the set. For example, the `reserve` procedure in Figure 1 concurrently iterates over the nodes in `nToExamine` (line 17), in order to check if any of these nodes is free.

While concurrency is often essential to achieve good performance, it can cause subtle errors that are difficult to understand and debug. For example, a purely concurrent semantics of the `cfor` in `reserve` can easily cause multiple free nodes to read a value of false for the `reserved` flag. This will have the effect of making each such node believe that it has been selected for the new car and is therefore no longer free. To help programmers obtain the benefits of concurrency while maintaining reliability, the Pleiades compiler and runtime system ensure that the execution of a `cfor` is always *serializable*: the effect of a `cfor` always corresponds to some sequential execution of the loop. In `reserve`, serializability ensures that only one free node will reserve itself for the new car; the other free nodes will see the updated value of the `reserved` flag at that point. Section 3.2 explains our algorithm for ensuring serializability for `cfor` loops.

Pleiades allows `cfor`s to be arbitrarily nested. The serializability semantics of a single `cfor` is naturally extended for nested `cfor`s. Intuitively, the inner `cfor` is serialized as part of the iteration of the serialized outer `cfor`. So, in Figure 1, the programmer could have replaced the simple `for` in line 19 with a `cfor`, and the execution would be correct. It would also increase the available concurrency because multiple threads from the nested `cfor` iterations would be active at a node. However, in this case, it would not be efficient to use a `cfor` because the message and

latency overheads involved in starting and terminating the concurrent threads and remotely accessing nExamined and nToExamine would offset the potential concurrency gain from executing on multiple neighboring nodes of nodeIter. In general, a programmer must weigh the benefits of fine-grained concurrency through nested cfors against the start-up and finalization overheads of such concurrency.

Loose Variables. While serializability provides strong guarantees on the behavior of cfor loops, sensor network applications often have variables that do not need serializability semantics and can obtain timeliness and message efficiency benefits by using a looser consistency model. Examples include routing beacons that are used to maintain trees for sensor data collection, and sensor values that need to be filtered or smoothed using samples from neighboring nodes. Pleiades lets a programmer annotate such variables as loose, in which case accesses to these variables are not synchronized within a cfor. The consistency model used for loose variables closely follows release consistency semantics [13]. Writes to a loose variable can be re-ordered. The beginning of a new cfor statement or the end of any active cfor statement act as synchronization variables, ensuring that the current thread of control has no more outstanding writes.

In Figure 1, variables nToExamine and nExamined are annotated as loose (lines 9 and 10) in order to gain additional concurrency and avoid lock overhead on them. These annotations are based on the two observations that it is safe to examine a node in nToExamine multiple times, and that only a cfor iteration on nodeIter can remove the candidate node nodeIter from nToExamine. Alternatively, the programmer can derive the same concurrency in this case without using loose by temporarily storing the set of nodes that would be added to nToExamine in line 21 and deferring the add_node operations on this set until after statement 31. In general, the programmer can derive maximum concurrency while ensuring serializability by organizing her code so that writes on serialized variables happen toward the end of a cfor.

By default, loose variables are still reliably accessed, but the programmer can further annotate a loose variable to be unreliable, so that the implementation can use the wireless broadcast facility. In Section 4, we evaluate the street parking example with reliable loose variables and a separate application that primarily uses unreliable loose variables.

Automatic Control Flow Migration. Ultimately a centralized Pleiades program must be executed at the individual nodes of the network. As described in Section 3, the Pleiades implementation automatically partitions a Pleiades program into units of work to be executed on individual nodes and determines the best node on which to execute each unit of work in order to minimize communication costs. For example, the first five statements of the code (lines 6–10) execute at the node invoking reserve. The implementation then migrates the execution of statements in lines 11–16 to node n. This is because it is cheaper to simply transfer control to n than to first read isfree@n and later write it back if necessary. Similarly, each iteration of the cfor loop will execute at the node identified by the current value of nodeIter (line 17). While it does not happen in this example, the execution of a single cfor iteration can also successively migrate to other nodes.

2.3 Parking Cars with nesC

Pleiades provides several important advantages over the traditional node-level programming for sensor networks in use today. To make things concrete, we consider how the street-parking algorithm would be implemented in nesC. We describe two different nesC implementations: a centralized version that is relatively simple and reliable but highly inefficient, and a more complex distributed version that is efficient but unreliable. In contrast, the Pleiades version is both reliable and efficient.

2.3.1 A Centralized nesC Implementation

First, it is possible to implement a centralized version of the algorithm in nesC, wherein most of the algorithm is executed on a single node. The major advantage of this approach is its relative simplicity for programmers. However, this version is extremely inefficient in terms of both message cost and latency. Figure 2 shows the core functions that comprise such a program. The overall logic is similar to that of the Pleiades version from Figure 1. However, programmers must explicitly manage the details of inter-node communication. Because nesC uses an asynchronous, *split-phase* approach to such communication [5], the application's logic must be partitioned across multiple callback functions at remote read/write boundaries.

The control flow is as follows. A task reserve (line 9) is spawned on the node closest to the car, which, in turn, calls the closest_node function (line 10) in the Topology component (this component is not shown). Since all tasks in nesC run to completion, and since Topology.closest_node performs a split-phase lookup operation for the desired closest node, the callback function found_node is later invoked by Topology (line 12). The callback creates a new task transfer_control (line 14), which ultimately triggers doReserve on the closest node (line 21).

The rest of the algorithm then runs centrally on the closest node. doReserve, executing on closest, either finds itself free (line 22) or creates the nToExamine set with its current neighbor set (line 26). Next, it concurrently and asynchronously reads isfrees at nToExamine (line 27) using aread of the RemoteRW component (not shown). When the asynchronous read completes, it signals aread_done (line 29), and continue_reserve is called (line 30). Such reads are locally cached in the RemoteRW component, so that continue_reserve can synchronously read them in line 37. If no node with a free spot is found (lines 37–41), more neighboring nodes of the current nodes are searched using another asynchronous read (line 42), which, ultimately calls build_more_nodes (line 31).

Since the code is executed on a single node, this approach maintains a relatively straightforward structure, similar to that of the Pleiades code. The main drawback of this approach to node-level programming is inefficiency. Message cost is high because isfree of every node is centrally fetched and checked from a single node. In contrast, the Pleiades version from Figure 1 uses a cfor to allow each node to locally process its own data, using the code migration techniques described in Section 3. Thus, even for small example topologies of two-hop radius, it can be shown that the Pleiades version requires around half the messages required by the nesC version; this message count for Pleiades includes all control overhead for code migration and for ensuring serializability of cfors. The concurrent cfor iterations in Pleiades also find a free spot earlier than is possible in the nesC version. In the nesC version, continue_reserve in line 42 waits on RemoteRW.aread for all remote neighbors in nToExamine to be asynchronously read, and build_more_nodes in line 51 similarly waits until all remote isfrees in nToExamine are read.

2.3.2 A Distributed nesC Implementation

The Pleiades version of car parking in Figure 1 does a breadth-first search around the closest node, moving to the next depth in a distributed fashion only if no free slot is found in the current one. Unfortunately, a distributed implementation in nesC that provides the same behavior as the Pleiades version would be exceedingly complex. Such an implementation would require the programmer to manually implement many of the same concurrency control techniques that Pleiades automatically implements for cfors, as discussed in Section 3.2. For example, to ensure that exactly one free

```
1: module ReserveM {
2:  uses { ... }
3:  provides { ... }
4: } implementation {
5:  nodeset nToExamine, nExamined;
6:  boolean reserved, isfree, is_remote_free;
7:  node closest, reserved_node, req, iter, iter1;
8:  pos dst;

9:  task void reserve() {
10:  call Topology.closest_node(dst);
11: }
12: event void Topology.found_node(node n){
13:  closest = n; req=TOS_LOCAL_ADDRESS;
14:  post transfer_control();
15: }
16: task void transfer_control() {
17:  uint8_t i;
18:  //Trigger remote doReserve() at ''closest'' node
19:  //Also, send ''req'' and ''closest'' node values
20: }
21: task void doReserve() {
22:  if (isfree) {
23:   reserved_node=TOS_LOCAL_ADDRESS;
24:   call MsgInt.send_reply(req,FOUND);}
25:  else {
26:   nToExamine=call Topology.get_neighbors();
27:   call RemoteRW.aread(nToExamine,ISFREE); }
28: }
29: event void RemoteRW.aread_done(done_t done) {
30:  if (done==ISFREE) continue_reserve();
31:   else if (done==NEIGHBORS) build_more_nodes();
32: }
33: void continue_reserve() {
34:  for(iter=get_first(nToExamine);iter!=NULL;
        iter=get_next(nToExamine)) {
35:   remove_node(iter, nToExamine);
36:   add_node(iter, nExamined);
37:   if(is_remote_free=call RemoteRW.read(iter,ISFREE)){
38:    reserved_node=iter; reserved=TRUE;
39:    call RemoteRW.awrite(iter,ISFREE,0); }
40:  }
41:  if (!reserved)
42:   call RemoteRW.aread(nToExamine,NEIGHBORS);
43: }
44: void build_more_nodes(){
45:  nodeset nl;
46:  for(iter=get_first(nToExamine);iter!=NULL;
        iter=get_next(nToExamine)) {
47:   nl=(call RemoteRW.read(iter,NEIGHBORS));
48:   for(iter1=get_first(nl); iter1!=NULL;
         iter1=get_next(nl))
49:    if(!member(iter1,nExamined))
50:     add_node(iter1,nToExamine); }
51:  call RemoteRW.aread(nToExamine,ISFREE);
52: }
53:}
```

Figure 2. Reliable but inefficient street-parking in nesC.

```
1: module ReserveM {
2:  uses { ... }
3:  provides { ... }
4: } implementation {
5: boolean isfree, seen, reserved;
6: pos dst;
7: node start_node[], req, orig, reserved_node;
8: uint8_t cnt_start_node, hopcount;

9: task void reserve() {
10: call Topology.closest_node(dst);
11:}

12:event void Topology.found_node(node n){
13: orig=TOS_LOCAL_ADDRESS;
14: start_node[0]=n, req=n, hopcount=HOP_MAX;
15: cnt_start_node=1;
16: post transfer_control();
17:}

18:task void transfer_control() {
19: uint8_t i;
20: for (i=0;i<cnt_start_node;i++) {
21:  //Trigger remote doReserve() at every start_node[i]
22:  //Also, send each node our req, orig, hopcount values
23: }
24:}

25:task void doReserve(){
26: if(!seen) {seen=TRUE;}
27: if (isfree && !seen){
28:  reserved_node=TOS_LOCAL_ADDRESS;
29:  isfree=FALSE;
30:  call MsgInt.send_reply(req,FOUND); }
31: else flood_neighbors();
32:}

33:void flood_neighbors() {
34: nodeset nl=Topology.get_neighbors();
35: node iter;
36: hopcount--;
37: if (hopcount>0) {
38:  cnt_start_node=0;
39:  for (iter=get_first(nl);iter!=NULL;iter=get_next(nl))
40:   start_node[cnt_start_node++]=iter;
41:  post transfer_control(); }
42:}

43:event void MsgInt.receive_reply(node rep,msg_t msg){
44: if (msg==FOUND) {
45:  if (!reserved){
46:   reserved_node=rep;
47:   call MsgInt.send_reply(rep,ACCEPT);
48:   call MsgInt.send_reply(orig,FOUND); }
49:  else call MsgInt.send_reply(rep,REJECT); }
50: else if(msg==REJECT){isfree=TRUE;}
51:}
52:} //end implementation
```

Figure 3. Efficient but unreliable street-parking in nesC.

space is reserved for a car, the programmer would have to implement a form of distributed locking for conceptually central variables. In general the use of locking would then require manual support for distributed deadlock detection or avoidance. Similarly, to ensure that the closest free space is always found, the programmer would have to manually synchronize execution across the nodes in the network, to ensure that a depth d is completely explored before moving on to depth $d+1$.

Therefore, in practice a distributed version in nesC would forgo synchronization, as shown in Figure 3. Here we do a distributed flooding-based search around the closest node, in order to find a free spot. The control flow is as follows. After `reserve` is invoked (line 9), `doReserve` is ultimately triggered, in a manner similar to the previous version. The only difference here is that `doReserve` may be active at multiple nodes that receive the flooding request and may be activated multiple times by several neighbors (lines 39–41). Since a node must process a request exactly once even if its `doReserve` is triggered multiple times by its neighbors, `doReserve` uses a flag `seen` (line 26) to ignore all but the first request.

To limit the number of duplicate requests at a node, the code also suppresses broadcasts to neighbors when the `hopcount` reaches 0 (line 37). This is an effective technique when the network diameter is unknown and when we want to ensure the flooded requests prefer shorter hops from the flooding initiator (node `req` in line 14). `receive_reply` (line 43) is a callback that is invoked by the local message interface component `MsgInt` (not shown) whenever a remote node sends a message. When a spot is found at a remote node, it sends `FOUND` to the flooding initiator (line 30), which rejects all but the first successfully replying node (lines 45–49). If a remote node is rejected, it sets itself back to free (line 50).

As described earlier, the Pleiades version performs a breadth-first search on the topology, distributedly determining if there is a free slot at depth d before moving on to depth $d+1$. By contrast, the flooding approach starts up the free-slot determination concurrently at all network nodes by flooding the transfer of control. Given this distinction, two things follow. First, the Pleiades approach is always more message efficient, since it avoids multiple requests to the same node. Second, the flooding approach has lower latency, since it can find a spot more quickly when the free spot is far away. The flooding approach is also much more efficient in terms of both messaging costs and latency than the centralized nesC version shown in Section 2.3.1.

Despite the latency advantage, the code in Figure 3 is significantly less understandable and reliable than the Pleiades version. The programmer is responsible for explicitly managing the communication among nodes. For efficiency, this requires maintaining information about hop counts and other network details. It also requires that conceptually "central" variables be packaged up and passed among the nodes explicitly, taking care to maintain consistency. For example, a special protocol is used in `receive_reply` (lines 44–50) to ensure a consistent view of the `reserved` flag, in order to avoid having multiple nodes be reserved for the same car. Similarly, in `transfer_control` (lines 21–22), a node explicitly sends the values of the node originating the request and the node closest to the destination that initiated the search. In the Pleiades version, the combination of central variables and `cfor`s takes care of these low-level details automatically. Finally, the flooding version, unlike the other two versions, makes no guarantee that the first node to reply is the topologically closest node. So, if we want it to reliably return only a closest node, the `req` node executing `MsgInt.receive_reply` (line 43) must wait for an indeterminable amount of time before accepting a replying node, negating the latency advantage.

2.4 Other Features of Pleiades

Pleiades includes other language constructs to support the implementation of common sensor network idioms, which we briefly describe.

Sensors and Timers. As mentioned earlier, Pleiades uses special kinds of variables as an abstraction for sensors, which are critical components of sensor-network applications. Sensor readings are asynchronous events, and Pleiades provides a facility to synchronously wait for such an event to occur. In particular, Pleiades's `wait` function takes a sensor variable and returns when the sensor takes a reading. At that point, the associated variable contains the most recent reading and the program can take appropriate action. For example, this mechanism is used in order for the car-parking application to wait for notification that a parked car has left its spot, at which point the spot's sensor sets its associated `isfree` variable defined in line 2 of Figure 1 to `TRUE` (this operation is not shown), so that it can once again service remote `reserve` requests. A similar technique is used to model timers, which fire at some user-specified rate.

Modules. A Pleiades program consists of a number of *modules*, which are executed concurrently. Each module encapsulates a logically independent application-level computation, such as building a shortest path tree rooted at a given node, computing an aggregate, or routing application data to a given node. A module is a set of functions that can invoke each other and define and use global and local variables of both central and `nodelocal` type. Since modules are meant to be independent tasks, we currently provide no synchronization among modules.

3. Implementation

This section describes the Pleiades compiler and runtime system. The Pleiades compiler is built as an extension to the CIL infrastructure for C analysis and transformation [21]. Our compiler accepts a Pleiades program as input and produces node-level nesC code that can be linked with standard TinyOS components and the Pleiades runtime system. The Pleiades runtime system is a collection of TinyOS modules that orchestrates the execution of the compiler-generated nesC code across the nodes in the network.

The Pleiades compiler and runtime cooperate to tackle two key technical challenges. First, they must partition a Pleiades program into chunks that can be executed on individual nodes and determine at which node to run each chunk, striving to minimize communication costs. Second, they must provide concurrent but serializable execution of `cfor`s. We discuss each challenge in turn.

3.1 Program Partitioning and Migration

Partitioning. The Pleiades compiler performs a dataflow analysis in order to partition a Pleiades program into a set of *nodecuts*. Each nodecut is then converted into a nesC task [5], to be executed by the Pleiades runtime system on a single node in the network. At one extreme, one could consider the entire Pleiades program to be a single nodecut and execute it at one node, fetching node-local and central variables from other nodes as needed (moving the data to the computation). The other extreme would be to consider each instruction in the Pleiades program as its own nodecut, executing it on the node whose local variables are used in the computation (moving the computation to the data). Both of these strategies lead to generated code that has high messaging overhead and high latency, in the first case due to the on-the-fly fetching of individual variables, and in the second case due to the per-instruction migration of the thread of control.

We adopt a compilation strategy for Pleiades that lies in between these two extremes, involving both control flow migration and data movement. A nodecut can include any number of state-

ments, but it must have the property that just before it is to be executed, the runtime system can determine the location of all the node-local variables needed for the nodecut's execution. We therefore define a nodecut as a subgraph of a program's control-flow graph (CFG) such that for every expression of the form `var@e` in the subgraph, the l-values in `e` have no reaching definitions within that subgraph.

Given this property, the runtime system can retrieve all the necessary node-local and central variables concurrently, before beginning execution of a nodecut, which improves the latency immensely over the first strategy above. At the same time, because the runtime system has information about the required node-local variables, it can determine the best node (in terms of messaging costs) at which to execute the nodecut, thereby obtaining the benefits of the second strategy above without the latency and message costs of per-statement migration.

Intuitively, the goal is to make each nodecut as large as possible, in order to minimize the control and data costs associated with a migration. Since a nodecut runs to its completion without any further communication, this approach would statically minimize the total communication cost of a program. We make the goal of minimizing migrations precise by striving to minimize the total number of edges in the program's CFG that cross from one nodecut to another, since each such edge represents a migration of the dynamic thread of control from one sensor node to another. This optimization problem is exactly equivalent to the directed unweighted multi-cut problem, which is known to be NP-complete [1]. Therefore, instead of finding the optimal partition of a CFG into nodecuts, the Pleiades compiler uses a heuristic algorithm that works well in practice, as shown in Section 4.

The algorithm starts by assuming that all CFG nodes are in the same nodecut and does a forward traversal through the CFG, creating new nodecuts along the way. For each CFG node n containing an expression of the form `var@e`, we find all reaching definitions of the l-values in `e` and collect the subset R of such definitions that occur within n's nodecut. If R is nonempty, we induce a new nodecut by finding a CFG node d that dominates node n and post-dominates all of the nodes in R. Node d then becomes the entry node of the new nodecut. Any such node d can be used, but our implementation uses simple heuristics that attempt to keep the bodies of conditionals and loops in the same nodecut whenever possible. The implementation also uses heuristics to increase the potential for concurrency. For example, the body of a `cfor` is always partitioned into nodecuts that do not contain any statements from outside the `cfor`, so that these nodecuts can be executed concurrently.

The five nodecuts computed by our algorithm for the street-parking example in Figure 1 are shown in Figure 4. Nodecut 2 is induced due to the use of `isfree@n` in line 11 of Figure 1, since `n` is defined in line 8. The transitions from nodecut 2 to 3 and nodecut 3 to 4 are induced to keep the `cfor` body separate from statements outside the loop, as mentioned above. Further, an extra nodecut is induced within the `cfor` body (nodecut 5) to maximize read concurrency. The heuristic attempts to separate read and written variables into different nodecuts so that the acquisition of write locks, which is done before a nodecut starts execution, can be delayed until the write locks are actually required.

In the current implementation we assume that a Pleiades program does not create aliases among node variables. Such aliasing has not been necessary in any of our experiments with the Pleiades language so far. It is straightforward to augment our algorithm for generating nodecuts to handle node aliasing by consulting a static may-alias analysis.

Control Flow Migration. The Pleiades runtime system is responsible for sequentially (ignoring `cfor` for the moment) executing each nodecut produced by the compiler across the sensor network.

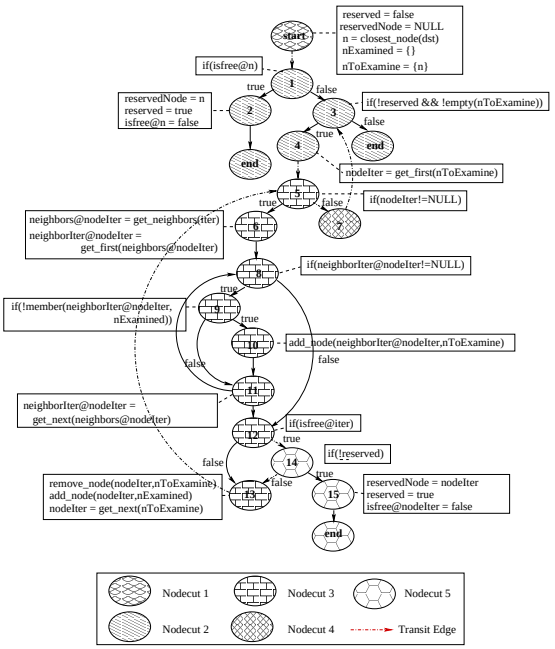

Figure 4. Nodecuts generated for the street-parking example.

When execution of a nodecut C completes at some node n, that node's runtime system determines an appropriate node n' at which to run the subsequent nodecut C' and migrates the thread of control to n'. All of the Pleiades program's central variables migrate along with the thread of control, thereby making them available to C'. Because of the special property of nodecuts, the runtime system knows exactly which node-local variables are required by C', so these variables are also concurrently fetched to n' before execution of C' is begun.

To determine where the next nodecut should be executed, the runtime uses the overall migration cost as the metric. The runtime knows the number of node-local variables needed from each node for executing the next nodecut as well as the distances (the number of radio hops) of these nodes relative to each other according to the current topology. The runtime chooses the node that minimizes the cost of transfers from within this set. For example, nodecut 2 in Figure 1 accesses the node-local variable `isfree@n`, as well as two central variables `reserved` and `reservedNode`. The cost of running this nodecut at the node executing nodecut 1 is the cost of fetching the value of `isfree` from `n` at the beginning of nodecut 2 and writing back `isfree` if necessary. This cost is two reliable messages across multiple radio hops. By contrast, if the runtime at nodecut 1 hands off nodecut 2 to node `n`, the cost is that of transferring the thread of control along with the central variables. This is only one reliable message across the same number of hops. So, Pleiades executes nodecut 2 at `n`.

Since the nodecuts along with the set of node-local variables accessed in each nodecut are statically supplied by the compiler, our migration approach thus exploits a novel combination of static and dynamic information in order to optimize energy efficiency. We note that this approach does not require every node to keep a fully consistent topological map, but only the relative distances of the nodes involved in the nodecut. In our current implementation, nodes use a statically configured topological map in order to make the migration decision; we will explore lightweight, dynamic

approaches to determine approximate topological maps as part of future work.

3.2 Serializable Execution of cfors

To execute a cfor loop, the Pleiades runtime system forks a separate thread for each iteration of the loop. We call the forking thread the cfor *coordinator*. Program execution following the cfor only continues once all the forked threads have joined. Each forked thread is initially placed at the node representing the value of the variable the cfor iterates over, and any subsequent nodecuts in the thread are placed using the migration algorithm for nodecuts described above. A forked thread may itself execute a cfor statement, in which case that thread becomes the coordinator for the inner cfor, forking threads and awaiting their join.

To provide reliability in the face of concurrency, Pleiades ensures serializability of cfor loops. This allows programmers to correctly understand their Pleiades programs in terms of a sequential execution semantics. The Pleiades compiler and runtime ensure serializability by transparently locking variables accessed in each cfor body. The use of locking has the potential to cause deadlocks, so we also provide a novel distributed deadlock detection and recovery algorithm.

Distributed Locking. To ensure serializability, the Pleiades implementation protects each node-local and central variable accessed within a cfor iteration with its own lock. We employ a pessimistic locking approach, since this consumes less memory than optimistic approaches such as versioning. To ensure serializability, a lock must be held until the end of the outermost cfor iteration being executed; thus, the implementation uses strict two-phase locking. However, locks are acquired on demand rather than at the beginning of the cfor iteration, thereby achieving greater concurrency. To further increase concurrency, our algorithm distinguishes between read and write locks. Readers can be concurrent with one another, while a writer requires exclusive access. The implementation acquires locks at the granularity of a nodecut. This allows the locks to be fetched along with the associated variables before the nodecut's execution, decreasing messaging costs.

Our algorithm acquires locks in a hierarchical manner. Each cfor coordinator keeps track of which locks it holds, the type of each lock (read or write), which of its spawned threads are currently using each lock, and which of its threads are currently blocked waiting for each lock. When a nodecut requires a particular lock, it asks the coordinator of its innermost enclosing cfor for the lock. If the coordinator has the lock, it either provides the lock or blocks the thread, depending on the lock's current status, and updates the lock information it maintains appropriately. If the coordinator does not have the lock, it recursively requests the lock from *its* cfor coordinator, thereby handling arbitrarily nested cfors. Once the top-level cfor coordinator has been reached, it acquires the lock from the variable's owner and grants the lock to the requesting thread (who will then grant the lock to *its* requesting thread, and so on down to the original requester). Once a thread has obtained the lock on a variable, it fetches the actual value of the variable directly from the owner. When a spawned thread joins, it returns its locks to its cfor coordinator, who may therefore be able to unblock threads waiting for these locks. Also, if any of the locks owned by the joining thread were write locks, before releasing the locks it writes back the current value of the variable at the owner. It is possible to argue that this locking scheme always results in a serializable execution of a cfor, but we omit the details due to space constraints.

Let us revisit the street parking example in Figure 1. For each cfor iteration, the Pleiades runtime at the coordinator sends a message containing the fork command to each of the remote nodes selected for execution. Each node initially acquires a read and write lock respectively on its own versions of the node-local variables isfree and neighbors. isfree uses a read lock instead of a write lock even though it can potentially be modified in line 26, because using a read lock first and then upgrading it to a write lock if the conditional in line 23 succeeds significantly enhances concurrency. On receiving these locks, the threads fetch the variable values from the owners and begin concurrent execution of the initial nodecut of the cfor (nodecut 3 in Figure 4). Threads that run on nodes with an occupied parking space fail the if condition in line 23, release their locks, and join with the cfor coordinator. Threads on nodes that have a free space contend for a write lock on central variables reserved and reservedNode and have to execute the second nodecut of the cfor sequentially. The first thread to do so is selected as the winner, and other nodes do not change their isfree status.

Distributed Deadlock Detection and Recovery. While the locking algorithm ensures serializability of cfors, it can give rise to deadlocks. One possibility would be to statically ensure the absence of deadlocks, for example via a static or dynamic global ordering on the locks. However, such an approach would be very conservative in the face of cfors containing multiple nodecuts, nested and conditional cfors, or cfors that contain updates to node variables, thereby overly restricting the amount of concurrency possible. Further, we expect deadlocks to be relatively infrequent. Therefore Pleiades instead implements a dynamic scheme for distributed deadlock detection and recovery. While such schemes can be heavyweight and tricky in general [4], we exploit the fork-join structure of a cfor to arrive at a simple and efficient state-based deadlock detection algorithm. Our algorithm requires only two bits of state per thread, does not rely on timeouts, and finds deadlocks as soon as it is safe to determine the condition. Furthermore, this algorithm is implemented by the compiler and runtime, without any programmer intervention.

We require every thread to record its state during execution, which is either executing, blocked, or joined. We define a cfor coordinator to be executing if at least one of the coordinator's spawned threads is executing, blocked if at least one of the coordinator's threads is blocked and none are executing, and joined if all of the coordinator's threads are joined. A thread can easily update its state appropriately as its locks are requested and released during the locking algorithm described above, in the process also causing the thread to recursively update the state of its cfor coordinator. The program is deadlocked if and only if the top-level cfor coordinator ever has its state set to blocked.

Once a deadlock has been detected, we use a simple recovery algorithm. Starting from the top-level cfor coordinator, we walk down the unique path to the highest thread in the tree of cfor coordinators that has at least two blocked child threads. We then release all locks held by these blocked threads and re-execute them in some sequential order. This simple approach guarantees that we will not encounter another deadlock after restart. To support re-execution, each thread records the initial values of all variables to which it writes, so that the variables previously updated at their owners can be rolled back appropriately during deadlock recovery. We assume that the iterations are idempotent, so there are no harmful side-effects of re-execution. This is true in many sensor networks programs, which primarily involve sensing and actuation as side effects.

4. Evaluation

We have implemented the Pleiades compiler and runtime described in Section 3. In this section, we describe an evaluation of this implementation for various applications, with Pleiades running on TelosB Tmote Sky motes. We first discuss the performance of a

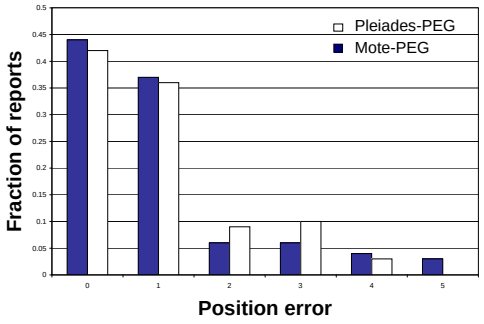

Figure 5. PEG application error.

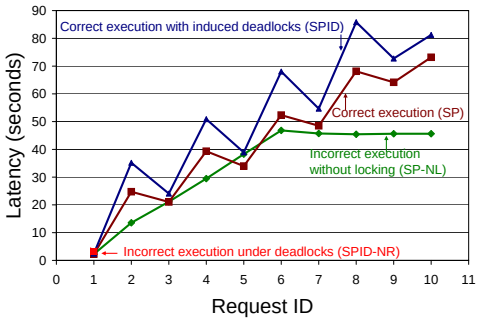

Figure 6. Street parking latency.

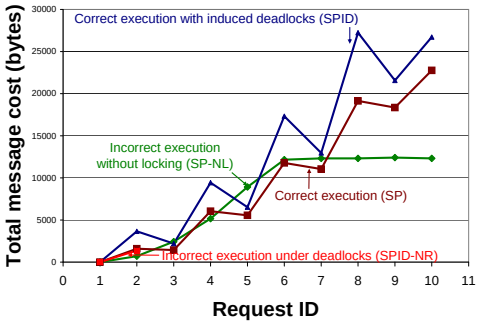

Figure 7. Street parking message cost.

Pleiades application relative to a nesC implementation of that same application. Then, we quantify the performance of Pleiades support for serializability and nodecut migration.

Pleiades and nesC Comparison. We compare a Pleiades implementation of a *Pursuit-Evasion Game (PEG)* against a hand-coded node-level nesC implementation of the same application written by others [7] on a 40 node mote testbed. PEGs [26] have been explored extensively in robotics research. In a PEG, multiple robots (the pursuers) collectively determine the location of one or more evaders using the sensor network, and try to corral them.

The mote implementation of this game consists of three components: a leader election module performs data fusion to determine the centroid of all sensors that detect an evader; a landmark routing module routes leader reports to a landmark node; in turn, the landmark routes reports to pursuers. The Pleiades version of PEG implements the leader election component of PEG, and leverages the routing provided by the Pleiades runtime to route the leader reports directly to the pursuer. It is less than a tenth of the nesC implementation in terms of lines of code (63 lines as opposed to 780). An important feature of this application is that it requires no serializability semantics for the core leader election module; in fact, the data we present below were obtained using a version of Pleiades that did not support serializability. We also implemented PEG on Pleiades with full serializability support for leader election, and found that it does not incur additional overhead due to locking, because leader election needs only read locks, which are acquired once at the beginning, and retained until the end.

Figure 5 depicts the main application-perceived measure of performance, the *error in position estimate* on a topological (reduced) map of the environment [16]. This figure is highly encouraging; the Pleiades program exhibits comparable error to a hand-crafted nesC program. The frequency of 2- and 3-hop errors is slightly higher for Pleiades-PEG than for mote-PEG. On the other hand, Pleiades-PEG does not incur instances of 5-hop error that mote-PEG does.

We also measured the latency between when a mote detects an evader and when the corresponding leader report reaches the pursuer. Mote-PEG has noticeably lower latency than Pleiades-PEG, but for most nodes (about 80%), this latency difference is within a factor of two. This is because our implementation of Pleiades is unoptimized for handling `cfor` forks and joins, and because our nodecut placement implementation relies on relatively static hop count information. There is scope for improving both significantly.

The average network overhead for mote-PEG is 193 messages per minute, while for Pleiades-PEG is 243. The minimum and maximum network overhead is 137 and 253 for mote-PEG and 146 and 341 for Pleiades-PEG. While these results merit further study, they suggest that Pleiades performance can be comparable to that of node-level programming.

Serializability Evaluation. We ran the street-parking application of Figure 1 on a 10-node chain mote topology. This topology is an extreme configuration, and thus stresses our serializability implementation, because the efficiency of packet delivery in a chain of wireless nodes drops dramatically with the length of the chain. In our experiments, 10 requests for free spots arrive sequentially at the node in the center of the chain. To illustrate the power of Pleiades's serializability guarantees, and to understand its performance, we ran four different versions of the application: **SP-NL**, in which we configured the Pleiades compiler and runtime to disable locking; **SP**, which uses the complete Pleiades compiler and runtime for locking, deadlock detection and recovery; **SPID-NR**, in which we induced a deadlock into the application and configured the Pleiades runtime to disable deadlock recovery; and **SPID**, which uses the complete Pleiades implementation with the deadlock-induced application. To improve performance, we implemented message aggregation for lock requests and forwarded locks across consecutive nodecuts.

As expected, **SP** and **SPID** execute correctly, assigning exactly one spot to each request. **SPID-NR** fails to allocate a spot to all but the first request; in the absence of recovery code, the program deadlocks after the first request. Finally, **SP-NL** violates the correctness requirements of the application, correctly satisfying the first request, but assigning two free spots in each direction of the center node for the next four requests; consequently, it also fails to satisfy the last four requests.

Figure 6 plots the time taken to assign a spot to the request, and Figure 7 plots the total number of bytes transmitted over the network for each request. The same qualitative observations may be drawn from both graphs. **SP** and **SPID** message cost and latency increase since successive requests have to search farther out into the network to find a free spot. However, for the initial requests,

the overhead of **SP** is comparable to that of **SP-NL**. Moreover, **SPID** message cost and latency are only moderately higher than **SP**. The difference is attributable to the sequential execution of the `cfor` threads during deadlock recovery, with rollback overhead being negligible. The periodic spikes in both plots arise because, for even-numbered requests, there are two free spots at the same distance away from the requester that contend to satisfy the request. These two free spots also cause a deadlock in the case of **SPID**. Finally, the latency and overhead of **SP-NL** flatten out for later requests because they each incur the same cost: they search the entire network for a free spot and fail, because spots were incorrectly over-allocated during earlier requests.

Thus, our Pleiades implementation correctly ensures serializability and incurs moderate overhead for deadlock detection and recovery. The absolute overhead numbers imply that even for the request which encounters the highest overhead, the average bandwidth of a node used by Pleiades is around 250bps, with the maximum being 1kbps at the node where the requests come in. This is quite reasonable, considering that the maximum data rate for the TelosB motes is 250kbps.

The absolute latency seems modestly high compared to the expected response time for human interactivity. For example, the last request takes almost a minute and a half to satisfy. This is an artifact of the end-to-end reliable transport layer that Pleiades currently uses, which waits for 2 seconds, before trying to resend a packet that has not been acknowledged as received. We believe that the overall latency can be significantly reduced by optimizing the transport layer.

The Benefits of Migration. Finally, we briefly report on a small experiment on a 5-node chain that quantifies the benefit of Pleiades's control flow migration. In this application, a node accesses node-local nodesets from other nodes more than a hop away, so that application-level network information can be gathered. Without migration, the total message cost is 780 bytes, while, with migration, it is 120 bytes. Thus, we see that, even for small topologies, control flow migration can provide significant benefits.

5. Related Work

Pleiades is related to many programming concepts developed in parallel and distributed computing. We classify related work into three broad categories. They are embedded and sensor systems languages, concurrent and distributed systems languages, and parallel programming languages.

Embedded and Sensor Networks Languages. Several researchers have explored programming languages for expressing the global behavior of applications running on a network of less-constrained 32-bit embedded devices (e.g., iPAQs). Pleiades's programming model borrows from our earlier work on Kairos [8], an extension to Python that also provides support for iterating over nodes and accessing node-local state. However, Kairos does not support automatic code migration or serializability. Kairos provides support for application-specific recovery mechanisms [9], which Pleiades lacks. SpatialViews [25] is an extension to Java that supports an expressive abstraction for defining and iterating over a virtual network. In SpatialViews, control flow migrates to nodes that meet the application requirements. To avoid concurrency errors, SpatialViews restricts the programming model within iterators.

Regiment [24] is a functional programming language for centrally programming sensor networks that models all sensor data generated within a programmer-specified region as a data stream. Regiment is a purely functional language, so the compiler can potentially optimize program execution extensively according to the network topology. On the other hand, since the language is side-effect-free, it does not support the ability to update node-local state. For example, the car parking application would be much harder to write in Regiment.

TinyDB [19] provides a declarative interface for centrally manipulating the data in a sensor network. This interface makes certain applications reliable and efficient but it is not Turing-complete. Because TinyDB lacks support for arbitrary computation at nodes, it cannot be easily used to implement the kinds of applications we support, like car parking. Research on Abstract Regions [29] provides local-neighborhood abstractions for simplifying node-level programming. This work is focused on programmability and efficiency and does not provide support for consistency or reliability.

Concurrent and Distributed Systems. Argus [17] is a distributed programming language for constructing reliable distributed programs. Argus allows the programmer to define concurrent objects and guarantees their atomicity and recovery through a nested transactions facility, but makes the programmer responsible for ensuring serializability across atomic objects and for handling any application-level deadlocks. Recently, composable Software Transactional Memory (STM) [10] has been proposed as an abstraction for reliable and efficient concurrent programming. Also, Atomos [2] is a new programming language with support for implicit transactions and strong atomicity features.

Our `cfor` construct, with its serializability semantics and nesting ability, is designed in a similar spirit—a concurrency primitive with simplicity, efficiency, reliability, and composability as goals. Unlike these systems, however, Pleiades derives concurrency from a set of loosely coupled and distributed, resource constrained nodes. Therefore, the Pleiades implementation of `cfor` emphasizes message and memory efficiency over throughput or latency. For the same reason, it uses a simple distributed locking algorithm for serializability and a novel low-state algorithm for distributed deadlock detection and recovery. Pleiades' `cfor`s are also similar to atomic sections in Autolocker [20] in that both implementations use strict two-phase locking. But Autolocker guarantees the absence of deadlocks through pessimistic locking, while Pleiades uses an optimistic locking model in which locks are acquired or upgraded as needed, and any deadlocks are detected and recovered by the runtime.

Approaches to automatic generation of distributed programs have also been explored. For example, Coign [12] is a system for automatically partitioning coarse-grained components. MagnetOS [18] also has support for partitioning a program written to a single system image abstraction. A program transformation approach for generating multi-tier applications from sequential programs is described in [22]. All these systems are primarily meant for partitioning and distribution of programs into coarse-grained components, that can then be run concurrently on multiple nodes. Pleiades differs from these systems in generating nesC programs with fine-grained nodecuts and supporting lightweight control flow migration across such nodecuts.

Parallel Processing Languages. Pleiades differs from prior parallel and concurrent programming languages such as Linda [6] and Split-C [3] by obviating the need for explicit locking and synchronization code. Pleiades also differs from automatic parallelization languages such as High Performance Fortran [14] by equipping the compiler and runtime with serializability facilities. This is because parallel programming languages focus on data parallelism on mostly symmetric processors, leaving to the programmer the responsibility of ensuring deadlock and livelock freedom at the application level. On the other hand, Pleiades offers task-level parallelism, where data sharing among sensor nodes is common, and where it is desirable to offload the correct implementation of concurrency to the compiler and runtime.

6. Conclusions and Future Work

Pleiades enables a sensor network programmer to implement an application as a central program that has access to the entire network. This critical change of perspective simplifies the task of programming sensor network applications on motes and can still provide application performance comparable to hand-coded versions. Pleiades employs a novel program analysis for partitioning central programs into node-level programs and for migrating control flow across the nodes. Pleiades also provides a simple construct that allows a programmer to express concurrency. This construct uses distributed locking along with simple deadlock detection and recovery to ensure serializability. Together, these features ensure that Pleiades programs are understandable, efficient, and reliable. Our implementation of these features runs realistic applications on memory-limited motes.

While our current Pleiades implementation is robust to one aspect of network dynamics (packet loss), the failure of a `cfor` coordinator can cause an application to fail. We are currently implementing support for handling node dynamics such as crashes and additions through a simple retry-based mechanism that extends the reliable routing and transport mechanisms already present in the runtime. The basic idea is that node failures trigger an undo mechanism similar to that already used for deadlock recovery, which allows the initiator of the computation to retry. This approach naturally fits the semantics of the `cfor` construct and complements our programmability, efficiency and reliability contributions.

In future work, we intend to optimize the message and latency costs of our implementation by exploring more efficient message batching alternatives. We also plan to support various relaxed consistency models as alternatives to serializability. In addition, we would like to allow the programmer to be able to easily trade off quality of results for time of distributed execution. Finally, we plan to examine approaches to specifying sophisticated power management policies in Pleiades.

Acknowledgments

We thank Ki-Young Jang and Marcos Veiera for help with motePEG, Jeongyeup Paek and Omprakash Gnawali for help with the transport software, Avinash Sridharan for help with mote hardware, and Ben Greenstein and the anonymous reviewers for their comments that helped improve the paper.

References

[1] G. Calinescu, C. G. Fernandes, and B. Reed. Multicuts in unweighted graphs with bounded degree and bounded tree-width. *LNCS 1998*.

[2] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos transactional programming language. In *PLDI 2006*.

[3] D. E. Culler, A. C. Arpaci-Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. A. Yelick. Parallel programming in Split-C. In *Supercomputing, 1993*.

[4] A. K. Elmagarmid. A survey of distributed deadlock detection algorithms. *SIGMOD Rec., 1986*.

[5] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI 2003*.

[6] D. Gelernter and N. Carriero. Coordination languages and their significance. *Commun. ACM, 1992*.

[7] O. Gnawali, B. Greenstein, K.-Y. Jang, A. Joki, J. Paek, M. Vieira, D. Estrin, R. Govindan, and E. Kohler. The TENET architecture for tiered sensor networks. In *SenSys 2006*.

[8] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using Kairos. In *DCOSS 2005*.

[9] R. Gummadi, N. Kothari, T. Millstein, and R. Govindan. Declarative failure recovery for sensor networks. In *AOSD 2007*.

[10] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP 2005*.

[11] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. *SIGOPS Oper. Syst. Rev., 2000*.

[12] G. C. Hunt and M. L. Scott. The Coign automatic distributed partitioning system. In *OSDI 1999*.

[13] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *ISCA 1992*, 1992.

[14] C. Koelbel. An overview of High Performance Fortran. *SIGPLAN Fortran Forum*, 11(4), 1992.

[15] L. Krishnamurthy, R. Adler, P. Buonadonna, J. Chhabra, M. Flanigan, N. Kushalnagar, L. Nachman, and M. Yarvis. Design and deployment of industrial sensor networks: Experiences from a semiconductor plant and the north sea. In *SenSys 2005*.

[16] B. J. Kuipers and Y.-T. Byun. A Robust Qualitative Method for Spatial Learning in Unknown Environments. In *AAAI 1988*.

[17] B. Liskov. Distributed programming in Argus. *Commun. ACM*, 31(3), 1988.

[18] H. Liu, T. Roeder, K. Walsh, R. Barr, and E. G. Sirer. Design and implementation of a single system image operating system for ad hoc networks. In *MobiSys 2005*.

[19] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *SIGMOD 2003*.

[20] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: Synchronization inference for atomic sections. In *POPL 2006*.

[21] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Conference on Compilier Construction, 2002*.

[22] M. Neubauer and P. Thiemann. From sequential programs to multi-tier applications by program transformation. In *POPL 2005*.

[23] R. Newton, Arvind, and M. Welsh. Building up to macroprogramming: An intermediate language for sensor networks. In *IPSN 2005*.

[24] R. Newton and M. Welsh. Region Streams: Functional macroprogramming for sensor networks. In *DMSN 2004*.

[25] Y. Ni, U. Kremer, A. Stere, and L. Iftode. Programming ad-hoc networks of mobile and resource-constrained devices. In *PLDI 2005*.

[26] C. Sharp, S. Schaffert, A. Woo, N. Sastry, C. Karlof, S. Sastry, and D. Culler. Design and implementation of a sensor network system for vehicle tracking and autonomous interception. In *EWSN 2005*.

[27] D. Shoup. New York Times Op-Ed: Gone Parkin', `http://www.nytimes.com/2007/03/29/opinion/29shoup.html`.

[28] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong. A macroscope in the Redwoods. In *SenSys 2005*.

[29] M. Welsh and G. Mainland. Programming sensor networks using Abstract Regions. In *NSDI 2004*.