

ViRe: Virtual Reconfiguration Framework for Embedded Processing in Distributed Image Sensors

Rahul Balani, Akhilesh Singhanian, Ram Kumar Rengaswamy, Chih-Chieh Han,
Mani Srivastava
University of California at Los Angeles
420 Westwood Plaza
Los Angeles, California, USA

{rahulb, akhi, ram, simonhan, mbs}@ee.ucla.edu

Abstract

Emergence of new technology in sensor networks, such as low-cost CMOS cameras and others, has introduced more sophisticated sensing modalities such as imaging. Applications involving image processing introduce new challenges to the design of sensor network systems. As the embedded processing becomes more complex, in-situ reconfiguration is seen as the key enabling technology to maintain and manage such systems. Reconfiguration can be used for bug-fixes, introducing new features, and tuning system parameters to the operating environment.

This paper presents the ViRe framework that provides for in-situ reconfiguration of complex image processing applications. Applications, modeled as data-flow graphs, are composed from a library of pre-defined and reusable elements. An efficient run-time system, called the wiring engine, is installed on the nodes to manage the graph and interaction between its elements. It facilitates communication by transferring data in the form of *tokens*. After initial deployment, the system permits reconfiguration of the graph by allowing modification to the edges and addition/removal of elements. Hence, it is able to support complex graphs comprising of elements with multiple fan-in and fan-out and feedback, while incurring a low memory and execution overhead.

Categories and Subject Descriptors: C.3 [Special - Purpose and Application-Based Systems]: Real-time and embedded systems

General Terms: Performance, Design, Reliability

Keywords: Sensor Networks, Reprogramming, Reconfiguration, Dataflow

1 Introduction

Wireless sensor networks have emerged as an important new tool for observing the physical world in many application domains. For a variety of reasons including cost, energy, processing, storage, and communication constraints, the initial generations of these systems focused on simple scalar physical sensing modalities, such as temperature, light, acoustics etc. Due to both application pull and technology push, there has recently been a transition towards more complex and sophisticated sensing modalities including image sensing. Inexpensive low-power CMOS imagers can be used, when coupled with suitable embedded algorithms, as versatile sensors capable of measuring a variety of physical world attributes such as light-level, shape, motion, color, size etc. from which behavioral inferences may be derived about the observed environment for recognition, monitoring and surveillance [32] [16]. Their ability to provide more complex forms of awareness about the situational state or the context of the events [4] have made image sensors the solution of choice for a number of societal, research and educational efforts [27].

Despite the vital importance of image sensing in understanding and characterization of diverse environment, applications involving these sensors pose new challenges to the design of sensor network systems [6], particularly in the context of recently developed low-power but resource-constrained image sensing platforms such as Cyclops [28], AER Imager [30], and [8]. The large data sizes and rates and complex computation associated with images severely stress or break many of the standard assumptions and optimizations underlying conventional sensor networks which have traditionally focused on simpler sensing modalities. In-network sensor information processing takes on a particularly important role with image sensors because to save communication bandwidth and energy, image sensor nodes must process the images *in-network* to extract relevant event or feature before transmission [31]. Local processing of image sensor data on resource constrained platforms raises several challenges not commonly encountered in sensor networks. Typical image processing algorithms are complex, and they require system software support for efficient and reliable functioning when embedded in resource-constrained sensor nodes. A common way of expressing image processing algorithms is as data-flow graphs involving feedback and elements with multiple

fan-ins and fan-outs.

Any embedded processing done at a sensor node with the goal of data reduction must naturally throw away data of lower application utility or poor integrity, and thus the embedded image processing has to be application-specific. While in the early days of sensor networks they were viewed as systems that will perform a single well-defined task over their entire lifetime, recently alternative usage models have emerged where the same hardware infrastructure gets used for limited term sensing tasks and campaigns over the system lifetime, perhaps by different users. Moreover, embedded sensor processing also needs to be environment-specific as the processing algorithms and parameters of these algorithms often have to be tuned either autonomously or interactively subsequent to deployment. This is particularly important in the case of complex sensing modalities such as imaging. Finally, the embedded processing may need to undergo bug fixes and addition of features. Clearly, the ability to reconfigure the application-specific embedded sensor processing on the nodes at run-time for purposes of re-tasking and environment-specific tuning, without sacrificing the efficiency of processing, is important for effective operation and maintenance of complex sensor networks. This recognition has led to emergence of systems such as Contiki [9], SOS [14], TENET [10], VanGo [12], and Mate [21] where run-time re-tasking and reconfiguration of application-specific processing at the node are directly supported. However, these prior systems are either too low-level (e.g. Contiki and SOS) and thus do not provide support for higher level programming abstractions common to image sensing, or significantly restrict the complexity or efficiency of application-specific processing and flexibility of reconfiguration (e.g. TENET, VanGo, and Mate) and thus unable to support the requirements of reconfigurable embedded image processing.

The main contribution of this paper is the ViRe (Virtual Reconfiguration) framework which explores a different point in the design space of re-taskable and reconfigurable embedded sensor processing, targeting specifically embedded image sensor processing on resource constrained devices. ViRe exposes a visual modular wiring diagram abstraction that is commonly used to express image processing algorithms. The modules encapsulate image processing functions, while the wiring is used to express dataflow which may be non-linear and with loops. Representing these algorithms as dataflow graphs has several advantages. First, it makes it easier for *non* software programmers to compose them. Second, this dataflow framework can help in automating application composition as it is easier to reason about and use a set of pre-defined general components [10] [23].

ViRe permits the wiring, module code, and module parameters to be incrementally reconfigured. The *virtual* reconfiguration of a image sensor node enabled by ViRe complements the physical reconfiguration of image sensors via pan-zoom-tilt capabilities as has been explored by researchers such as [17] and [29]. The ViRe run-time is optimized for the high rates imposed by image data and for the complex recursive algorithms encountered in image processing. For example, each image sample or frame is, in the context of these resource-constrained platforms, large (a

128x128 8-bit frame corresponds to more than 16K ADC samples of a simple 10-bit scalar sensor on a mote), and therefore mere filter chains (e.g. VanGo), linear dataflow graphs (e.g. TENET), and SQL aggregation functions (e.g. TinyDB [24]) are not enough to affect adequate data rate reduction. Rather, significantly richer and complex recursive algorithms are needed to compress image frames, or to detect features or events of interest. Another reason for the recursive nature of the computation is that the high cost of image acquisition itself usually motivates application structures where a feedback control loop controls when to acquire the next image frame based on history. ViRe also optimizes the handling of image data since copying an entire sample (image frame), or worse a full block of multiple samples, between processing functions would be prohibitively expensive requiring multiple reads and writes of large and typically off-chip frame-buffer memory [28].

1.1 Motivational Example

To motivate the need for ViRe consider a surveillance application scenario, where a network of wireless image sensors, or cameras, has been deployed to monitor objects in a given area. Initially, the deployment begins with a simple *image capture* application as shown in figure 1(i). The nodes capture images periodically and transmit them to a central server for processing. Such high-rate imaging applications pose significant design challenge. A single measurement sample from an image sensor is actually composed of many separate measurements, one for each pixel in the imager, and may range from a small number in low-resolution linear imagers to many millions for a high-resolution two-dimensional color imager [8]. To prevent loss of fidelity due to network congestion, local image processing at the node is often essential to extract an event or feature summary that is instead sent to the back-end.

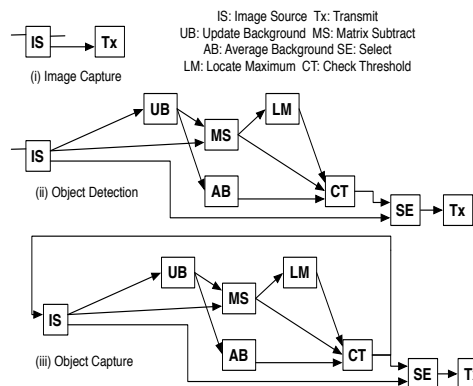


Figure 1. Application Graphs

Image processing algorithms are commonly expressed by domain experts as data-flow graphs where the raw input data is processed by a sequence of computational modules to extract desirable feature sets and perform application-specific action. In our sample system, after an initial learning phase to characterize the environment, we may want to switch to an operation mode where instead of extracting raw images at a low rate we start to extract feature summaries at a higher

rate and a full raw image only upon demand. This would require installation on the nodes of a custom *object detection* algorithm tuned to the application and the environment, as shown in figure 1(ii). The new object detection algorithm causes the nodes to transmit the image only when an object is detected by the camera. This significantly improves the bandwidth utilization and the quality of relevant images.

The *object detect* application operates on the image as follows: (i) MS takes the difference of the current image from IS against the stored background image in UB. (ii) LM locates the point of maximum difference in the resulting image, and (iii) CT compares the surrounding pixels against the average background in AB to detect the presence of an object. If an object is detected, SE sends the image, from the image source (IS), to the transmit (Tx) module that uses a simple *radioDump* service to propagate the image to the base station. This algorithm is not very robust, but performs satisfactorily well in static background scenarios. It is also comparatively simple in terms of computation requirements, but reasonably complex in terms of flow graph representation. Hence, it provides a good case for evaluation later.

Back to our sample scenario, in various experiments it was found that the objects were too far from the camera for performing reliable recognition. Therefore, the application was modified (figure 1(iii)) to zoom in and take a picture of the object before transmitting it to the base station. We call this third configuration as the *object capture* application. Here, the image source (element : IS) is fed back the coordinates of the maxima obtained in the previous step. It uses these coordinates to zoom in and take a picture of the object.

As seen here, in-situ reconfiguration of software was required to transform a simple, but inefficient, image capture graph to a more sophisticated and efficient object capture application.

The rest of the paper is organized as follows. Section 2 describes the design objectives of the ViRe framework based on the discussion above. It follows with a brief system overview, ending with a list of contributions of this work. Section 3 discusses related systems in detail to give an insight into the current state-of-the-art and support our objectives. The next section focuses on developing the flow graph syntax to help create complex applications easily. It also describes the execution semantics of the application, followed by a discussion on an alternative mechanism and its impact on the system characteristics. Section 5 describes the design and implementation of the embedded run-time in terms of the desired goals. Finally, section 6 evaluates the framework for memory, execution performance and reconfiguration costs using the application scenario discussed above. Section 7 concludes the paper.

2 Design Objectives

Till recently, much of the research had been geared towards providing a simple, manageable and efficient system by restricting the complexity of applications and flexibility of reconfiguration on the resource-poor sensor nodes [26] [13]. We argue that this approach under-utilizes the processing capabilities on the nodes. This restriction is more pronounced in imaging applications where it is necessary to use com-

plex algorithms for in-network processing of data [7] [32]. Typically, these algorithms are both memory and computation intensive. Memory accesses are required for manipulating image buffers to enable their transformation into concise application specific data structures. Thus, run-time support for these image processing algorithms should incur *minimal memory overhead* while allowing *efficient execution on resource constrained* nodes. These goals are often conflicting, and require a thorough analysis of the memory vs execution trade-offs before finalizing system design. Through this project, we aim to provide a lean framework for developing, executing and reconfiguring complex image processing applications on embedded sensor nodes. In addition, the system should provide explicit control over application functioning to facilitate *reliable* functioning on the sensor nodes. Finally, it should support *energy efficient* updates to both parameters and logic of embedded image processing algorithms at run-time.

We envision this system to be used by three different groups of people. First, *system developers*, like us, who will try to provide an efficient framework to achieve the design objectives and meet the constraints imposed by the domain or the platform. We aim to simplify their job by providing them with easy compile time configuration options. They also benefit from the detailed trade-off analysis mentioned above. Second, *application developers*, who will provide a library of image processing elements that can be used to form an application by the third group of people - the *general users*. The application developers are expected to be reasonably good at software programming, while the general user can be any person or domain expert who wishes to use the deployed sensor network irrespective of his programming background. We favor a simple element design, wherever feasible, to enable application developers to extend the system easily by adding more elements. Finally, the graphical front-end to this framework provides an easy-to-use interface for the general users to develop applications from pre-defined elements.

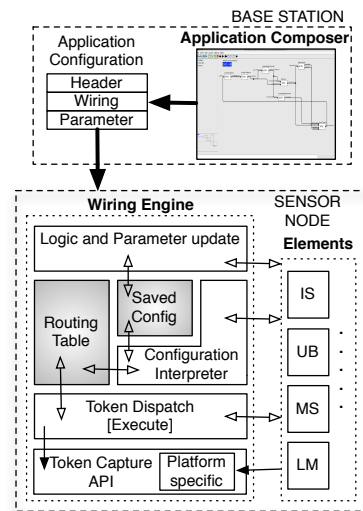


Figure 2. ViRe Framework

2.1 System Overview

The ViRe framework supports complex applications that can be represented by dataflow graphs with the following properties:

- cyclic or acyclic topology
- *elements* [18] with multiple input and output *ports*
- ports with fan-outs or fan-ins greater than one; and
- multiple instances of an element, in the same, or across different graphs.

The framework consists of two components: an *application composer*, that runs on the back-end server, and a *run-time engine* that is installed on the embedded sensor nodes. The software architecture is shown in figure 2. Users compose image processing graphs on the application composer that generates a concise configuration file, or *script*, consisting of edge and parameter information. This file is sent across to the target nodes that interpret it and install or update the corresponding application through the resident run-time engine, also referred to as the *wiring engine* in this text. A new file is generated at each reconfiguration and synchronized across the whole network as described above.

The application composer is implemented as a separate domain in Ptolemy II [5]. It provides for a loose form of reliability through ad-hoc type checking to ensure *consistent* connections between input and output ports of the elements. Information about the computational elements, such as number and type of input and output ports, is provided as an input to Ptolemy. It is currently maintained consistent with the element implementation manually, but work is going on to automate it.

The wiring engine provides support for updating the application parameters and wiring by appropriately patching the changes in the currently saved configuration through the *Logic and Parameter update* component. *Configuration Interpreter* then uses this updated configuration to install the new application. Separation of data (gray components in figure 2) and logic in the design of run-time engine provides for incremental updates to the graph wiring that can be applied unobtrusively to the system to prevent loss of historical data or work done on previous input. This is called a *hot-swap* [18].

Basic communication abstraction of the engine, a *token*, provides for efficient management of image data through cooperative memory sharing at run-time. The *Token Dispatch* mechanism ensures that execution follows correct data-flow semantics by coordinating the passage of tokens between elements. This coordination provides explicit control to the engine over application functioning. It is exploited to enable recovery from execution errors and facilitate a *safe* hot-swap during graph update.

Thus, in this paper, *we demonstrate that it is feasible to design and implement an efficient framework to support complex image processing applications on resource constrained sensor nodes*. We present a detailed analysis of the memory vs execution trade-offs provided by the system that allows simple compile time hooks to optimize either depending on the system requirements. For instance, in one experiment, it

is shown that sacrificing only 60 bytes of additional memory dramatically reduces execution overhead by a factor of 2. But even with memory conserving optimizations that sacrifice execution efficiency, the processing overhead is shown to be small (approx 5%) for our custom object detection algorithm dealing with large (> 4 KB) images. The system can also support multiple concurrent graphs, and allow modifications to the application logic and parameters by transmitting updates of sizes of the order of only 10 bytes. In addition, the element design is kept simple to promote ease in maintenance and extensibility by the application developers. Support for complex graphs also provides for general element implementation that can be re-used in multiple applications. It should be noted here, that even though the framework has been designed and tested for image processing sensor network applications, the concepts discussed here can be applied to other signal processing applications and the system can be used *as-is* for various domains.

The ViRe framework is implemented on top of SOS operating system [14] for the cyclops platform [28]. Cyclops is built as an imager sensor board for mote class devices like the Mica motes. Besides other components, it consists of a CMOS imager, an AVR AtMega 128L micro-controller [2], and 60 KB of external SRAM that is mainly used for buffering images.

3 Related Work

Data-flow graphs are a universally recognized specification medium for a large class of applications, in particular signal processing applications. This representation has been successfully extended to the sensor network applications as well which follow a typical *sensing - processing - transmission* model. Various systems have been built around this idea, each focusing on a different design objective. Some of the systems discussed below also aim to provide an efficient, though restricted, reconfiguration mechanism.

Flask: The Flask [25] abstraction aims to provide an easy-to-use dataflow programming model where the application composition language is based on OCaml [20]. It leverages the benefits of a programmatic wiring language to support complex application graphs on embedded nodes. This naturally lends strict type checking features of the language to the Flask system. But it restricts the graphs to be acyclic and the elements to have only one output port. Further, providing a reconfiguration mechanism was not an objective of the Flask system, and hence, it is bound to use the update capabilities provided by its base run-time TinyOS [22] and networked boot-loader [15]. However, our visual composition system and run-time support overcome both these limitations.

Discussion: The Flask compiler generates NesC code, and hence is tightly coupled with the TinyOS run-time. An interesting alternative direction to our project could be re-implementing the Flask compiler to produce code for a dynamic operating system like SOS, Contiki [9] or Mantis [1]. Besides the limitations discussed above, a drawback to this approach is that the learning curve for non software programmers, for using a programming language like OCaml, will be steep. This goes against our basic design objective of allow-

ing anyone to create and deploy sensor network applications easily. Implementing a visual graph abstraction on top of Flask will also result in many levels of indirection and making the overall system further complex and difficult to manage.

VanGo: VanGo [12] is a high data-rate collection framework where the sensor data processing chain spans from data acquisition on a mote, to data presentation on a micro-server to facilitate interactive tuning. The ability to choose where data filtering takes place enables flexible signal processing and decreases losses due to network congestion. But, the data path is restricted to a linear chain of filters for simplicity and ease in analysis, maintenance and debugging. Hence, we observe that VanGo allows restricted updates to the application to meet its design goals while keeping the system as simple as possible.

Tenet: The Tenet architecture [10] constrains multi-node fusion to the resource-rich *master* tier in hierarchical sensor networks to reduce complexity and improve manageability, and promote software re-use on the *slave* (mote) tier. Further, master nodes compose mote applications from a fixed set of *tasklets* in the Tenet task library. These tasklets expose parameters, and provide as much functionality as possible, while still retaining simplicity and ease of use. It trade-offs expressiveness in favor of simplicity for ease of construction and analysis. Hence, these applications are restricted to linear graphs.

The ViRe framework is able to demonstrate that the overhead incurred in supporting non-linear data processing graphs, and providing a more flexible reconfiguration mechanism on embedded sensor nodes, is insignificant over a similar system implemented in native code.

Semantic Services: A semantic services framework from Liu *et al* [23] aims to optimize resource utilization in collecting, storing and processing data from a sensor network, while facilitating its easy integration into the IT infrastructure. A data processing chain, consisting of semantic information extracting services, is automatically composed from the user queries, and can span across its hierarchical network of field servers and sensors. It is able to choose from a set of pre-defined services offered by the system, and thus can adapt to resource changes by switching to different services. The run-time system from the ViRe framework can complete this system by enhancing its adaptation, or reconfiguration, capabilities.

Snack: Finally, the Snack [11] framework aims to provide smart application service libraries, whose components weave to form applications by sharing control flow and state wherever possible. It emphasizes on producing memory and space efficient code, and leaves the reconfiguration mechanism to its static run-time [22]. Snack components can expose configuration level parameters, which are defined at compile time, and can accept a range of values for them to facilitate better sharing of control path. The Snack composition language is motivated from the Click router [18] which proposes a modular software architecture for building flexible and configurable routers. Click elements provide handlers for parameter reconfiguration, and the system is capable of *hot-swapping* a new router configuration seamlessly.

Since it is designed for a different class of systems, we do not discuss it further, but borrow the terms *elements* and *hot-swapping* for our system.

Discussion: From the above excerpt, we can observe that none of the existing systems try to provide run-time support for complex application graphs combined with a flexible and efficient reconfiguration mechanism. It can be argued that non-linear graphs can be modeled as linear graphs by collapsing multiple inputs and outputs of the elements, or collapsing several elements, to form elements with single input and output. For instance, a linear representation of the object detection application is shown in figure 3. However, such representations may have application specific components that can not be re-used easily in another application.

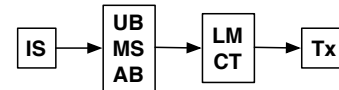


Figure 3. Linear representation of the Object Detect application shown in figure 1(ii)

Besides this, considerable amount of research has also gone into providing embedded run-times which support various reconfiguration mechanisms for sensor networks. TinyOS provides support for full image upgrades through its boot loader mechanism [15], while modular operating systems like SOS, Contiki and Mantis enable updates to smaller application modules. These systems do not provide direct support for higher level programming abstractions common to image sensing. Therefore all these systems can provide a base run-time for the ViRe framework.

Virtual machines [21] [3] [19] enable flexible and efficient application level changes to the system. The ViRe run-time engine can be considered a specialized virtual machine that interprets application configuration files generated by the visual composer, and installs the data-flow graph on the node. Application execution is performed by dynamic run-time linking using native code as opposed to using virtual machine scripts. This is necessary to support efficient execution of complex image processing applications on the sensor nodes.

4 Application Data-Flow Graph

In this section, we discuss syntax and execution semantics of the flow graph based representation of sensor network applications. The graph syntax should allow creation of complex applications, and the execution semantics should be efficient in terms of *resource utilization* on the embedded nodes.

Syntax : An application is created by composing a flow graph from a set of pre-defined *elements*. Each element is a computational module placed at a vertex of the graph. It communicates through the use of *ports* and can have multiple input and output ports. These elements can be connected to form a directed graph, where the graph edges (interchangeably called *wires*) represent flow of data between them. Each output port can be connected to multiple input ports. This is necessary to provide the ability to process same data in different ways within the same application. Likewise, each input port can be connected to multiple output ports and get

data from either of those. This enables the user to create complex application graphs, and promotes implementation of elements that is general and re-usable. Additionally, a graph can either be cyclic i.e. involving feedback, or acyclic.

An element also has associated parameters that control its functioning. These parameters can be accessed and modified through the application composer (section 5.3). It is also possible to have multiple instances of a module within the same graph, or across many concurrent graphs. In this case, each instance of the element gets a separate copy of its parameters, state and ports, but shares program code. This promotes code re-use, thus saving precious memory on the embedded nodes.

Execution Semantics : Execution begins when new input data is generated by the source element. Data is exchanged amongst elements in the form of *tokens*. An element is *fired* whenever its ready to accept new input on a port, and a token is placed on that port. It generally runs to completion once executed, and optionally produces an output and places it on a port. Depth-first traversal is followed, but ordering of branches is not defined [25]. Execution continues in that order till the input data is completely processed or queued in transit. Queuing of tokens occurs when an element is busy processing a previous token and is not ready to accept new tokens on one or more of its input ports. It can happen in either of the two cases: (i) The element may want to synchronize the tokens on its input ports before processing them simultaneously; or (ii) It may be involved in a long-running computation. In the latter case, it is said to follow *split-phase* semantics by breaking the computation into multiple phases and yielding control after processing a token partially. The input port is marked busy till the processing is complete.

Discussion : For the purpose of analysis, the application graph can be regarded as a pipeline where consecutive stages are separated by (i) elements that use split-phase semantics, or (ii) elements that depend on data from elements in set (i). The above execution semantics result in a truly asynchronous behavior with respect to the pipeline abstraction. For instance, consider the sample application given in figure 5(i). Assuming that element B invokes a split-phase operation after accepting a token from A, figure 4 shows the asynchronous pipeline representation of the application. If the sample rate of A is faster than the time taken by B to complete its processing, token queues will need to be maintained at the intermediate stage boundary. The analysis of such systems is complex, and execution can only be regarded as best-effort.

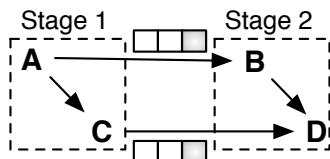


Figure 4. Pipeline representation of sample application given in figure 5(i)

An alternative, to the above operation, is to impose flow control at source and allow input data into the pipeline only

when previous data has been completely processed. The main advantage of this approach is that it keeps the system simple by queuing tokens at the source and provides better control over application functioning. But, it is a very conservative approach and can potentially decrease responsiveness of the system to input bursts faster than total application processing rate. For instance, it becomes critical in video processing applications where the effective frame rate is determined by the data processing time. In cyclops, the application processing rate was never a bottleneck to its net throughput because it consumed considerably more time in capturing an image (approx 970 ms) than processing it. But, we still chose to keep the asynchronous approach and provide a more responsive system at the expense of marginally increasing its complexity.

5 Wiring Engine: Design

Reconfigurable embedded image processing requires efficient support for complex algorithms and large data sizes and rates on the resource constrained sensor nodes. The ViRe framework run-time component - the wiring engine - has been designed to provide a reliable system for these applications with minimal memory and execution overhead. Its basic abstraction, a token, facilitates efficient sample processing and memory utilization during execution. Supporting complex data-flow representation on the sensor nodes introduces several challenges, including how to link the elements dynamically, at run-time, to support multiple fan-ins and fan-outs; how to design an inexpensive communication mechanism to permit efficient exchange of data; how to avoid race conditions due to feedback in recursive image processing algorithms; and how to detect errors in the system to provide reliable functioning.

In this section, we discuss the design of the embedded run-time engine with respect to its three main responsibilities: (i) Installing the graph on the node, (ii) Application execution, and (iii) Providing support for parameter and wiring reconfiguration. Each of these responsibilities is mapped on to a specific component in the run-time system as discussed in section 2.1.

5.1 Application Installation: Configuration Interpreter

During application installation, wiring engine transforms the configuration file into a simple and efficient representation of complex data-flow graphs on the embedded nodes. It is optimized to support fast execution with minimal memory overhead. A concise routing table, supported by a clear element design as shown in figure 5, is used to represent the graph edges. It is constructed by the wiring interpreter from a valid configuration file whenever the linking information is updated. It is looked up by the engine each time an element places a token on the output port. This step is repeated frequently during execution, and hence needs to be *efficient*.

The routing table consists of a group of *output port records*. A record begins with the number of destination ports connected to the output port, followed by a pointer to each one of them. Hence, each access to the record provides quick access to all fan-out ports when a token is placed on the corresponding output port. A port can be identified by

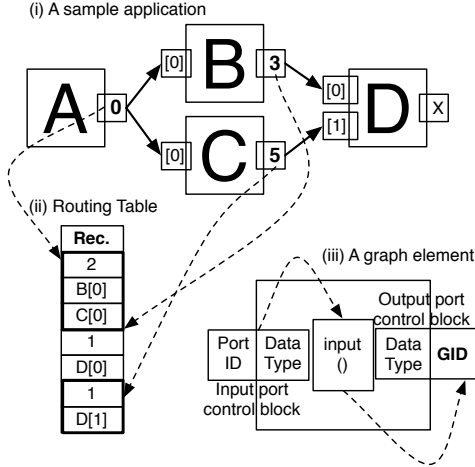


Figure 5. A sample application graph, its corresponding routing table and element design showing its input and output control blocks.

a $\langle \text{element ID}, \text{port ID} \rangle$ tuple. An output port is mapped to a globally unique identifier, called GID, that acts as a direct index into the routing table resulting in a constant access cost per entry in the record. Hence, this optimization reduces memory consumption and enables faster access to all the destination ports to which the corresponding output port is connected. This is critical to element communication described later (section 5.2), and hence provides interesting trade-offs in memory consumption vs execution discussed in section 6.2. On the down side, it only imposes an upper bound on the total number of wires in the graph.

Output port GID's can change with each wiring update. Thus, they need to be communicated to the elements at graph initialization. This communication can either be explicit, where the elements provide an interface to update the GID's for each output port, or implicit, where the engine automatically updates the corresponding output ports. Providing the former mechanism imposes an overhead on the elements that goes against our design of keeping the elements as simple as possible. So, an *output control block* is defined for each port, which stores the respective GID and *output-type*. This is part of the element meta-data and can be accessed by the engine directly to update the corresponding GID, assuming that the underlying run-time system supports it. SOS enables this through function control blocks in the module header. We expect it to be trivial, and otherwise useful also, to provide hooks to access the element meta-data as a default part of other systems.

Similarly, an *input control block* is defined for each input port which stores its locally unique ID, *input-type* and a pointer to the respective data processing block (*input()* in figure 5). The output-type and input-type are representative of data types produced or accepted at each port. They are used at initialization to validate the linking of input and output ports. Currently, it is a loose type checking mechanism as these *types* are user defined strings following a certain convention. Note that these strings are also used in the application composer to perform type checking on graph edges,

and thus this extra validation on the embedded node can help detect any inconsistency in the element implementation and Ptolemy input.

5.2 Application Execution: Token Dispatch and Token Capture API

Execution of image processing algorithms in ViRe framework consists of *application specific computation* and *communication* between graph elements. Communication makes use of the *token* abstraction provided by the engine. The token dispatch mechanism coordinates the exchange of data between graph elements to facilitate efficient communication. It has been designed to enable centralized control over the application. This is achieved through implicit interaction with the elements during data transfer. It is thus able to monitor the status of composing elements and maintain token queues for them. Further, it aids in wiring reconfiguration, specifically *hot-swap*, and error handling at the application level. It also prevents race conditions in elements that might occur due to feedback in complex image processing algorithms. Thus, it helps keep the element design simple by making the engine responsible for the above functionality. In addition, a *token capture API* provides an interface for the elements and engine to access the tokens. It provides explicit support for sharing memory in order to achieve efficient resource utilization during execution.

```

dispatch(output_block_ptr optr, token_type_t *t) {
    gid = get_gid(optr);
    element_id = get_caller_element(optr);
    num_dest_ports = read_routing_table_num(gid);
    set_element_busy(element_id);

    for i: 1 to num_dest_ports
        dest_port_cb = read_routing_table(gid + i);
        if (is_ready(dest_port_cb)) {
            status = call_input(dest_port_cb, t);
            if (status = BUSY) {
                set_port_busy(dest_port_cb);
            } else if (status < 0) {
                reset_graph();
                return;
            } else {
                queue_token(t, dest_port_cb);
            }
        }
    set_element_ready(element_id);
    schedule_pending_tokens_for(element_id);
}

```

Figure 6. Token Dispatch pseudo-code

Token Dispatch : After initialization, execution begins when the source element produces an output token. Elements call into the engine, through the *dispatch()* interface (figure 6), to place a token on their output port. The engine directly accesses the routing table to obtain the destination ports - one per call to *read_routing_table()* - connected to the particular port. It passes the token to each of them through the *call_input()* interface that invokes the corresponding *input()* function in the element from figure 7 or figure 8 respectively. Thus, both

`read_routing_table()` and `call_input()` are critical to performance of token dispatch. Section 6.2 discusses trading off memory to decrease the communication overhead due to `read_routing_table()` by a factor of 3.

The `call_input()` interface can be implemented either by using synchronous function calls, or asynchronous message passing to access the desired input port. Function call implementation was chosen over messages to lower memory consumption during execution and enable simple error reports by the elements. The process of constructing the routing table, thus consists of dynamically linking multiple functions to a single call and storing the pointers for future use. It makes extensive use of the run-time function subscription mechanism in SOS. As a result, all the destination port functions connected to an output port are conveniently invoked using the same copy of the token. It lowers memory consumption as tokens are only copied when required, as discussed later in this section.

The chain of synchronous calls results in a depth-first traversal of the graph. After accepting new input on a port, an element returns a busy status if it needs to invoke a split-phase operation (figure 8) or synchronize data on its input ports. Thus, the engine is able to monitor the status of all elements implicitly, and queue the tokens destined toward busy ports. Placing a token on an output port signifies the end of computation for an element. Due to synchronous nature of communication, it results in a complete traversal of the sub-graph rooted at the output port. We refer to it as one *output port iteration*. Hence, to avoid race conditions due to feedback in the same iteration, the element is marked busy, and all input tokens destined towards it are queued. The element status is reset at the end of the iteration to continue normal functioning. Next, the tokens queued for the element are processed in FIFO order to maintain their correct ordering and ensure consistent output.

In case of an error during execution, the elements simply return an appropriate error code when their input port is accessed (figure 7). Only the elements involved in a split-phase operation need to use a separate interface to signal an error that may occur later (figure 8). Currently, the error recovery mechanism comprises of complete graph reset. It involves re-initializing all the elements with latest parameters and discarding the token queues. Thus, complete application configuration needs to be saved across resets so that it can be used during the recovery process. This has been found to be very effective, and it keeps the system simple and manageable on resource-constrained motes.

Finally, the engine is capable of updating the graph incrementally while saving most of previous state whenever possible. It performs a simple status check for all elements, purges *dead* tokens from the queues, and proceeds with the *hot-swap* if none of the elements are involved in a split-phase operation. This is discussed in detail in section 5.3.

Token capture API : Communication between elements involves transfer of data wrapped in a token structure shown in figure 9. It simplifies management of data by providing support for tracking its read-write permissions, ownership and platform specific type. This information is used by the token capture API, shown in figure 10, to allow elements to

```
input(token_type_t *t) {
    data = get_token_data(t);
    length = get_token_length(t);
    length = process(data, length);
    if (error) return -EINVAL;
    my_token = create_token(data, length);
    dispatch(out_port_cb, my_token);
    destroy_token(my_token);
    return 0;
}
```

Figure 7. Input() function in elements without split-phase operation

```
input(token_type_t *t) {
    data = capture_token_data(t);
    length = get_token_length(t);
    length = process_partial(data, length);
    schedule_remaining_process(data, length);
    if (error) return -EINVAL;
    else return BUSY;
}
resume_split_phase(void *data, token_length_t length) {
    length = process_remaining(data, length);
    if (error) signal_error_to_engine();
    my_token = create_token(data, length);
    dispatch(out_port_cb, my_token);
    destroy_token(my_token);
}
```

Figure 8. Input() function in elements with split-phase operation

create tokens and cooperatively share memory by releasing and capturing them. It thus aids in memory and execution efficiency by avoiding multiple expensive reads and writes to copy large image buffers unnecessarily at each transfer. Further, it also enables support for platform specific data management through the *type* field in its structure.

The API lays down the following policies for token creation and usage at run-time. Token data can only be owned by either an element or the engine. By default, its assumed to be marked read-only, except for its owner that has exclusive write privileges. Thus, when an element needs to modify a token (data) belonging to another element, it must grab ownership of the data through the `capture_token_data()` interface. This interface transfers ownership and provides implicit copying of token data to relieve the elements of this responsibility. The owner can also facilitate memory sharing by releasing the token to enable other components to capture its data without copying.

```
typedef struct token_type_t {
    uint8_t type : 5;
    uint8_t owner : 2;
    uint8_t perm : 1;
    token_length_t length;
    void *data;
} token_type_t;
```

Figure 9. Token data structure

```

token_type_t *create_token(void *data,
    token_length_t length);
void *get_token_data(token_type_t *t);
void *capture_token_data(token_type_t *t);
void release_token(token_type_t *t);
void destroy_token(token_type_t *t);

```

Figure 10. Token Capture API

The token interface requires a platform (or domain) specific component (figure 2) that provides access to the data structures which might reference an external memory unit on the platform. For instance, cyclops uses an external RAM to store images captured by its sensor. The capture API invokes this platform specific component to manipulate structures (CYCLOPS_Image and CYCLOPS_Matrix) that access data stored on this external memory. Wiring engine needs to capture a token before queuing it, while the elements need to capture a token if they plan to modify it, or store it for later use.

An alternative approach to this policy would be to treat each token as strict read-only, and proactively pass a copy of token to each destination port. But, it was decided to use the former mechanism to reduce unnecessary memory consumption at run-time when destination elements do not need to modify the tokens. It should be noted that the policy is not enforced by our system, but is provided as a guideline to element writers.

5.3 Application Reconfiguration

The wiring engine is responsible for providing support for a *flexible* and *efficient* update mechanism on the embedded target. Flexibility in reconfiguration implies ability to update both logic and parameters of the application, while efficiency refers to minimizing the amount of energy spent in transmitting the update over the air. Its design is also dictated by three other objectives: (i) The update process should provide the ability to *hot-swap* the new configuration; (ii) It should not cause any data corruption; and (iii) Elements should be simple to write.

The reconfiguration can be divided into two categories for the purpose of discussion -

- Logic reconfiguration, which involves changes to graph wiring including addition or removal of elements at run-time
- Parameter reconfiguration

In both the categories, we separate the content of updates, transmitted over the air, from the software architecture built to support them on the motes. The update size needs to be kept as small as possible, while the mote software architecture should be kept simple for ease in maintenance and debugging. We observe that the above separation of concerns helps us to meet the design objectives effectively.

5.3.1 Logic Reconfiguration

We first discuss the software architecture required to support application logic reconfiguration on embedded nodes. Updates to an application can be further decomposed into two categories - (i) Major updates, which include significant modifications to the graph edges and elements such that the

application, or its implementation, changes as a whole. For instance, the change from image capture application to object detection application in figure 1; (ii) Minor, or incremental updates, which include small changes to the graph like addition or removal of a wire or an element. The ViRe framework provides support for both of these updates, but the option is left to the user at run-time and he can choose to use any of them depending on the application semantics. For major updates, it is necessary to halt the currently executing application, remove its components, and initialize the new application as described in section 5.1, to prevent data corruption from the previous installation. This is referred to as the *complete graph reset* in this paper.

It is interesting to discuss the update mechanism for incremental changes to the application. Ideally, such a mechanism should be able to automatically choose the best time to apply the update such that the changes are seamlessly incorporated into the system. There should be minimal loss of work on previously processed tokens and the data dependencies between elements should be maintained correctly. But, on the resource-constrained motes, it is not possible to precisely determine if, and when, an update can be unobtrusively applied to the system. Thus, to keep the system simple, we follow a conservative, but safe, policy. The engine does not apply the incremental update if a composing element is found to be *active*. An element is regarded as active if (i) it is busy processing previous input, or (ii) it is waiting for input on one or more ports. If none of the elements is active, the engine proceeds with the *hot-swap*. In the process, it destroys the unused elements in the latest graph, initializes new elements and re-installs the routing table according to the latest configuration. The state of the old unused elements is discarded, i.e. not migrated to the new elements that may have replaced them. Thus, the *hot-swap* is suggested to include only the elements that never contain application pertinent state. But, this does not prove to be a big concern as most of the processing done in data-flow graphs is stateless.

Additionally, during the *hot-swap*, the wiring engine purges tokens from the queue that were meant for unused elements destroyed in the previous step. These tokens are called the *dead* tokens. Thus, the new routing table ensures correctness of data dependencies, and the clean up of token queues prevents corruption from dead tokens. Additionally, ensuring that none of the elements are busy before installation, prevents inconsistency in output due to old tokens saved in the elements. The engine falls back to complete graph reset in case it is unable to apply the incremental update.

Besides our experience with using the SOS system, one of the main reasons for choosing it for our implementation was its low-cost dynamic module loading. We use this ability to add, modify and replace individual graph elements at run-time. Modifications to an element can be arbitrary - for instance, a change in the number of input or output ports. Thus, it becomes necessary to halt the complete application before installing the new version of the element. After installation, the old graph configuration has to be discarded as it might contain inconsistent information about the replaced element.

For the content of updates, the choice is only between

transmitting complete graph topology (*new graph update*) or the modifications since last synchronized configuration (*delta graph update*). Since, this is computed at the backend, it is easy to determine the smaller of the two update sizes and choose accordingly. The delta update will follow a compressed Unix *diff*-like format that can be interpreted by the engine to patch the previously stored complete configuration at the node. The new patched configuration will then be used to re-install the application graph according to user choice as discussed above.

Hence, we observe that keeping the wiring configuration separate from the engine logic helps achieve two objectives: (i) Small update size; and (ii) The ability to *hot-swap* the graph. An alternate design for the run-time engine is to generate a customized wiring logic for each graph configuration. The resulting engine is more efficient at memory utilization and execution performance, but fails to achieve the goals mentioned above. It was observed that the custom engine was approximately 3.8 KB in size for the object detection application, as compared to 140 bytes required to represent its dynamic configuration. Thus, we consciously chose to keep the current design instead of the latter.

5.3.2 Parameter Reconfiguration

The parameter reconfiguration mechanism on the embedded nodes is designed primarily to keep the element implementation simple. To support these updates, the engine needs to communicate the latest parameters to the elements, either explicitly or implicitly. Ideally, the engine should patch the updates directly so that the new values can be used seamlessly by the element on subsequent invocations. However, this approach is potentially unsafe as the element can be involved in a long-running computation which may have used previous values of the parameters for partially processing the input. Hence, we favor explicit communication where the engine passes a buffer of parameters to the element which is then responsible for updating its own parameters whenever it deems safe. This decision is pushed towards the elements as the engine does not have precise information about the element implementation. Thus, correctness of output is favored over simplicity in element design. However, we would like to add that this mechanism is not a novel contribution of the paper, and is mentioned here briefly only to highlight the reasoning behind it.

6 Evaluation

This section presents an experimental evaluation of the ViRe framework through the surveillance application series discussed in section 1.1. We demonstrate that it is actually possible to support complex data-flow graphs efficiently on resource constrained micro-controllers. Even with implementation decisions favoring memory conservation over execution efficiency, we show that the communication overhead imposed by the system over native code execution (SOS) is insignificant (approx 5%) for image processing dominant applications (image size > 4 KB). However, this overhead increases to about 26% for the applications as their computation time decreases (image size = 1 KB). We identified and eliminated some communication bottlenecks, reducing the overhead by a factor of 2 at a small expense of increasing

memory consumption. Furthermore, we can dramatically reduce the logic reconfiguration costs of the application by at least an order of magnitude, as compared to similar modifications supported by SOS run-time without the ViRe framework.

For our experiments, we implemented the applications in SOS with and without the ViRe framework, called *x-vire* and *x-sos* respectively. The cyclops nodes were initially installed with the base SOS kernel. For *x-vire*, the wiring engine and a library of image processing elements were pre-compiled and installed on the nodes; alternatively they could also be installed remotely over the network at run-time. Finally, application configurations were sent over the radio as required and the output from the nodes was obtained at the base station. Correspondingly, *x-sos* implementations assumed pre-compiled image processing libraries on the nodes, which provided various operations as direct function calls to SOS modules. Thus, the *x-sos* applications can also be viewed as graph wiring implementations in native code; hence provide a fair basis for evaluation. We evaluated the *x-vire* applications against the *x-sos* versions to obtain reconfiguration costs, and memory and execution overheads.

Currently, there is no emulation support for the cyclops port of SOS. Thus, for measuring execution time for platform specific operations, using an oscilloscope, we probed a GPIO pin for expected transitions. For the remaining operations, which are platform independent, we used the Avrora simulator. To characterize the measurement noise in the oscilloscope, we ran very simple applications on Avrora and oscilloscope and measured the execution times. We measured the operating frequency of the testbed hardware to be 7.30 Mhz and used it to convert between CPU cycles and real time. The measurement noise ranged from about 5% for a short term 8K CPU cycles application to 3% for a long running 500K CPU cycles application.

6.1 Update Cost

The ViRe framework allows both logic and parameter reconfiguration for the application graphs. Parameter reconfiguration capability provided by the system can be easily replicated in corresponding *x-sos* applications. Hence, in this section, we try to measure the decrease in logic update cost due to the ViRe framework. Researchers in [3] have observed that the energy spent in transmitting updates over the network is directly proportional to the update size. We use a similar metric to measure the application reconfiguration cost.

Applications	Update Size (bytes)	
	<i>x-sos</i>	<i>x-vire</i>
Image capture	620	26
Object Detect	4733	140
Object Capture	1263	21

Table 1. Logic reconfiguration cost

Table 1 compares the update size for *x-vire* applications, while moving from image capture to object detect and object capture, against the corresponding modifications in *x-sos* applications. It can be observed that the update size - and hence, update energy - for *x-vire* versions is at least an or-

der of magnitude lower than its x-sos counterparts. All x-sos versions were implemented as a single SOS module to conserve memory, and decrease communication overhead and code size. Any update to them required transmitting either the whole module, or a binary *diff* image, wirelessly over the network. For image capture and object detect, the whole module was sent as they were the first application and vastly different from previous application, respectively. We used a *diff* computing algorithm similar to [33] to compute the size of object capture update under x-sos. Even though, SOS currently does not support diff patching, we used the diff size to give a fair evaluation of the reconfiguration capabilities of the ViRe framework. The corresponding changes in x-vire required transmission of only the application configuration files. Tasking the nodes to image capture and retasking from image capture to object detection required a complete graph configuration file, while the minor change from object detect to object capture required a small delta-update.

6.2 Execution Performance

We next evaluate the execution overhead imposed by the ViRe framework over SOS run-time system. This is the dynamic communication overhead in the system due to token dispatch and consequent accesses. First, we measure the time taken to process an image during object detection and try to identify the most expensive operations. We observe that the image processing time is dominated by memory accesses to the off-chip image buffer, and hence is directly proportional to the image size, or resolution. Conducting similar experiments for both x-vire and x-sos application series confirms that the ViRe communication overhead is independent of the data that is exchanged between components. Further investigations reveal that the token dispatch cost is closely tied to the graph topology i.e. total number of output ports and the fan-out of each port. Thus, for a constant graph configuration, the proportion of communication overhead decreases as the image size, or the computation, increases. Finally, using the ViRe communication cost as a metric, we analyze the trade-offs between memory consumption and execution efficiency. In the experiment with object detection application, it is observed that by sacrificing an additional 57 bytes of RAM, the execution overhead can be lowered by a factor of 2.

Experiments and Results: Initially, a set of six experiments were carried out, one for each version of the surveillance application series. We measured the time taken to process one image sample and invoke the *radioDump* service for the result whenever required. This is referred to as the total execution time and is displayed in table 2 for all the applications. It neither includes the time taken to capture an image sample, nor the time taken to transmit it over the network. For each experiment, the default sample rate was kept low enough to measure system performance without any queuing delays. In addition, each experiment was carried out for 10 sample periods where an object was introduced at each alternate iteration. The image resolution (hence, size) was kept constant at 128 X 128 pixels (at 8-bit, size = 16 KB).

First, consider the x-sos series from table 2. It can be observed that the execution time of object detect and object capture applications is dominated by image processing. This

Applications	Execution time		
	x-sos	x-vire	
		Mem. Opt.	Exec. Opt.
Image capture	82 μ s	322 μ s	162 μ s
Object Detect	136 ms	141.6 ms	139.2 ms
Object Capture	136.2 ms	142 ms	139.5 ms

Table 2. Total execution time
[using oscilloscope]

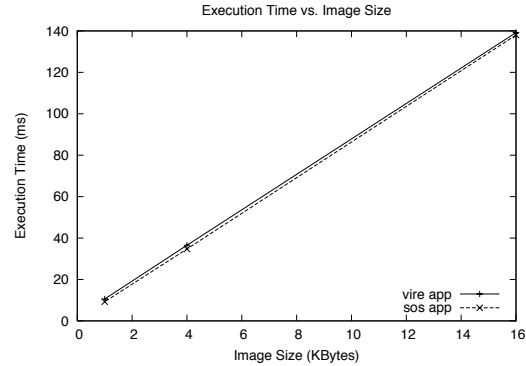


Figure 11. Object detection: Execution time as a function of image size

is evident from the fact that image capture, which does not process any image, takes lesser time by four orders of magnitude. Operations corresponding to *Matrix Subtract* and *Locate Maximum* together account for 132.6 ms in object detection. This is mainly because of slow SRAM access to image buffers on the cyclops platform.

The default implementation of the ViRe framework for cyclops favors memory optimizations given the meagre 4 KB of RAM on the AVR micro-controller. This affects only a 5% execution overhead in object detection and object capture, but a whopping 3x overhead in image capture as shown in the memory optimized section under x-vire series in table 2. In absolute terms, it can be observed that the communication in ViRe depends on the structure of the graph - a bigger graph incurs 5.6 ms overhead, while a smaller graph incurs only 240 μ s overhead. Later, in this section, it is demonstrated that this cost depends on the total number of output ports and the fan-out of each output port.

Another set of experiments was conducted for the object detect application (x-sos) with varying image resolutions, in order to verify the dependence of image processing time on the size of image buffer. Figure 11 confirms our claim by showing that the application execution time is directly proportional to the image size. Additionally, for similar experiments with x-vire series, it can be observed that the execution overhead is constant and independent of data size. This is in accordance with the design of the wiring engine. But, on the downside, it implies that the communication costs account for a significant proportion (26% for image size = 1 KB) of the total execution costs as the computation time decreases. This is reinforced by the comparison presented for the simple image capture application in table 2.

Next, we try to analyze the communication cost imposed by the ViRe framework and identify the bottlenecks for faster

execution. Since it is data - and hence, domain - independent, we use the Avrora simulator to measure the cost in terms of CPU cycles. Communication between elements can be broken down into two distinct components, namely, token dispatch and data access through the token capture API, as discussed in section 5.2.

Token Dispatch: The two main components of token dispatch mechanism are `read_routing_table()` and `call_input()` (figure 6). Each access to a destination port through `call_input()` is constant (87 CPU cycles) as it uses a uniform call interface across all elements, and has direct access to the input function control block through the routing table. This dynamic function call across modules is extensively used in SOS to support synchronous communication at run-time when the destination function is unknown at compile time [14].

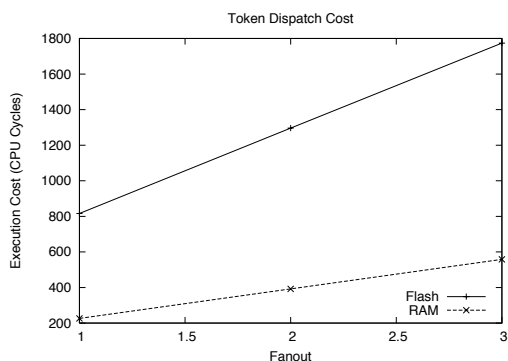


Figure 12. Token dispatch cost as a function of output port fanout

Further, each access to the routing table is direct and constant as described earlier. We decided to store this routing table in the program memory (128 KB internal flash) of the AVR micro-controller to optimize RAM consumption. This results in a high fetch cost (331 CPU cycles) due to abstractions imposed by SOS for accessing the flash memory. A corresponding access to RAM costs only 26 CPU cycles. Thus, we demonstrate that the token dispatch cost per output port is a linear function of its fan-out. Figure 12 presents this dispatch cost as the time required to call engine dispatch and access all the fan-out ports excluding their data processing time. Thus, we conclude that moving the routing table to RAM decreases the token dispatch cost by at least a factor of 3. It implies an additional 57 bytes of RAM for the object detect application, but only a 6 byte increase for image capture. Finally, it is trivial to observe that the communication overhead in an application, due to token dispatch, is a function of the total number of output ports and the fan-out of each port.

Interface	Execution Cost (CPU cycles)
<code>create_token</code>	429
<code>get_token_data</code>	15
<code>capture_token_data</code>	443 / 76
<code>destroy_token</code>	186

Table 3. Token capture API: Execution cost

Token Capture API: Table 3 represents the platform (or domain) independent cost of providing the token abstraction through the capture API. The token creation and capture cost is significant as compared to the token dispatch cost. It can be attributed to expensive calls to allocate memory using the default SOS dynamic memory manager. Typically, placing a token on an output port will invoke one call to `create_token()` and possibly multiple calls to `get_token_data()` or `capture_token_data()` depending on the fan-out of the port. From our experiments with object detection application under x-vire, we observed that (i) majority of the dynamic memory allocated and freed during execution was for tokens (section 6.3); and (ii) using default memory allocator for tokens resulted in twice as much RAM wastage (a phenomenon typically observed in other dynamic memory management systems as well for small size allocations). Thus, we implemented a custom block based memory management system, on top of default memory allocator, specifically for tokens. It allocates memory for n number of tokens (compile time configurable) at initialization, and provides a constant cost allocate (94 cycles) and free (28 cycles) interface to the rest of the framework. As a result, the communication overhead for object detection decreased by 14%. The value of n will vary depending on the application, environment and domain; thus, this optimization can be enabled or disabled at compile time to allow the users to determine a good value of n empirically by using the default allocator.

The combined effect of both compile time optimizations can be seen under the execution optimization section under x-vire in table 2. It varies from a 42% decrease in communication overhead in object detection to a 66% decrease in image capture.

6.3 Memory Consumption

We next measure the additional memory consumption in x-vire applications over their x-sos counterparts. In the process, we demonstrate that the support for complex application graphs on embedded nodes is extremely memory efficient, in terms of both static and dynamic usage. The static memory allocated for the wiring engine is less than 1% of the total available memory on the AVR micro-controller, and accounts for only 5% of the memory allocated for the SOS kernel on cyclops. Further, additional memory consumed by the elements after initialization is only 2% of the total dynamic memory (2 KB) available to SOS modules. Finally, during installation and execution of the graph, it is shown that the maximum dynamic memory allocation is dependent only on the graph properties like total number of elements, maximum fan-out observed and maximum height. This is, of course, assuming that no queuing of tokens takes place during execution and all execution optimizations have been disabled.

Experiments and Results: Memory consumption of the ViRe framework can be broken down into three separate components: Static, Semi-Dynamic, and Dynamic. Static and Semi-Dynamic consumption was measured manually by recording the state of the engine and elements, whereas dynamic consumption was logged in separate experiments, but using same data set and parameters as those in section 6.2.

Static allocation, shown in table 4, is independent of the

Allocation Type	Component	Memory (bytes)
Static	Wiring Engine	36
	Graph Elements (RAM)	41
Semi-Dynamic	Routing Table (Flash)	IC - 6
		OD - 57
		OC - 60
	Saved Configuration (Flash)	IC - 18
		OD - 120
		OC - 126

Table 4. Memory Allocation

IC - Image Capture, OD - Object Detect, OC - Object Capture

application graph and consists of memory allocated to the wiring engine. It is required to record input port status of each element in the graph, and keep track of pointers to routing table and saved graph configuration. *Semi-Dynamic allocation* consists of RAM allocated to the elements after initialization, which is static only for the lifetime of the graph. It is required to support communication abstractions of the ViRe framework and application specific parameters. Table 4 shows memory consumption incurred only due to the ViRe abstractions. Another component of this memory is allocated to the routing table and complete graph configuration, both of which are currently saved on flash to conserve data memory.

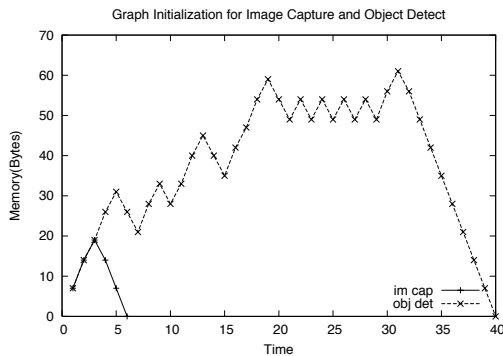


Figure 13. Dynamic memory allocation during graph installation

Dynamic allocation consists of memory allocated at run-time to support graph installation, update and execution. Memory consumption during installation and update is only a one-time cost and depends on the total number of elements and maximum fan-out observed in the graph. Figure 13 shows that it reaches a maximum of only 60 bytes for the complex object detection graph, and a mere 20 bytes for the simple image capture application. Hence, it is feasible to support these operations in parallel with other processing on the embedded node.

It is observed that during execution, memory is used predominantly for creation and capture of tokens by the elements or the engine. For systems without any queuing, the maximum dynamic allocation is directly proportional to the height of the graph and the tokens saved in the elements for input synchronization. For instance, figure 14 shows that maximum additional memory consumption for the object detection application is 25 bytes corresponding to 5 tokens cre-

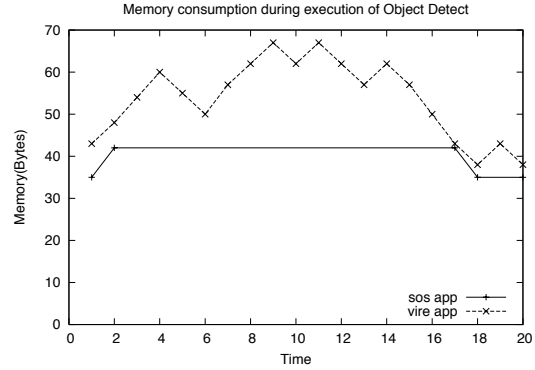


Figure 14. Dynamic memory Allocation during object detect execution

ated in the path from element IS to Tx via MS (figure 1(ii)). Thus, the height of the graph may be used as an initial estimate to determine the value of n discussed in the previous section for token capture optimization.

7 Conclusion

Traditionally, researchers have believed that complex run-time support for high level (re)programming of sensor network applications will be inefficient on resource constrained sensor nodes. Thus, they have focused on simple systems that are efficient but support only simple application programming. To achieve this, they restricted the flexibility in application composition to create non-complex applications. For instance, they allowed support for only linear data-flow graphs, forcing the use of application specific elements and hence, reducing reusability and generality of their application composition framework. For various reasons such as simple sensing modalities prevalent at that time, and lack of application pull and technology push, complex sensor network applications were not considered.

However, with the advent of more sophisticated sensing modalities such as imaging, it has become necessary to perform complex in-network processing. Contrary to the past belief, this paper demonstrates that a complex embedded run-time does not imply inefficiency. Further, it permits flexibility in representation of complex applications as data-flow graphs that may involve feedback and elements with multiple fan-ins and fan-outs. Finally, it allows efficient high level reconfiguration of image processing algorithms while incurring only a minimal memory and execution overhead.

In future work, we plan to test this framework, in domains other than image sensing, to evaluate how it scales down to lesser computation and memory intensive applications. Additionally, it would be interesting to explore how the system can be enhanced to support data-flow graphs that span across multiple nodes in a network, or across multiple processing cores on the same node.

8 References

- [1] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. MAN-TIS: System Support for Multimodal Networks of In-situ Sensors. *WSNA*, pages 50–59, 2003.

- [2] Atmel Corporation. *AVR 8-bit RISC Micro-controller* (<http://www.atmel.com/products/AVR/>).
- [3] R. Balani, C. Han, R. Rengaswamy, I. Tsigkogiannis, and M. Srivastava. Multi-level software reconfiguration for sensor networks. *EmSoft*, pages 112–121, 2006.
- [4] M. Baldauf and S. Dustdar. A survey on context-aware systems. *JUCI*, 2006.
- [5] J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *IJCS*, 4(2):155–182, 1994.
- [6] C. Chiasserini, E. Magli, and D. di Elettronica. Energy consumption and image quality in wireless video-surveillance networks. *ISPIMRC*, 5, 2002.
- [7] M. Chu, J. Reich, and F. Zhao. Distributed Attention for Large Video Sensor Networks. *IDSS*, 2004.
- [8] I. Downes, L. Rad, and H. Aghajan. Development of a Mote for Wireless Image Sensor Networks. *COGIS*, 2006.
- [9] A. Dunkels, B. Gronvall, and T. Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. *EMNETS*, 2004.
- [10] O. Gnawali, K. Jang, J. Paek, M. Vieira, R. Govindan, B. Greenstein, A. Joki, D. Estrin, and E. Kohler. The tenet architecture for tiered sensor networks. *Sensys*, pages 153–166, 2006.
- [11] B. Greenstein, E. Kohler, and D. Estrin. A sensor network application construction kit (SNACK). *SenSys*, pages 69–80, 2004.
- [12] B. Greenstein, C. Mar, A. Pesterev, S. Farshchi, E. Kohler, J. Judy, and D. Estrin. Capturing high-frequency phenomena using a bandwidth-limited sensor network. *Sensys*, pages 279–292, 2006.
- [13] R. Guy, B. Greenstein, J. Hicks, R. Kapur, N. Ramanathan, T. Schoellhammer, T. Stathopoulos, K. Weeks, K. Chang, L. Girod, et al. Experiences with the Extensible Sensing System ESS. *proceedings of CENS Technical Report*, 60, 2006.
- [14] C. Han, R. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava. Sos: A dynamic operating system for sensor networks. *Mobisys*, 2005.
- [15] J. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. *Sensys*, pages 81–94, 2004.
- [16] T. Kanade, R. Collins, A. Lipton, P. Anandan, P. Burt, and L. Wixson. Cooperative multi-sensor video surveillance. *DARPA Image Understanding Workshop*, 1:3–10, 1997.
- [17] A. Kansal, W. Kaiser, G. Pottie, M. Srivastava, and G. Sukhatme. Virtual High Resolution for Sensor Networks. In *Sensys*, November 1-3, 2006.
- [18] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek. The click modular router. *TOCS*, 18(3):263–297, 2000.
- [19] J. Koshy and R. Pandey. Vm*: Synthesizing scalable runtime environments for sensor networks. *Sensys*, 2005.
- [20] X. Leroy. *The Objective Caml system* (<http://caml.inria.fr>).
- [21] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. *SIGOPS*, 36(5):85–95, 2002.
- [22] 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 wireless sensor networks. *Ambient Intelligence*, 2005.
- [23] J. Liu and F. Zhao. Towards semantic services for sensor-rich information systems. *IEEE BROADNETS*, pages 44–51, 2005.
- [24] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. *SIGMOD*, pages 491–502, 2003.
- [25] G. Mainland, M. Welsh, and G. Morrisett. Flask: A Language for Data-driven Sensor Network Programs. Technical report, EECS, Harvard, 2006.
- [26] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *WSNA*, pages 88–97, New York, NY, USA, 2002. ACM Press.
- [27] K. Obraczka, R. Manduchi, and J. Garcia-Luna-Aveces. Managing the Information Flow in Visual Sensor Networks. *WPMC*, pages 27–30, 2002.
- [28] M. Rahimi, R. Baer, O. Iroezzi, J. Garcia, J. Warrior, and M. Srivastava. Cyclops: in situ image sensing and interpretation in wireless sensor networks. *EmNets*, pages 192–204, 2005.
- [29] D. Sundarraj, P. Gibbons, and P. Pillai. Ensuring Spatio-Temporal Consistency in Distributed Networks of Smart Cameras. *ICDSC*, October, 2006.
- [30] T. Teixeira, E. Culurciello, J. Park, D. Lymberopoulos, A. Barton-Sweeney, and A. Savvides. Address-event imagers for sensor networks: evaluation and modeling. *IPSN*, pages 458–466, 2006.
- [31] M. Wu and C. Chen. Multiple bitstream image transmission over wireless sensor networks. *Proceedings of IEEE Sensors*, 2, 2003.
- [32] D. Yang, H. Gonzalez-Banos, and L. Guibas. Counting people in crowds with a real-time network of simple image sensors. *ICCV*, pages 122–129, 2003.
- [33] T. Yeh, H. Yamamoto, and T. Stathopoulos. Over-the-air reprogramming of wireless sensor nodes. UCLA EE202A Project Report (2003) (http://lecs.cs.ucla.edu/~thanos/EE202a_final_writeup.pdf).