MacroLab: A Vector-based Macroprogramming Framework for Cyber-Physical Systems

Timothy W. Hnat, Tamim I. Sookoor, Pieter Hooimeijer, Westley Weimer, and Kamin Whitehouse
Department of Computer Science, University of Virginia
Charlottesville, VA, USA
{hnat,sookoor,pieter,weimer,whitehouse}@cs.virginia.edu

ABSTRACT

We present a macroprogramming framework called *MacroLab* that offers a vector programming abstraction similar to Matlab for Cyber-Physical Systems (CPSs). The user writes a single program for the entire network using Matlab-like operations such as `addition`, `find`, and `max`. The framework executes these operations across the network in a distributed fashion, a centralized fashion, or something between the two – whichever is most efficient for the target deployment. We call this approach *deployment-specific code decomposition* (DSCD). MacroLab programs can be executed on mote-class hardware such as the Telos [24] motes. Our results indicate that MacroLab introduces almost no additional overhead in terms of message cost, power consumption, memory footprint, or CPU cycles over TinyOS programs.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and Embedded Systems; D.1.3 [**Programming Techniques**]: Concurrent Programming–Distributed programming

General Terms

Design, Languages, Performance

Keywords

Cyber-Physical Systems, Embedded Networks, Macroprogramming, Programming Abstractions

1. INTRODUCTION

Cyber-Physical Systems (CPSs) combine low-power radios with tiny embedded processors in order to simultaneously cover large geographic areas *and* provide high-resolution sensing/actuation. This revolutionary technology has begun to deliver a new generation of engineering systems and scientific breakthroughs. However, CPSs are extremely difficult to program; building even a simple application entails several complex tasks such as distributed programming, resource management, and wireless networking. CPSs have reached a reasonable degree of technological maturity, but their impact and widespread adoption is limited by the complexity of their software.

In this paper, we present the *MacroLab* framework for CPS software development. We call MacroLab a *macroprogramming* system because the user writes a single macroprogram for the entire CPS and the framework automatically decomposes it into to a set of *microprograms* that are loaded onto each node. MacroLab provides a vector programming abstraction using syntax similar to Matlab, which already has broad adoption among scientists and engineers. Data from sensors and actuators are manipulated just like other numerical vectors, making MacroLab programs similar to and easy to integrate with existing scientific software. Its traditional, imperative programming model supports general-purpose programming and is a natural way to encode CPS applications involving both sensing and actuation.

MacroLab introduces a new data structure called a *macrovector* which can be used to store in-network data such as sensor readings. Conceptually, each element of a macrovector corresponds to a different node in the network, but macrovectors can be stored in different ways. Each element can be on its corresponding node, all elements can be on a central server, or all elements can be replicated on all nodes. No matter how a macrovector is stored, it can support standard vector operations such as `addition`, `find`, and `max`. These operations may run in parallel on a distributed macrovector, sequentially on a centralized macrovector, or somewhere in between. Thus, by changing the representation of each macrovector, MacroLab can decompose a macroprogram in the way that is most efficient for a particular deployment. For example, it may use centralized representations for small star topologies and distributed representations in large mesh networks. We call this approach *deployment-specific code decomposition (DSCD).*

In contrast to systems like TinyOS [15], the MacroLab programmer specifies application logic in terms of abstract computation and does not need to explicitly control data partitioning or message passing from within the source code. Instead, these tasks are performed automatically as compile-time and run-time optimizations. By separating application logic from program decomposition, MacroLab can improve code portability, increase code reuse, and decrease overall development costs. Furthermore, it can reduce overall resource consumption. Our results show that automatically

SenSys'08, November 5–7, 2008, Raleigh, North Carolina, USA.

choosing a decomposition for each deployment can reduce message passing by up to 50 percent over using a single decomposition for all deployments.

MacroLab provides a clear cost model so that the programmer can write code that produces efficient and optimized decompositions. This is analogous to the idea in Matlab that vectorized code is more efficient than `for` loops. MacroLab does not compromise on power or memory efficiency in order to provide a high-level vector programming abstraction and our results indicate that MacroLab programs are just as efficient as normal TinyOS programs.

2. BACKGROUND AND RELATED WORK

The predominant way to program a CPS today is with *node-level programming*. The user writes a *microprogram* that is loaded onto each node that specifies when it should sense, actuate, or send messages and how to respond to incoming messages or hardware events. This is a difficult and error-prone way to design a system and would be similar to an architect designing a building by generating step-by-step instructions for each construction worker instead of generating blueprints. Node-level programming is so difficult that many so-called *macroprogramming* systems have recently been proposed: the user writes a single program that specifies global operations for the entire CPS and the framework automatically decomposes this into a set of microprograms for each node.

Macroprogramming systems have been proposed with a wide variety of abstractions and programming models, each of which is designed to make programming easier for some class of applications. For example, database-like systems such as TinyDB [18] and Cougar [34] allow the user to specify the desired data using declarative SQL-like queries. These systems are most suitable for data collection applications where the desired data can be described with a declarative query. Several systems such as Hood [32], Regions [30], and Proto [2] are designed for spatial applications and allow users to specify operations over groups, neighborhoods, or regions in space. Other systems such as Semantic Streams [31], Flask [20], and Regiment [22] allow users to specify global operations in terms of *stream operators*. These are most suitable for defining a static set of long-running operations over streams of sensor data. MacroLab is perhaps most similar to imperative macroprogramming abstractions like Marionette [33], Pleiades [14], and Kairos [10]. These systems support general-purpose programming with a traditional imperative programming model. MacroLab is the first macroprogramming system for CPSs to provide vector programming, a powerful and concise abstraction that already has wide adoption among scientists and engineers.

Several existing systems allow users to write imperative programs that can then be distributed across multiple processors for the purposes of high performance computing. These include High Performance Fortran (HPF) [26], Fortran D [11], and Split-C [5]. The fundamental difference between these approaches and MacroLab is their dependence on the user to specify how the data and operations should be distributed. For example, Fortran D uses the statments `decomposition`, `align`, and `distribute` to specify how to execute a program on multiple processors. In contrast, MacroLab programs do not specify how to map the computation onto the network. In fact, the system will create a different mapping for each network on which the program is executed.

Several existing systems such as MagnetOS [16], Coign [12], and J-Orchestra [29] can automatically decompose a program and distribute it across a network in order to minimize network traffic. Similar to MacroLab, these systems use program profiling to tailor the decomposition to a specific network topology. In contrast to MacroLab, these systems decompose programs at the *object level*. MagnetOS and J-Orchestra break a program up at the boundaries of Java objects and use Java RMI between segments of the program. Coign requires programs to conform to Microsoft's Component Object Model (COM) and breaks them up at the boundary of the COM objects. MacroLab introduces parallelism at the level of individual operations instead of at the level of objects or software components.

Several existing systems allow the user to specify parallel operations using parallel data structures. SET Language (SETL) [27] provides primitive operations such as set membership, union, intersection, and power set construction, which can be applied in parallel to elements of *unordered sets*. Starlisp (*Lisp) can apply vector operations such as vector addition and multiplication over *Parallel Variables (PVARS)* which are vectors with one element per processor. Similarly, NESL allows parallel operations on *sequences*. These are similar to MacroLab's parallel vector operations on *macrovectors*. However, MacroLab goes beyond these systems by employing *multiple* underlying representations of a macrovector. Unordered sets, PVARS, and sequences can only be decomposed in one way while macrovectors are decomposed in one of many different ways depending on the topology over which the program is executed. To our knowledge, MacroLab is the first system that can perform automatic, topology-specific decomposition on programs describing parallel operations on parallel data structures.

3. MACROLAB

MacroLab allows the user to write a single program that is simple, robust, and manageable and then automatically decomposes it depending on the target deployment. A *program decomposition* is a specification of where data is stored in the network and how messages are passed and computations are performed in order to execute the program. A macroprogram may be decomposed into *distributed* operations for a large mesh network, where data is stored on every node and network operations are performed in-network. It could also be decomposed into *centralized* operations for a small star topology, where all data is collected to a central base station. A program may also be decomposed into many points on the spectrum between purely centralized or purely distributed code. The implementation could also use group-based data processing or in-network aggregation.

MacroLab's overall architecture is depicted in Figure 1. A macroprogram (described in Section 3.1) is passed to the decomposer (Section 3.2) which generates multiple decompositions of the macroprogram. Each decomposition is passed to the cost analyzer (Section 3.3) which calculates the cost of each with respect to the cost profile of the target deployment. This cost profile must be provided by the user and may include information such as the topology, power restrictions, and descriptions of the radio hardware. The cost analyzer chooses the best decomposition and passes it to the compiler and run-time system (Section 3.4) which converts the decomposition into a binary executable and executes it. While it executes, the program and the run-time

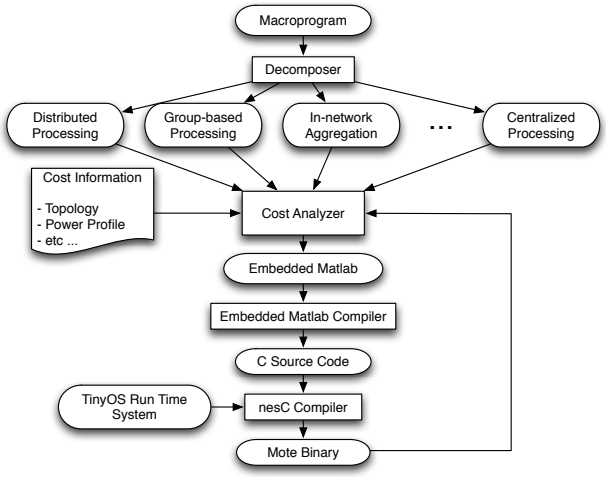


Figure 1. MacroLab consists of a decomposer, a cost analyzer, and a run-time system. In our implementation, we generate all possible decompositions of a macroprogram and then analyze and compare them based on the cost profile of a target deployment.

system continue to collect information about the cost profile of the deployment and feed this information back to the cost analyzer. If the cost profile changes or if the cost profile at compile time was incomplete or incorrect, the cost analyzer may decide to reprogram the network with a new decomposition.

3.1 Abstraction

MacroLab provides a *vector programming* abstraction similar to Matlab. A vector is a data structure containing values called *elements* that are arranged in rows and columns. For example,

$$r = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

is a 3 x 3 vector with 9 elements. Vectors can be indexed by dimension: the element in the second row and the third column of r can be selected with `r(2,3)` resulting in f. In MacroLab, like in Matlab, the ":" is used to select an entire dimension. For example, `r(3,:)` selects the 3rd element of the first dimension (rows) and the entire second dimension (columns), namely `[g h i]`. Operations such as *vector addition* and *vector multiplication* operate on the data structures. The operation `find(r==f)` produces the index of the element in r that has the value f, which is $[2, 3]$. In addition to standard vector programming, MacroLab introduces three new concepts to facilitate software development for CPSs: the macrovector, the dot-product index, and the neighborhood. These concepts are discussed in the next three subsections.

3.1.1 The Macrovector

In MacroLab, we define a new data structure called the *macrovector*. Macrovectors differ from traditional vectors in that each element of a macrovector is associated with a particular node in the network. Thus, macrovectors are *unordered* and are *indexed by node ID*. This abstraction can be useful, for example, to store sensor readings. If `light` is

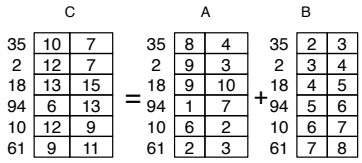


Figure 2. Two $n \times 2$ macrovectors A and B can be added and stored into a third macrovector C. The values on the left of each vector indicate which node ID the cells are associated with.

a macrovector storing the light values of each sensor, then the operation `light(5)` would retrieve the light value of the sensor node with $ID = 5$. Since sensor node IDs may be non-sequential, the elements in a macrovector do not form a strict sequence. Macrovectors can have multiple dimensions, but only a single dimension is indexed by node ID. The other dimensions are normal vectors indexed sequentially. Macrovectors can be created using the command

```
light = Macrovector(<scope>, [length], [length], ...)
```

where the `scope` of the macrovector is the set of nodes with which the elements are associated. This scope is a vector of node IDs and the length of the first dimension will be the number of IDs. The lengths of subsequent dimensions must be given for a multi-dimensional macrovector. These `length`s are simply integer values indicating the size of each dimension.

Macrovectors support many standard Matlab vector operations such as `addition`, `subtraction`, `cross-product`, `find`, `max`, and `min`. These operations can be combined to perform *macro* operations that operate on data associated with many different sensor nodes. For example, the operation

```
maxLight = max( light )
```

will return the maximum light value in the network. The operation

```
hotLight = light( find( temp > 100 ) )
```

will return the light values on nodes where the `temp` value is higher than 100. If these vectors were tables in TinyDB, this would be similar to posing the SQL query

```
SELECT light WHERE temp > 100
```

Elements associated with the same node ID are paired together for binary operations that involve multiple macrovectors. For example, Figure 2 shows the operation `C = A + B` performed on three $n \times 2$ macrovectors. In this operation, the elements of A and B corresponding to node 35, for example, are added together and stored in the elements of C corresponding to node 35.

3.1.2 Dot-product Index

We provide a new way of indexing into macrovectors called the *dot-product index*. For example, with `s = [1 2 3]` and `t = [2 1 3]`, the aforementioned vector `r` can be indexed as follows:

$$r(s,t)[1,2] == \begin{bmatrix} & b & \\ d & & \\ & & i \end{bmatrix}$$

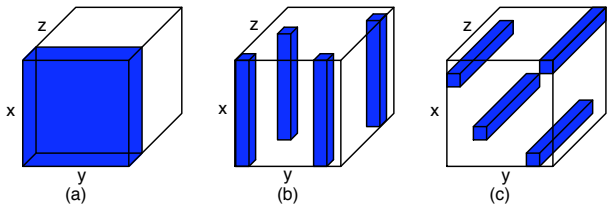


Figure 3. A three-dimensional vector D can be indexed with the (a) cross-product index `D(x,y,1)`; (b) dot-product index `D(:,y,z)[2,3]`; or (c) dot-product index `D(x,y,:)[1,2]`.

The two values in square brackets indicate that the elements of the first and second dimension indices should be matched pair-wise before values are selected from the matrix. Since s and t each contain 3 elements, this dot-product index would select 3 elements from r in total. With the values of s and t above, the dot-product index would select elements $[1,2]$, $[2,1]$, and $[3,3]$. This is different from traditional indexing in Matlab, what we might call the *cross-product index*, in which the same index vectors would produce

$$r(s,t) = \begin{bmatrix} b & a & c \\ e & d & f \\ h & g & i \end{bmatrix}$$

In other words, all values of index vector s are paired with all values of index vector t, selecting 9 values in total. Figure 3 illustrates how cross-products and dot-products can be used to select different elements in a three-dimensional vector.

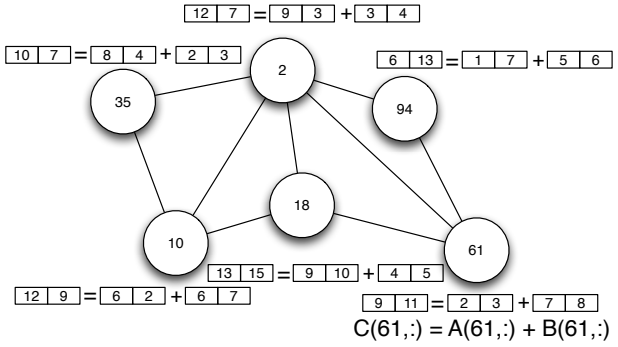
Dot-product indexing can be used to efficiently perform operations in which the element to be selected on each node is different. For example, if an $n \times 10$ macrovector `circularBuffer` stored the last 10 light values read on each node and an $n \times 1$ macrovector `lastIndex` stored the index of the most recent value stored, then the operation `circularBuffer(:,lastIndex)[1,2]` would provide the most recent value on each node. The capabilities of macrovectors and dot-product indexing will be demonstrated in Section 4.

3.1.3 Neighbor-based Representation

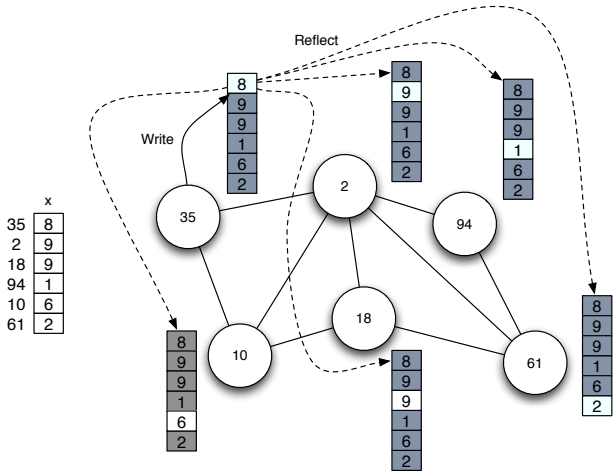
A node's *neighborhood* is the set of nodes that are within radio range. This is a very useful type of *group* in which a node is guaranteed to have cheap communication to all other nodes. Connectivity-based neighborhoods are often a critical part of efficient in-network data processing algorithms. Neighborhoods are a special type of group since each node has a different neighborhood. Because of this, we introduce new syntax for defining a neighbor-based macrovector:

```
lightReflection = neighborReflection(light)
```

which indicates that lightReflection is a vector of neighbor-based macrovectors that should store *reflections* of the `light` macrovector. When a node writes to its own element of the `light` macrovector, that value is cached in the rows corresponding to its neighbors. Thus, `lightReflection` is a two dimensional vector where each row contains cached values of a node's neighbors' light readings. Since each node may have a neighborhood of different size, this is not necessarily



(a) Distributed Macrovector: Nodes can read and write their own values in the vector.



(b) Reflected Macrovector: Nodes can read all values in the vector, but can only write to their own value.

Figure 4. Example macrovector representations. Macrovectors can be (a) distributed across nodes or (b) reflected across nodes. They can also be represented in other ways that are not shown.

a rectangular matrix; each row may be a different length. This abstraction is very similar to the Hood programming abstraction [32].

3.2 Program Decomposition

The MacroLab decomposer converts a macroprogram into a set of microprograms that can be executed on nodes. The goal is to preserve the semantics of the macroprogram while allowing for efficient distributed operation. The decomposition algorithm has two steps. First, it chooses a data *representation* for each macrovector, which can be *distributed*, *centralized*, or *reflected* (Section 3.2.1). Based on the representations chosen, it then uses rule-based translation to convert the vector operations in the macroprogram into network operations (Section 3.2.2).

3.2.1 Choosing Macrovector Representations

Macrovectors provide a uniform interface to several underlying *representations*, which are different ways that the macrovector can be stored in the network. MacroLab currently supports three representations: distributed, central-

ized, and reflected, the trade-offs of which are described in more detail below. Other representations are possible and would allow MacroLab to support different classes of distributed algorithms. Vector operations can be applied to macrovectors regardless of their representation, making them ideally suited for DSCD.

Distributed Representation: The first way to represent a macrovector is to store each row on its associated node. Figure 4(a) shows how the elements of the macrovectors A, B, and C from Figure 2 can be stored on each node and how the `addition` operation is performed. Since elements are only added to corresponding elements on the same node, this operation can take place without message passing between nodes.

In general, the distributed representation of macrovectors allows for the efficient implementation of vector operations that do not span multiple rows. Conversely, this representation requires significant message passing for aggregate operations like `max` that require values resident on multiple nodes. If a macroprogram uses the `max` operation frequently on a particular macrovector, then a distributed decomposition would be very costly.

Centralized Representation: The second representation supported by our framework stores all elements on a single node, typically the base station. This representation is in diametric opposition to a *distributed* representation. It allows operations like `max` to be applied with virtually no explicit message passing cost. However, there is a potentially significant cost associated with keeping the elements of the centralized vector up-to-date. If the values are frequently updated remotely by the sensor nodes, they need to be frequently transmitted for storage. The centralized representation is favorable if the vector participates frequently in aggregate operations that span rows (like `max`). It is less favorable if the vector is frequently updated with sensor data.

Reflected Representation: The third macrovector representation stores all elements on all nodes. The microcode on each node has read/write access to its associated element and read-only access to cached versions of all other elements in the vector. This precludes the need for write-write synchronization since only one node may write to any given element. However, nodes do need to communicate their updated values after performing a write, as illustrated in Figure 4(b); when node 35 writes to its own element in the vector, the value is *reflected* to all other nodes currently caching it.

This representation is conceptually similar to Reflective Memory (RM), which is a form of shared memory for parallel systems [13]. It is different from a neighbor-based macrovector because the scope of a reflected macrovector is *globally defined*; it is not different for each node. The reflected representation is beneficial when used by a small *group* of nodes that are relatively close to each other but comparatively far from the base station. In that situation, the cost of sending information to the base station to perform a `max` operation is higher than the cost of transmitting to all other nodes in the group. The reflected representation may also make an operation cheaper when all nodes need the result.

3.2.2 Rule-based Microcode Translation

Given a representation for each macrovector, the decomposer must produce the appropriate microcode for each operation in the macroprogram. For example, the `max` oper-

		$\mathbf{lhs = M(a_1, \ldots, a_n)}$ % **Synchronous Read**
Cnt. or Ref.	R	`owner_id = RTS.owner(cur_pc(), M);` `RTS.notify(cur_pc(), owner_id);`

	L	`node_ids = source_nodes(a1);`
		`RTS.wait(cur_pc(), node_ids);`
		`lhs = local( M(a1,...,an) );`

Distributed	R	`if (a1 contains node_id()) then`
		`  owner_id = RTS.owner(cur_pc(), M);`
		`  RTS.send(owner_id,`
		`  local(M(node_id(), a2,...,an) ))`
		`  RTS.notify(cur_pc(), owner_id);`
		`fi;`

	L	`node_ids = source_nodes(a1);`
		`RTS.wait(cur_pc(), node_ids);`
		`lhs = local( M(a1,...,an) );`

		$\mathbf{M(a_1, \ldots, a_n) = rhs}$ % **Synchronous Write**

Centralized	R	`if (a1 contains node_id()) then`
		`  owner_id = RTS.owner(cur_pc(), M);`
		`  RTS.wait(cur_pc(), owner_id);`
		`fi;`

	L	`node_ids = source_nodes(a1);`
		`local( M(a1,...,an) ) = rhs;`
		`RTS.notify(cur_pc(), node_ids);`

Reflected	R	`if (a1 contains node_id()) then`
		`  owner_id = owner(cur_pc(), M);`
		`  RTS.receive(owner_id,`
		`  local( M(a1,...,an) ));`
		`  RTS.wait(cur_pc(), owner_id);`
		`fi;`

	L	`node_ids = source_nodes(a1);`
		`local( M(a1,...,an) ) = rhs;`
		`foreach (node_id in node_ids) do`
		`  RTS.send(node_id,`
		`  local( M(a1,...,an) ));`
		`done;`
		`RTS.notify(cur_pc(), node_ids);`

Distributed	R	`if (a1 contains node_id()) then`
		`  owner_id = RTS.owner(cur_pc(), M);`
		`  RTS.receive(owner_id,`
		`  local(M(node_id(), a2,...,an)));`
		`  RTS.wait(cur_pc(), owner_id);`
		`fi;`

	L	`node_ids = source_nodes(a1);`
		`local( M(a1,...,an) ) = rhs;`
		`foreach (node_id in node_ids) do`
		`  RTS.send(node_id,`
		`  local( M(a1,...,an) ));`
		`done;`
		`RTS.notify(cur_pc(), node_ids);`

Table 1. Pseudocode for the mote-level microcode translation for common macrovector operations. For each data representation, the row marked L (for *local*) denotes the code for the mote that will perform the operation (i.e., the locus of synchronization); R (for *remote*) marks the code for all other nodes. `M` is a macrovector; `lhs` and `rhs` are normal vectors. The mote-local representation of `x` is given by `local(x)`. The `owner(PC,M)` function gives the ID of the node requesting the read or write operations on macrovector `M` at location `PC` in the macroprogram.

ator must perform different actions when operating over a distributed vector than when operating over a centralized vector. To accomplish this, MacroLab uses a library of microcode templates for each operator and the different representations of its input parameters. Thus, `max` would require three implementations and a binary operator would require 3×3 implementations. This approach of using a library of operator implementations to deal with different matrix representations has been used before [3] and most of this complexity is hidden from the user.

An implementation of a vector operation will typically consist of multiple functions, each of which is loaded onto the domain of one of the input parameters. Table 1 shows the various implementations of two basic macrovector operations: reading and writing one or more elements in a vector. In this context, the vectors `lhs` and `rhs` are normal vectors and `M` is an n-dimensional macrovector that is being accessed by indices `a1` through `an`. These indices are themselves vectors and index an entire dimension of the matrix `M`. The microcode for these operations is different for each of the three representations discussed in 3.2.1: centralized, reflected, and distributed. For each representation, the operation is divided into two functions: one for the L (*local*) nodes and one piece of code for the R (*remote*) nodes.

For illustrative purposes, Table 1 shows the microcode for the synchronous `read` and `write` operations based on the standard `notify` and `wait` primitives instead of using message passing primitives. `RTS.wait(PC, node_ids)` causes the current node to block until each of the nodes in `node_ids` has called `RTS.notify` with a matching value for PC. A call to `RTS.notify(PC, node_ids)` signifies that the caller has "caught up" to a particular program point in the macroprogram. `RTS.notify` blocks until the corresponding `RTS.wait`. This is done to prevent the notifying node from overwriting data before it can be used by the waiting thread. `RTS.owner` takes the current location in the macroprogram (`cur_pc()`) and returns the node ID of the local node. This ID may be fixed at compile time. Operations over a centralized macrovector are always performed on the node that has the local copy of that macrovector. In other cases, such as neighborhood-based operations, the ID of the local node is unknown until runtime and may change over time.

In principle, this implementation of synchronous `read` and `write` could be used to implement most macrovector operations. Here we show the "local" half of naïve implementations of the synchronous `max` and `find` functions:

```
1 % smax(A)
2 sread(A, lvar);
3 lmax = max(lvar);
4 write(A, lmax);
5 .
```

```
1 % sfind(A(1) = 5)
2 sread(A, lvar);
3 lres =
4   find(lvar(1) = 5);
5 write(Temp, lres);
```

The synchronous `max` operation works by reading `A` into a local variable (which is a normal vector), performing the operation, and then writing the results. Thus, this implementation caches the entire macrovector on a centralized node before performing the operation. The code for synchronous `find` assumes the existence of a temporary vector `Temp` created by the translator to store the result of the find in single-column form. Both operations are synchronized because the initial read is synchronized, although this naïve approach incurs a round-trip message between the local node and each remote node.

In practice, many macro-operations will require specialized implementations to further reduce messaging overhead. For example, the `max` operation could be performed using in-network aggregation such as that used in Tiny AGgregation (TAG) [19]. This implementation is a *semantics-preserving* optimization. It is computationally equivalent to the naïve implementation shown above, but would result in lower messaging overhead. In the following section, we discuss optimizations to reduce messaging overhead that are not semantics-preserving.

3.2.3 Reducing Synchronization Overhead

Synchronization is one of the main costs of a MacroLab program; nodes must send messages to indicate that they have "caught up" to a point in the macroprogram, even if they have no useful data to provide. This messaging overhead is necessary to preserve the semantics of the original macroprogram, but many CPS applications do not need strict synchronization for proper operation. Consider the example below, where the intent of this code is to provide a frequently updated maximum light value.

```
1 every(1000) {
2   light = sense(lightSensors);
3   maxLight = max(light);
4 }
```

In this case, synchronizing line 3 is unnecessary since the user is probably not explicitly interested in having `maxLight` represent the maximum for a particular loop iteration. Similarly, the synchronized version of the operation `find(light > 100)` would require a round trip message from all nodes, including those that have `light` values less than 100. An unsynchronized implementation could require messages only from those nodes that have values greater than 100. These optimizations improve parallelism and reduce messaging overhead, but do not preserve the original semantics of the vector operations. To allow users to employ these optimizations, MacroLab must provide both synchronized and unsynchronized versions of each operation. We adopt the convention that the synchronized versions take function names that start with an *s*: `smax`, `swrite`, `splus`, etc. The usual notations (e.g., `A + B`, `max(A)`) will refer to the *un*synchronized version of that operation.

It is important to note that not all vector operations require synchronization. Operations over macrovectors generally fall into two categories: *row-parallel* operations and *inter-row* operations. An inter-row operation is any expression that "mixes" macrovector domains. For example, an expression like `max(A)` returns a single-valued result by combining information from all rows of `A`. Inter-row operations require synchronization to be semantics-preserving, but alternative implementations with different semantics can be used in order to reduce overhead. Conversely, a statement like `A = A + B` is *row-parallel*: the operation over any particular row does not require information from any other row, and so the operation can be performed over each row without waiting for A and B to be fully updated. When multiple row-parallel operations occur consecutively, their execution may overlap.

Generally, inter-row operations are more expensive than row-parallel operations because of the synchronization required. The difference between row-parallel and inter-row operations is evident from the source code, providing the user with a *clear cost model* of the macroprogram and empowering the user to write optimized code that will produce efficient decompositions. Furthermore, the user can con-

trol the amount of synchronization overhead by choosing between synchronized and unsynchronized versions of each inter-row operation.

3.3 Cost Analyzer

The decomposition process produces many feasible candidate program decompositions. The goal of cost analysis is to predict which candidate will be most efficient for a target deployment. We use two different techniques for cost analysis: compile-time analysis and run-time analysis. As shown in Figure 1, an initial decomposition might be chosen using the static analysis until richer run-time profile information is available. Cost analyses can be based on a number of different cost functions such as power, bandwidth, messages, or latency. In this paper, we consider only messaging costs.

3.3.1 Static Cost Analysis

Our static analysis approximates the true messaging cost of a MacroLab program based on (1) user-provided cost information for messages, (2) sensing event frequencies, (3) a description of the deployment network, and (4) a conservative analysis of the source code to locate network sends. The cost information is provided as a matrix indicating hop counts between nodes. The high-level structure of the static cost anlayzer is as follows:

```
1  totalCost = 0;
2  foreach node x in the network do
3    cost[x] = 0;
4    foreach predicted send in one run of x do
5      cost[x] += hop_count[x,target_of_send] *
6        frequency_of_this_event_at_this_node;
7    done;
8    totalCost += cost[x];
9  done;
10 return totalCost;
```

We assume that the MacroLab program follows a main event-loop format (as in lines 4–7 of Figure 6, lines 6–19 of Figure 7, or lines 10–17 of Figure 9) and predict the cost for one run through that loop. We use an intraprocedural dataflow analysis to scan the statements in the program's main loop for predicted sends (line 4).

We consider each statement that contains a macrovector operation, such as a read from a sensor array. Based on the decomposition, the operation is analyzed to determine if it involves remote or distributed operations and thus, message sends. If we cannot statically determine the destination of the message, we conservatively use the maximal hop cost from the current node. The hop cost is weighted by the relative frequency of that event.

Note that it is not possible for a distributed operation to trigger other distributed operations in MacroLab; the user cannot write messaging code directly and all sends and receives are inserted by the decomposer and mediated by the run-time system. There are no message loops to consider and it suffices to consider each node separately to calculate a total predicted cost.

If the user does not have a model of the sensing event frequencies at each node, our analysis assumes events will occur with equal frequency across all nodes for all decompositions. A final cost estimate can be produced that is relative to the number of sensing events. In such a scenario, our cost analyzer does not predict the actual messaging cost of a decomposition but still provides a useful heuristic for distinguishing *between* two candidate decompositions.

getID() *returns the node's ID*
getProperty('property') *returns a generic property of a node*
getNodes('group') *returns the current membership in a global group*
getTime() *returns the current global time*
getNeighbors() *returns the current radio neighbors of a node*
remoteFeval(nodeIDs,funcName,{P1,P2,...,Pn}) *remote function invocation*

Table 2. The RTS must provide an interface for neighbor discovery, time sync, and remote function calls.

Note that we could use a more precise dataflow analysis for predicting statement frequencies (e.g., [25]). However, we would still need to model message costs, understand the network topology, and predict event frequencies. In our experiments these were the determining factors and the MacroLab programs themselves were small and easy to analyze.

3.3.2 Run-time Cost Analysis

We can also perform a run-time analysis to measure the costs of a deployed decomposition. We use the decomposer to inject logging code at appropriate locations in the microprogram to count the messages that are actually being sent. We can also inject logging code to estimate how many messages *would be* sent by other decompositions (as in Table 1). The logged information is periodically analyzed by the cost analyzer to ensure that an efficient version of the implementation is executing. If the currently-executing version is more costly than an alternative, the network can be reprogrammed with the alternative decomposition. Currently, we do not have the capability to reprogram the network without losing the state of the program that was generated by a different decomposition, but this will be analyzed in future work.

3.4 Compilation and Run-Time System

The run-time system supports three operations for MacroLab microprograms: (1) networking, (2) hardware access, and (3) accessor functions to information such as node ID, current time, location, radio neighbors, etc. The RTS is written as a nesC module and supports the functions shown in Table 2. The first three functions are provided directly by the RTS while the last three functions must interface with TinyOS libraries for capabilities such as time synchronization and networking operations. By interfacing with TinyOS, MacroLab leverages an existing suite of distributed algorithms as well as ongoing advances and future software development.

The `remoteFeval` function is a generic messaging interface provided to the MacroLab microprograms. It is similar to Matlab's `feval` function in that it takes a function handle and a set of arguments and invokes that function with those arguments. However, `remoteFeval` invokes the function on a set of remote nodes indicated by the `nodeIDs` parameter. The function name and arguments are marshalled into a packet, sent to those nodes, and unmarshalled before the function is invoked. The `remoteFeval` function can take an arbitrary set of nodeIDs and the RTS decides the best

```
1 module LightSensorP {
2   provides interface LightSensor;
3   uses interface Read<uint16_t> as Read;
4 }
5 implementation
6 {
7   command void LightSensor.sense() {
8     call Read.read();
9   }
10  event void Read.readDone(error_t err,
11   uint16_t val) {
12    if(err == SUCCESS) {
13      #CALLBACK(val);
14    }
15  }
16 }
```

Figure 5. A split-phase hardware driver for reading from the light sensor. The `LightSensor.sense` function is called by the microcode through the RTS. The decomposer must automatically replace `#CALLBACK` with an appropriate function to continue microprogram execution.

way to send the message. For example, if nodeIDs only contains the ID of the base station node, the RTS sends the message using a standard TinyOS routing protocol. If nodeIDs only contains IDs of neighboring nodes, the RTS sends the message using a local broadcast. In our current implementation, the RTS will flood the message to all nodes for any other set of nodeIDs, but multi-cast algorithms or other routing algorithms could easily be inserted as they are developed.

Access to hardware such as sensors and actuators must be done through new functions provided by user defined hardware drivers. The `BASE_DISPLAY` and `CAMERAFOCUS` functions shown in Figures 6 and 7 would simply be C functions provided by the user that would set the input/output pins of the microcontroller based on the input parameters. In the case of split-phase functions like `sense`, the driver must declare the name of the callback that will be triggered when the function is complete. The decomposer then generates the appropriate code using this callback to continue execution in the microprogram. The actual light sensor driver used by the code in Figure 6 is shown in Figure 5.

In order to conserve power, MacroLab enables the TinyOS 2.x low-power listening capabilities [23], which automatically duty cycles the radio and sends messages with long preambles. The RTS dynamically sets the sleep interval of the radio based on the number of messages currently being sent by the application. The entire network starts with a default sleep interval of 100 milliseconds and nodes will flood the network to halve or double the sleep interval when total transmission time is greater than 80 percent or less than 20 percent. Thus, the radio sleep interval for the entire network is dynamically set based on the node with the highest load. This simple algorithm will not always be optimal, but it works well for our existing applications, as shown in Section 4.2, and we will explore more sophisticated adaptive algorithms in future work. We currently execute MacroLab programs on Telos [24] nodes, for which the TinyOS libraries automatically use low-power mode when idle.

The MacroLab microprogram, the RTS, and the TinyOS libraries are compiled together into a single binary executable (Figure 1) that can run on mote-class devices such as the MICA [4] and the Telos. The microprogram generated by the decomposer is written in Embedded Matlab, a simplified form of the Matlab syntax that does not support dynamic typing or dynamic memory allocation. This is compiled down to C code by the Embedded Matlab compiler, provided by The MathWorks [1]. This C code is then compiled together with the nesC RTS module and the TinyOS libraries by the nesC compiler.

4. EVALUATION

We evaluate MacroLab in four parts. First, we evaluate the programming abstraction by showing that it is expressive enough to implement canonical CPS applications, such as data collection and object tracking. Second, we measure the overhead of running MacroLab programs as compared with similar programs written using nesC and TinyOS. Third, we deploy one MacroLab application in multiple different scenarios and measure the effect of DSCD on message cost. Finally, we evaluate the accuracy of the static cost analyzer.

4.1 Expressiveness of the Abstraction

We evaluate the expressive power of our programming abstraction by showing that it can be used concisely to implement two canonical CPS applications: *tree-based data collection* in Surge [8] and *object tracking* in the Pursuer-Evader Game (PEG) [28]. We selected these two applications because they represent basic algorithms that have been incorporated into many other CPS applications. Table 3 presents a comparison of the number of lines of code necessary to implement Surge and PEG in MacroLab as compared to their original implementations in nesC/TinyOS. The number of lines of code for the nesC/TinyOS implementations of Surge and PEG were cited from previous publications [14, 21]. The MacroLab applications are about one-fiftieth the size of equivalent nesC/TinyOS implementation in terms of lines of code. This ratio is typical of the difference in size between equivalent Matlab and C programs. MacroLab builds a considerable amount of logic such as routing algorithms and vector operations into the run-time system. We thus claim that MacroLab dramatically reduces the amount of code and therefore the amount of development and maintenance time required to implement basic CPS applications.

The MacroLab programming abstraction is suitable for many application domains, but it cannot express algorithms that require explicit message passing. For example, it cannot easily be used to implement routing protocols, time synchronization protocols, or other distributed middleware libraries. Instead, all of these operations must be incorporated into the run-time system. Thus, MacroLab programs are limited to distributed operations that can be neatly stored in a library and provided by some interface. We claim that this it not a restricting requirement for most CPS software. However, it can make it difficult to provide application-specific distributed operations. Developers must encapsulate such functionality by extending the run-time system. Code movement in systems like Agilla [7] and EnviroSuite [17] might be difficult to implement in MacroLab.

4.1.1 Surge

Surge is a simple application that periodically collects sensor readings from all nodes and routes them back to a base

	MacroLab	nesC/TinyOS
Surge	7	400
PEG	19	780

Table 3. Comparison of lines of code required for equivalent functionality in two basic CPS applications.

station. Figure 6 shows the source code for Surge written in MacroLab. In line 1, the run-time system is instantiated and in lines 2 and 3, it is used to instantiate a vector of light sensors (one for each node in the network) and a macrovector to hold the light values. Line 4 is the beginning of a loop that occurs every 1000 milliseconds. The light sensors are read in line 5 and the values are displayed at the base station in line 6.

There is hidden complexity in the interaction between lines 5 and 6. The decomposer identifies that the sensor resources are on the nodes while the `BASE_DISPLAY` function is only available on the base station. It therefore infers that the information created in line 5 must be routed across machine boundaries in order to be used in line 6 and the compiled version of the code automatically invokes the routing algorithm via the `remoteFeval` interface described in Section 3.4. The high-level algorithm can be expressed in seven lines of MacroLab. The automatically generated microcode that runs on the nodes is closer in size to the nesC/TinyOS implementation.

```
1 RTS = RunTimeSystem();
2 lSensors = SensorVector('lightSensor','uint16');
3 lightValues = Macrovector('uint16');
4 every(1000)
5   lightValues =  sense(lSensors);
6   BASE_DISPLAY(lightValues);
7 end
```

Figure 6. A data collection application (Surge) in MacroLab. `BASE_DISPLAY` is implemented within the RTS and sends a message to a base station for display.

4.1.2 Pursuer-Evader Game (PEG)

The Pursuer-Evader Game (PEG) is a distributed CPS application that detects and reports the position of a moving object within a sensor field. Figure 7 shows an implementation of PEG in MacroLab in which the network routes the location of the evader to a camera which visually follows the evader. In line 1, the run-time system is instantiated and in lines 2 and 3, it is used to instantiate a vector of magnetometer sensors, one for each node in the network. In line 5, a `neighborReflection` macrovector is created that automatically reflects and stores the values of each node's neighbors' `magVal` elements if they are within the threshold. In the main loop, the sensors are sampled and each node's neighbors' magnetometer readings are checked against a threshold value to see if there are more than two neighboring nodes that sense the evader. If so, a leader is chosen from amongst them by finding the node with the highest of all magnetometer readings. Its ID is then passed to the `CAMERAFOCUS` function, which focuses the camera on the location of that node. The purpose of this example is to demonstrate that MacroLab can concisely represent efficient, neighborhood-based, in-network processing. We do not elaborate on how to focus the camera on the location of the leader node.

```
1 RTS = RunTimeSystem();
2 magSensors = SensorVector('MagSensor', 'uint16');
3 magVals = Macrovector('uint16');
4 THRESH = uint8(3);
5 nRefl = neighborReflection(magVals,magVals>THRESH)
6 every(1000)
7   magVals =  sense(magSensors);
8   nodes = find(nRefl > THRESH);
9   rowSum = sum(~isnan(nRefl(nodes,:)),2)
10  toCheck = find(rowSum>2);
11  if(toCheck)
12    maxID = find(magVals(nodes(toCheck)) ...
13                 == max(nRefl(nodes,:),[],2));
14  end
15  leaderID = nodes(toCheck(maxID));
16  if(leaderID)
17    CAMERAFOCUS(leaderID);
18  end
19 end
```

Figure 7. A tracking application (PEG) in MacroLab. `CAMERAFOCUS` is implemented within the RTS and sends a message to the camera, which is at the base station.

4.2 Performance Overhead

To evaluate the performance overhead of MacroLab, we measured the resource consumption of the two applications described in Section 4.1 and compared it against existing applications written in nesC/TinyOS. We found that MacroLab programs are very similar to TinyOS programs in terms of memory footprint, execution speed, and power consumption.

The program size and heap size of two MacroLab programs is shown in Table 4, along with those of several existing TinyOS programs. The MacroLab programs actually have a smaller memory footprint than their corresponding TinyOS implementations, both in terms of program memory and RAM. Table 5 shows the breakdown of our MacroLab programs' memory footprints. This information was collected by examining the symbol table of the final binary executables. Therefore any variables or code removed by compiler optimizations are not included. The vast majority of the program size of MacroLab applications (approximately 17KB of ROM and 600 bytes of RAM) is due to imported TinyOS libraries for multi-hop routing, access to the ADC, and the Timer. The run-time system module requires 558 bytes of program memory and 66 bytes of RAM. For both PEG and Surge, the application logic itelf required less than 1.3KB of ROM and 150 bytes of RAM.

We also compared the maximum run-time stack sizes of the MacroLab and nesC Surge implementations. We filled memory with a special reference pattern, ran the program for 600 seconds, and then computed the high water mark for stack growth by looking for the last byte not containing the reference pattern. MacroLab's RTS layer introduces additional functions and could therefore potentially require more memory for the stack. However, aggressive function inlining by the nesC compiler causes the function call depth to be almost the same for both applications. The TinyOS version requires 120 bytes while the MacroLab version requires 124 bytes for the stack, as shown in the last column of Table 6.

To compare the execution speed of a MacroLab program with a TinyOS program, we measured the time to execute Surge from the beginning of the loop when the node reads the sensor until the radio has accepted the message for trans-

Application	Program Size	Heap Size
TelosB	49,152	10,240
MICAz	131,072	4,096
Blink	2,472	38
CountRadio	11,266	351
Oscilloscope	9,034	335
OscilloscopeRF	14,536	449
SenseToRfm	14,248	403
TOSBase	10,328	1,827
MacroLab_Surge	19,374	669
SurgeTelos	24,790	911
MacroLab_PEG	18,536	770
PEG [28]	61,440	3,072

Table 4. Program and heap size comparison for common TinyOS applications and two MacroLab applications for TelosB nodes.

Application	TinyOS	RTS	MacroLab
MacroLab_PEG	16714/579	558/66	1264/125
MacroLab_Surge	15714/579	558/66	1144/24

Table 5. A breakdown of the amount of flash/RAM in the TinyOS libraries, RTS, and program logic of MacroLab applications

mission. These times do not include the MAC and transmission delays. This time was measured for both the MacroLab and TinyOS implementations using an oscilloscope. The first two columns of Table 6 show that the total execution time was about 18 milliseconds for both programs, but the MacroLab program takes about 3 percent longer (0.5 milliseconds). The CPU was idle for most of the execution, waiting for the ADC to return a value. The non-idle time for the MacroLab program was 705 microseconds compared with 361 microseconds for the TinyOS program. Thus, the MacroLab program requires almost twice as many instructions to be executed as the TinyOS program. This is a large multiple, but the effect on overall execution time and power consumption is small.

The power consumption of both implementations of the Surge application is shown in Figure 8. In both cases, power consumption was measured using an oscilloscope on a node that was forwarding messages from exactly two children. The period with which Surge sampled the sensor and forwarded the message to the base station was varied from 100 milliseconds to 10 seconds. The results show that the average power consumption over a sample run of 100 seconds is nearly identical for both implementations, even as the sampling frequency changes by over two orders of magnitude. This evidence suggests that MacroLab programs can match TinyOS programs in terms of power efficiency.

4.3 Effect of DSCD on Performance

To evaluate the effect of DSCD on the performance of an application in multiple scenarios, we implement a *bus tracking* application (Figure 9). In order to easily modify the deployment scenario, we performed this part of the evaluation in simulation. In the bus tracking application, each bus stop records arrival times for the buses and computes estimated arrival times for all other buses. The application logic is shown in Figure 9, which maintains state about the last time a bus was seen at every stop, the time it takes to travel from each stop to every other stop, and the estimated time that each bus will next arrive at every stop.

Application	Execution	CPU	Stack
Surge	17.7msec	361usec	120bytes
MacroLab_Surge	18.2msec	705usec	124bytes

Table 6. An evaluation of the execution time of the application, logic (CPU), and maximum consumed stack memory.

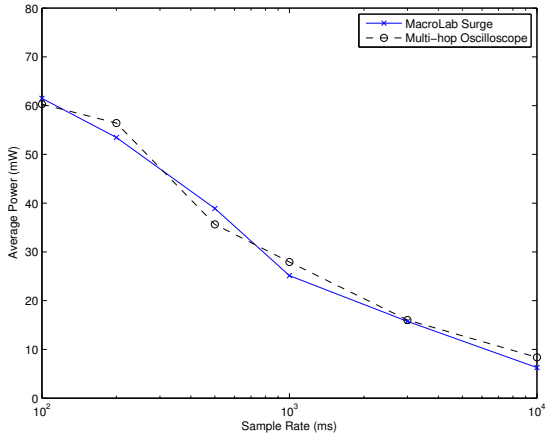


Figure 8. Oscilloscope power measurements of MacroLab and nesC Surge implementations.

First, we sense for a bus at each stop and collect a time stamp of the bus arrival as `busTime`. The *arrivals* matrix stores the last time before now that each bus arrived at each stop, so we update *travelTime* to be *busTime* minus *arrivals*. In other words, we estimate the travel time between stops to be the current time that each bus arrived less the last time that bus was seen at every other stop. *Arrivals* is then updated with the current arrival time of the bus. Next, the *estimates* vector is updated to be *travelTime* plus *busTime*. We estimate the predicted arrival time for each bus to be its travel time plus the last time it was seen at all stops. Finally, the estimated arrival times are displayed to potential passengers using the `BASE_DISPLAY` operation. These main matrix operations in this application make heavy use of the dot product notation described in Section 3.1.2 for conciseness and efficiency.

MacroLab's row-level parallelism allows the matrix operations to occur in parallel. In this particular application, the program runs correctly without using the synchronized implementation of any inter-row operations, and so the assignment in line 15 does not block until all values are collected. Each row can be processed in parallel as buses arrive at different stops.

We evaluate this application in four scenarios: 1) the estimated bus arrival times are displayed to the passengers on a website which is updated from a centralized base station, 2) the estimated bus arrival times are displayed to the passengers at each bus stop, 3) the base station radio range is increased substantially for better coverage, and 4) an additional bus route is added by the bus company. Our results show no decomposition is best for all target deployments

```
RTS = RunTimeSystem();
busstops = RTS.getNodes('stopnode');
buses = RTS.getNodes('bus');
estimates = Macrovector(busstops, length(buses) ,'uint16');
arrivals = Macrovector(busstops, length(buses) ,'uint16'));
travelTime = Macrovector(busstops, length(busstops), length(buses) ,'uint16'));
busSensors = SensorVector('BusSensor',busstops,'uint16');
routes = uint8({[1 2 3 4], [ 5 6 7 8]}); %Example routes

while(1)
  [busID,r] =  sense(busSensors);
  busTime = RTS.getTime();
  travelTime(routes{r},routes{r},busID)[1,3] = busTime - arrivals(routes{r}, busID);
  arrivals(routes{r},busID)[1,2] = busTime;
  estimates(routes{r},busID) = travelTime(routes{r},routes{r},busID)[2,3] + busTime;
  BASE_DISPLAY(estimates(routes{r},:));
end
```

Figure 9. MacroLab code for the bus tracking application.

and choosing the correct decomposition can reduce messaging costs by an average of 44 percent.

MacroLab can optimize for a number of cost metrics (such as latency, power consumption, or message cost) by using a cost profile and a cost analyzer that analyzes the cost of each program decomposition for a target deployment scenario. In this evaluation, we only optimize the total number of messages that must be sent by the network to achieve the global objective.

4.3.1 Experimental Setup

Our test scenario consists of actual bus routes at the University of Virginia, provided by the University Transit Service (UTS), shown in Figure 10. We assume that each bus stop has a mote and can sense the bus currently at its stop. The cost profile of the test scenario was created based on the actual locations of the bus stops and assuming a reliable communication range of 500 meters

In order to test this wide variety of deployment scenarios, we built a Matlab-based simulator for this part of the evaluation. The simulation environment for MacroLab only requires a slight modification to the RTS in order to support function calls into the simulator instead of directly to

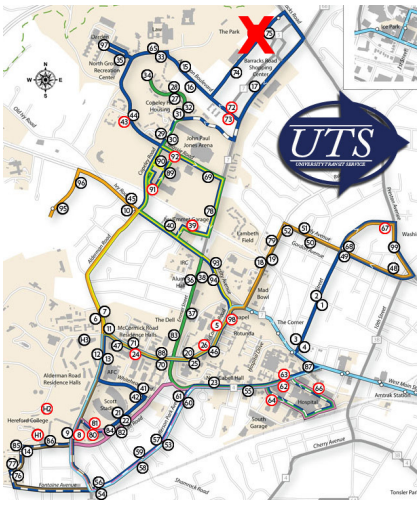


Figure 10. UTS Bus Routes at U.Va. The base station is indicated as a cross at the top of the figure.

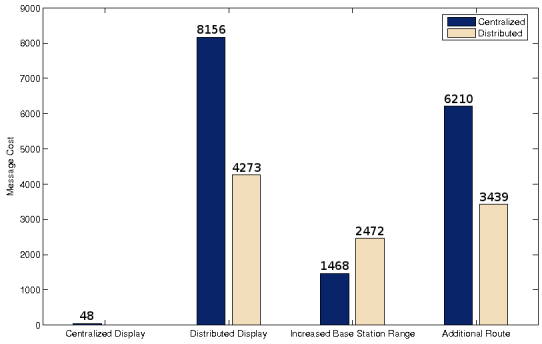


Figure 11. Neither decomposition is best for all deployment scenarios. Small changes in the deployment scenario changes the optimal implementation between centralized and distributed.

the nodes. The simulator runs code similar to microcode that would run on a mote-based RTS. In this experiment, we only evaluate two decompositions: a centralized and distributed decomposition.

Communication between nodes is provided by the RTS. Based on the cost profile of the network, the simulation observes the total message cost for each decomposition and scenario. The cost matrix is computed from the topology of our scenario and encodes the cost to send messages between pairs of nodes. The RTS currently supports point-to-point routing but more routing algorithms can be added as they are developed.

4.3.2 Scenario 1: Website Display

In the first scenario, we record and display the bus information on a website using a centralized base station. This is accomplished by using the `BASE_DISPLAY` operation which can only be executed in a centralized fashion. In order to accomplish this version of the application, the nodes will sense buses and send their data directly back to the base station which will maintain state in all of the macrovectors and perform all of the vector computations. The messaging cost of this application is the same as the Surge application. Only one bar is shown in Figure 11 because the `BASE_DISPLAY`

operator is only defined for centralized decompositions. By running this decomposition in simulation, we found the cost of running the application using this scenario to be 48 messages over a 30 minute simulation.

4.3.3 Scenario 2: Bus Stop Displays

Changing `BASE_DISPLAY` to `MOTE_DISPLAY` allows us to show the data at each bus stop rather than at the base station. Sending and receiving bus arrival information between all nodes within each route totals 8156 messages for the centralized implementation and 4734 for the distributed version; the distributed decomposition is almost twice as efficient. The large increase in the number of messages is due to the fact that each time a bus arrives at a stop, its arrival time must be transmitted to all the stops in the route. In the centralized implementation, this requires $O(n^2)$ messages per route, where n is the number of stops on that route. This cost increases dramatically for routes that are far from the base station. All nodes must send each bus arrival event all the way to the base station which must then forward the message back to each node on the route individually.

The cost of the distributed implementation does not incur this overhead. Each node forwards bus arrival events to all other nodes on its route; it does not need to transmit back and forth to the base station. If a stop is on two or more routes, it forwards the message to all other stops on all such routes. In the compiled code, this is achieved by distributing all vectors in the program and making the `arrivals` vector a reflected vector across all nodes on a route. For this target deployment scenario, the messaging cost of the distributed decomposition is about 50 percent lower than the centralized decomposition.

The only change to the bus tracking application in this scenario versus Scenario 1 is that the `BASE_DISPLAY` function was changed to `MOTE_DISPAY`, changing the library function from a centralized operator to a distributed operator. It should be noted that if this were a TinyOS application, we would be required to implement a completely new version of the program in order to make this change. Thus, small changes in the program can result in large changes to the cost profile of each decomposition. In MacroLab, the single line addition is handled by the decomposer and the RTS in order to choose the optimal solution.

4.3.4 Scenario 3: Increased Base Station Range

In the third deployment scenario, a high-gain antenna is added to the base-station which increases the coverage of the base station and changes the cost profile of the network. Figure 11 shows that adding the antenna reduces the cost of messages dramatically. This cost reduction is because the increased range allows all of the nodes to more cheaply communicate with the base station. However, this change affects the centralized decomposition to a greater extent. This is because all nodes in the network use the base station link in the centralized implementation while in the distributed implementation, only the nodes that can opportunistically route through the base station to reduce messaging costs to other nodes on the route will actually do so. Increasing the base station range increases the number of nodes that opportunistically route through the base station, reducing messaging costs but not as dramatically as in a centralized decomposition. Figure 12 shows that for more then a 1500 meter range, the centralized implementation gives superior performance.

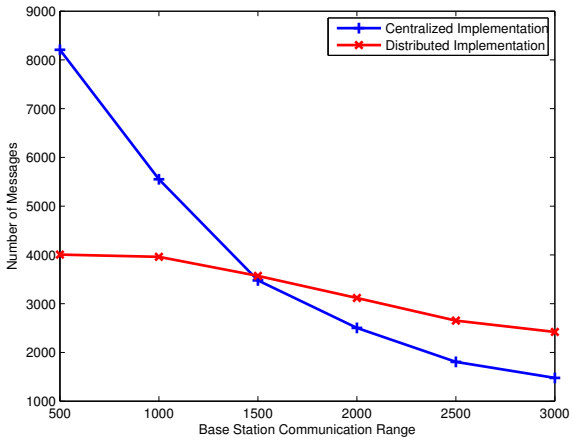


Figure 12. Changing the base station range changes the balance between the two decompositions.

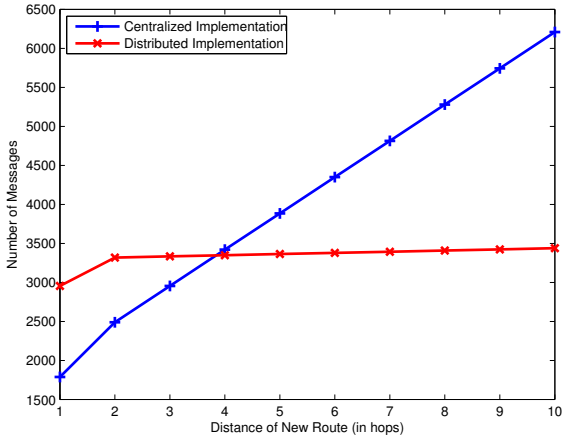


Figure 13. Adding a new route at various distances changes the balance between the two decompositions.

In this scenario, a centralized decomposition costs 1468 messages while the distributed version costs 2472 messages. This is a reversal of the cost trade-off in the previous scenario. We show that small changes to radio hardware can lead to large changes in the cost profile of the target deployment. Redesigning a TinyOS program to be efficient given this new cost profile would be difficult, but the MacroLab framework makes this change automatically.

4.3.5 Scenario 4: Additional Route

In the fourth deployment scenario we add a route that runs from the main campus to a new location 2500 meters away from the base station. Figure 13 shows the distributed and centralized costs as a function of the number of hops from the base station. As the number of hops increases, the centralized decomposition must send messages farther to reach the base station, while the cost of the distributed

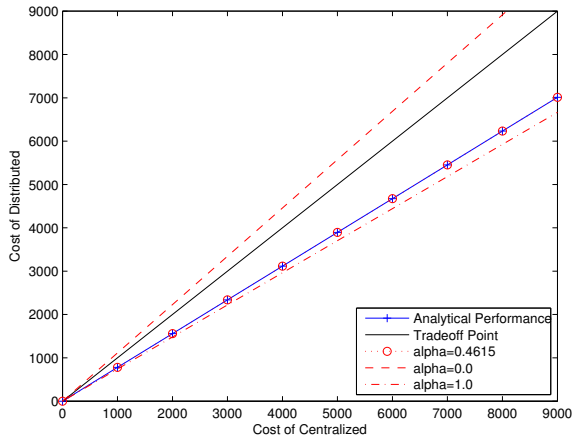


Figure 14. Estimated and measured messaging costs. The parameter α is the ratio of buses on long routes versus those on all other routes.

decomposition does not change. Once this messaging cost to and from the base station exceeds the cost of sending messages directly to the other nodes in the route, it is more expensive to utilize the centralized implementation.

Figure 11 shows that at an additional 4000 meters, the distributed version becomes more efficient than the centralized decomposition. The centralized cost is 6210 and the distributed cost is 3439. Because the new route is farther away from the base station, communication with the base station becomes more expensive. In this scenario, we demonstrate that small changes to the network topology can cause large changes to the cost profile of the target deployment.

4.4 Accuracy of Static Cost Analyzer

Our static cost analysis (Section 3.3.1) can make use of information about expected event frequency. For simple event-loop MacroLab programs, the presence and quality of this event frequency information is the determining factor in the accuracy of the analysis. In the worst case, such information is not available. For the bus tracking application, lacking any other information, our static analysis assumes that each bus stop observes a bus event with the same frequency. However, this is not necessarily true and there will be some discrepancy between the cost estimated by the static analysis and the actual cost of the decompositions. For example, if a larger percentage of buses appear on very long routes, this will increase the cost of the distributed decomposition relative to the static estimate. On the other hand, if many buses appear on routes far from the base station, this would increase the cost of the centralized decomposition.

Figure 14 shows static estimated costs as well as dynamic measured costs as the distribution of buses and therefore sensed events within the network changes. To highlight the differences between the centralized and distributed implementations, we focus on the longest routes. The parameter α is defined as the ratio of buses on the long routes versus those on all other routes. The crossover point for centralized and distributed message costs is represented by the $y = x$ line. Ratios above the crossover line will be optimally run as centralized applications, those below as distributed. By computing the ratios for various scenarios, we can see how much each change will affect the total cost of the application and whether it will cause a switch in the optimal implementation. The bounds of our application are plotted by the $\alpha = 0$ and $\alpha = 1$ lines, in which either all buses or no buses appear on the longest routes. Our static cost analysis algorithm is quite accurate for this application and is not extremely sensitive to the assumptions made about the frequency of events; it will only be incorrect with extremely non-uniform event distributions. In such cases, the run-time system can dynamically monitor changes or errors in the cost profile of the target deployment. As shown in Figure 1, such dynamic cost information can be fed back into the cost analyzer which can decide to reprogram the network.

5. CONCLUSIONS

MacroLab's approach of *deployment-specific code decomposition* addresses a central question in the area of Cyber-Physical Systems software design: should programs be implemented in a centralized or distributed fashion? Most early sensor network research focused on "*localized* algorithms – where sensors only interact with other sensors in a restricted vicinity, but nevertheless collectively achieve a desired global objective" [6]. For example, early object tracking applications argue that neighborhood communication and local processing are necessary to efficiently filter false positives [32] and services like TAG use *in-network aggregation* to calculate network statistics *en route* to decrease message passing [18]. More recently, several architectures have proposed the use of centralized algorithms to control distributed systems. *Marionette* allows the user to write a centralized Python script to control all nodes in a network [33]. It argues that centralized algorithms are easier to write and debug and that, once debugged, functionality can be *migrated* to the sensor nodes for efficiency reasons if necessary [31]. The *Tenet* [9] architecture takes a stronger stance by arguing that all application-specific code should *always* execute on master devices while sensor nodes should be restricted to a small set of predefined operations on locally-generated data. The rationale here is to separate the application code from the networking code so that changes in the application do not cause cascading changes to the networking middleware.

In this paper, we argue that programs can actually be implemented in both a centralized *and* a distributed fashion. We re-frame the architectural question from *where code should execute* to *how code should be written*. The central tenet of our architecture is that all application-specific logic should be contained in a macroprogram and all distributed operations must be contained as libraries in the run-time system. When code is written in this way, we get the best of both worlds: (1) the decomposer and the run-time system can choose the manner of implementation that provides the best performance in terms of cost metrics like bandwidth, power, and latency, and (2) the user is free to write *deployment-independent* programs that are simple, robust, and easy to understand. In future work, we will use new macrovector representations to decompose programs into many points on the spectrum between purely centralized or purely distributed code, such as hierarchical or group-based data processing, and in-network aggregation.

6. REFERENCES

[1] The mathworks. http://www.mathworks.com/.

[2] J. Bachrach and J. Beal. Programming a Sensor Network as an Amorphous Medium. Technical report, MIT '06.

[3] R.-G. Chang, T.-R. Chuang, and J.-K. Lee. Compiler optimizations for parallel sparse programs with array intrinsics of fortran 90. In *ICPP '99*.

[4] Crossbow. Micaz datasheet. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf.

[5] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. V. Eicken, and K. Yelick. Parallel programming in Split-C. In *Supercomputing '93*.

[6] D. Estrin, R. Govindan, J. S. Heidemann, and S. Kumar. Next century challenges: Scalable coordination in sensor networks. In *MobiCom '99*.

[7] C.-L. Fok, G. C. Roman, and C. Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *ICDCS '05*.

[8] 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 '03*.

[9] O. Gnawali, K.-Y. Jang, J. Paek, M. Vieira, R. Govindan, B. Greenstein, A. Joki, D. Estrin, and E. Kohler. The tenet architecture for tiered sensor networks. In *SenSys '06*.

[10] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using Kairos. In *DCOSS '05*.

[11] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C.-W. Tseng. An overview of the Fortran D programming system. In *LCPC '91*.

[12] G. C. Hunt and M. L. Scott. The Coign automatic distributed partitioning system. In *OSDI '99*.

[13] M. Jovanovic and V. Milutinovic. An overview of reflective memory systems. *IEEE Concurrency '99*.

[14] N. Kothari, R. Gummadi, T. D. Millstein, and R. Govindan. Reliable and efficient programming abstractions for wireless sensor networks. In *PLDI '07*.

[15] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al. TinyOS: An Operating System for Sensor Networks.

[16] 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 '05*.

[17] L. Luo, T. Abdelzaher, T. He, and J. Stankovic. EnviroSuite: An environmentally immersive programming framework for sensor networks. *TECS '06*.

[18] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst. '05*.

[19] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *OSDI*, December 2002.

[20] G. Mainland, M. Welsh, and G. Morrisett. Flask: A language for data-driven sensor network programs. Technical report, Harvard '06.

[21] R. Müller, G. Alonso, and D. Kossmann. A virtual machine for sensor networks. In *EuroSys '07*.

[22] R. Newton, G. Morrisett, and M. Welsh. The regiment macroprogramming system. *IPSN '07*.

[23] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *SenSys '04*.

[24] J. Polastre, R. Szewczyk, and D. Culler. Telos: enabling ultra-low power wireless research. *IPSN '05*.

[25] G. Ramalingam. Data flow frequency analysis. In *PLDI '96*.

[26] H. Richardson. High performance fortran: history, overview and current developments. Technical report, Thinking Machines Corporation '96.

[27] E. Schonberg, J. T. Schwartz, and M. Sharir. Automatic data structure selection in setl. In *POPL '79*.

[28] 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 '05*.

[29] E. Tilevich and Y. Smaragdakis. J-orchestra: Automatic java application partitioning. In *ECOOP '02*.

[30] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *NSDI'04*.

[31] K. Whitehouse, J. Liu, and F. Zhao. Semantic streams: a framework for composable inference over sensor data. In *EWSN '06*.

[32] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *MobiSys '04*.

[33] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: using rpc for interactive development and debugging of wireless embedded networks. In *IPSN '06*.

[34] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD '02*.