Wishbone: Profile-based Partitioning for Sensornet Applications

Ryan Newton, Sivan Toledo, Lewis Girod, Hari Balakrishnan, and Samuel Madden
MIT CSAIL

Abstract

The ability to partition sensor network application code across sensor nodes and backend servers is important for running complex, data-intensive applications on sensor platforms that have CPU, energy, and bandwidth limitations. This paper presents Wishbone, a system that takes a dataflow graph of operators and produces an optimal partitioning. With Wishbone, users can run the same program on a range of sensor platforms, including TinyOS motes, smartphones running JavaME, and the iPhone. The resulting program partitioning will in general be different in each case, reflecting the different node capabilities. Wishbone uses *profiling* to determine how each operator in the dataflow graph will actually perform on sample data, without requiring cumbersome user annotations. Its partitioning algorithm models the problem as an integer linear program that minimizes a linear combination of network bandwidth and CPU load and uses program structure to solve the problem efficiently in practice. Our results on a speech detection application show that the system can quickly identify good trade-offs given limitations in CPU and network capacity.

1 Introduction

An important class of sensor computing applications are data-intensive, involving multiple embedded sensors each sampling data at tens or hundreds of kilohertz and generating many megabytes per second in aggregate. Examples include acoustic localization of animals, gunshots, or speakers; structural monitoring and vibration analysis of bridges, buildings, and pipes; object tracking in video streams, etc. Over the past few years, impressive advances in sensor networking hardware and software have made it possible to prototype these applications. However, two challenges confront the developer who wants to deploy and sustain these applications:

- **Heterogeneity:** Thanks to hardware advances, one can run these applications on a variety of embedded devices, including "motes", smartphones (which themselves are varied), embedded Linux devices (e.g., Gumstix, WiFi access points), etc. This richness of hardware and software is good because it allows the developer to pick the right platforms for a task and evolve the infrastructure with time. On the other hand, it poses a software nightmare because it requires code to be developed multiple times, or ported to different platforms.
- **Decomposition:** A simple way of designing such systems would deliver all the gathered data to a central server, with all the computation running there. This approach may consume an excessive amount of bandwidth and energy. A different approach is to run all of the computation "in the sensor network", but often the computational capabilities of the sensor nodes are insufficient. The question is: how best to partition an application between the server(s) and the embedded nodes? Improper partitioning can lose important data, waste energy, and may cause applications to simply not work as desired.

No current solution addresses both of these challenges. To support heterogeneity, one might be able to write programs in a language like Java. Unfortunately, some platforms do not support Java, or may not support it in its full generality; in addition, Java virtual machines for embedded devices are of uneven quality. More importantly, it is difficult to partition such a program in a way that will perform well on any given platform without a significant amount of tuning and manual optimization. That, in turn, limits the ability to swap out the underlying hardware platform, or even to move computation between the embedded nodes and servers.

We have developed Wishbone, a system that allows developers to achieve both goals for applications that satisfy two conditions:

- Streaming dataflow model: The application should be written as a stream-oriented collection of operators configured as a dataflow graph.
- Predictable input rates and patterns: The input data rates at the sensors gathering data don't change in unpredictable ways.

To use Wishbone, the developer writes a program in a high-level stream-processing language, WaveScript [16], which has a common runtime for both embedded nodes and servers. We have extended our open-source WaveScript compiler to produce efficient code for several embedded platforms: TinyOS 2.0, smartphones running Java J2ME, the iPhone, Nokia tablets, various WiFi access points, and any POSIX compliant platform supporting GCC. These platforms are sufficiently diverse that generating high-performance native code from a shared high-level language is itself a challenge. Fortunately, we have an advantage in WaveScript's domain-specificity: the compiler has additional information that it can use to optimize programs for specific streaming workloads.

We have used WaveScript in several applications, including: locating wild animals with microphone arrays, locating leaks in water pipelines, and detecting potholes in sensor-equipped taxis. For the purposes of this paper,

we chose to focus on two applications that highlight the program partitioning features of Wishbone: a speech detector that identifies when a person is speaking in a room and a 22-channel EEG application. Each is based on an application currently in use by our group (EEG) or by other groups (speaker detection). Both were ported[1] to WaveScript for the evaluation in this paper.

The key function of Wishbone is, given a WaveScript-produced dataflow graph of stream operators, to partition it into in-network and server-side components. It uses a *profile-driven approach*, where the compiler executes each operator against programmer-supplied sample data, using real embedded hardware or a cycle-accurate simulation. After profiling, we are able to estimate the CPU and communication requirements of every operator on every platform. Wishbone depends on this sample data being representative of the actual input the sensor will see during deployment; we believe this is a valid assumption and justify it in our experiments.

Determining a good partitioning is difficult even after one uses a profiler to determine the computational and network load imposed by each operator. Wishbone models the partitioning problem as an integer linear program (ILP), seeking to minimize a combination of network bandwidth and CPU consumption subject to hard upper bounds on those resources. With these criteria, our ILP formulation will find optimal solutions—and although ILP is an NP-hard problem, in practice our implementation can partition dataflow graphs containing over a thousand operators in a few seconds.

Our results show that the system can quickly identify the optimal partition given constraints on CPU and network capacity. And picking the right partition matters. In our evaluation, our weakest platform got 0% of speaker detection results through the network successfully when doing all work on the server, and 0.5% when doing all work at the node. We can do 20× better by picking the right intermediate partition. Because the optimal partitioning changes depending on the hardware platform and the number of nodes in the network, manual partitioning is likely to be tedious at best. For larger graphs (such as our 1412 node electroencephalography (EEG) application), doing the partitioning by hand with any degree of confidence becomes extremely difficult.

Finally, we note that we do not intend that Wishbone be used only as a completely automated partitioning tool, but also as a part of an interactive design process with the programmer in the loop. In addition to recommending partitions, Wishbone can find situations in which there is no feasible partitioning of a program; e.g., because the bandwidth requirements will always exceed available network bandwidth, or because there are insufficient CPU resources to place bandwidth-reducing portions of the program inside the sensor network. In these cases, the programmer will have to either switch to a more powerful node platform, reduce the sampling rates or the number of sensors, or be willing to run the network in an overload situation where some samples are lost. In the overload case, Wishbone can compute how much the data rates need to be reduced to achieve a viable partition.

[1] WaveScript is an imperative language with a C-like syntax. An initial port of an application from C/C++ is very quick: cut, paste, and clean it up. Refactoring to expose the parallel/streaming structure of the application may be more involved.

```
fun FIRFilter(coeffs, strm) {
    N = Array:length(coeffs);
    fifo = FIFO:make(N);
    for i = 1 to N-1 { FIFO:enqueue(fifo, 0) };
    iterate x in strm {
        FIFO:enqueue(fifo, x);
        sum = 0;
        for i = 0 to N-1 {
          sum += coeffs[i] * FIFO:peek(fifo, i);
        };
        FIFO:dequeue(fifo);
        emit sum;
    }
}
fun LowFreqFilter(strm) {
  evenSignal = GetEven(strm);
  oddSignal  = GetOdd (strm);
  // even samples go to one filter, odds the other:
  lowFreqEven = FIRFilter(hLow_Even, evenSignal);
  lowFreqOdd  = FIRFilter(hLow_Odd,  oddSignal);
  // now recombine them
  AddOddAndEven(lowFreqEven, lowFreqOdd)
}
fun GetChannelFeatures(strm) {
  low1 = LowFreqFilter(strm);
  low2 = LowFreqFilter(low1);
  low3 = LowFreqFilter(low2);

  high4 = HighFreqFilter(low3); // we need this
  low4  = LowFreqFilter(low3);
  level4 = MagWithScale(filterGains[3], high4);

  high5 = HighFreqFilter(low4); // and this one
  low5  = LowFreqFilter(low4);
  level5 = MagWithScale(filterGains[4], high5);

  high6 = HighFreqFilter(low5); // and this one
  level6 = MagWithScale(filterGains[5], high6);
  zipN([level4, level5, level6]);
}
```

Figure 1: Excerpts from running code in EEG-application. The "low level" FIRFilter function constructs new dataflow operators using `iterate`. FIRFilter is stateful because it maintains and modifies `fifo`. Higher level functions such as LowFreqFilter and GetChannelFeatures wire together a larger graph.

2 Language and front-end compiler

The developer writes a program in WaveScript that constructs a dataflow graph of stream operators. Each operator consists of a work function and optional private state. The job of the WaveScript front-end compiler is to partially evaluate the program to create the dataflow graph, whereas the WaveScript backend performs graph optimizations and reduces work functions to an intermediate language that can be fed to a number of backend code generators. Each work function contains an im-

perative routine that processes a single stream element, updates the private state for that dataflow operator, and produces elements on output streams. (Later, we will single out *stateless* operators that maintain no mutable state between invocations.)

A WaveScript source program can manipulate streams as values and thereby wire together operator graphs, as seen in Figure 1. The example in Figure 1 contains psuedocode that wires together the cascading filters found in one of the 22-channels of our EEG application. The evaluation of the `iterate` form creates a new dataflow operator and provides its work function. The return value of an `iterate` is its output stream. For example, the function `FIRFilter` in Figure 1 takes a stream as one of its inputs and returns a stream. Within the body of the `iterate` the `emit` keyword produces elements on the output stream. The equal (=) operator introduces new variables and the last expression in a {...} block is its return value. Type annotations are unnecessary.

2.1 Program Distribution

Thus far, our description applies to WaveScript programs that run on a single node. To support distributed execution, we extended the language to allow developers to specify which part of the dataflow graph should be replicated on all embedded nodes. This specification is *logical* rather than *physical*; the physical locations of operators are computed by Wishbone's partitioner using the programmer's annotations and profiler data.

To create the logical specification in Wishbone, the user places a subset of the program's top-level stream bindings in a `Node{}` namespace. All operators in the `Node{}` namespace are replicated once per embedded node. This separation is particularly important for stateful operators, because stateful operators in the `Node` partition have an instance of their state for every node in the network. Stateful operators on the server side are instantiated only once.

As an example, consider the code snippet in Figure 2, which shows a node/server program that samples data from the microphone and filters it. The operator `readMic`, producing the stream `s1`, must reside on each node, as it samples data from hardware only available on the embedded node. Because the `filtAudio` call producing `s2` is in the `Node` partition, its operators will be replicated once per node, but can be physically placed either on the embedded node or the server, depending on what the partitioner determines would be best. If `filtAudio` creates stateful operators, their state will need to replicated once per node, regardless of where they are placed. This example illustrates the basic repartitioning model, and shows that, while the system is free to move some operators, there are certain *relocation constraints* the partitioner must respect, discussed in the next section.

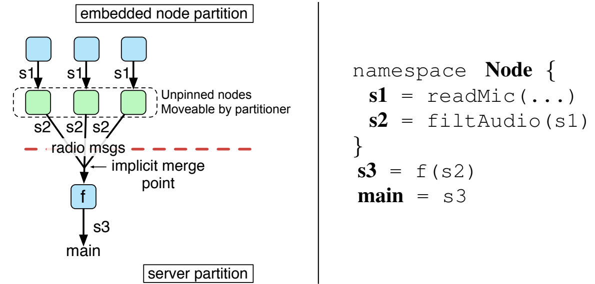


Figure 2: A program skeleton specifying a replicated stream computation across all embedded nodes.

2.1.1 Relocation Constraints

Operators are classified as *movable* or *pinned* as follows. First, operators with side-effects—for example, OS-specific foreign calls to sample sensors and blink LEDs—are pinned to their partition. Likewise, operators on the server that print output to the user or to a file are pinned. Stateless operators without side-effects are not pinned and are always moveable, allowing them to be moved into the other partition if the system determines that to be advantageous. Finally, stateful operators are treated differently for the node and server partitions. It is not generally possible to move stateful server operators into the network—they have a serial execution semantics and a single state instance. However, it is possible to move stateful operators from the node partition to the server. The state of the operator is duplicated in a table indexed by node ID. Thus, a single server operator can emulate many instances running within the network.

Relocating stateful operators in this way raises a different issue—message loss on wireless links. Operators in the node partition may safely assume that all edges between the raw sensors and themselves are *lossless*. Relocating an operator to the server means putting potential data loss upstream of it that was not there previously. Stateless operators are insensitive to this kind of loss because they process each element without any memory of preceding elements, but stateful operators may perform erratically in the face of unexpected missing data, unless they have been intentionally engineered to tolerate it.

Because tolerance to data loss in stateful operators is an application-specific issue, Wishbone supports two operational modes that can be specified by the programmer at compile time. In *conservative* mode it will not relocate stateful operators onto the server, refusing to add lossiness to a previously lossless edge. In *permissive* mode, the system will automatically perform these relocations. In the future, it would be possible to extend the system to make many finer distinctions, such as labeling individual edges as loss-tolerant, or grouping operators together in blocks that cannot be divided by a lossy edge.

2.1.2 Restrictions

The system we present in this paper targets a restricted domain: first, because we focus on a specific dataflow model and, second, because of limitations of our current implementation. (Section 9 will discuss generalizing and extending the model.) Presently, our implementation requires that any path through the operator graph connecting a data source on the node to a data sink on the server may only cross the network once. The graph partitioning algorithm in Section 4 does, however, support back-and-forth communication. The reason for the restriction is that we haven't yet implemented arbitrary communication for all of our platforms. Note that this does not rule out all communication from the server to the nodes, it is still possible, for example, to have configuration parameters sent from a server to in-network operators.

We make the best of this restriction by leveraging it in a number of ways. As we will see, it enables a simplified version of the partitioning algorithm. It can also further filter the set of moveable operators as described in Section 2.1.1, because pinning an operator pins all up- or down-stream operators (can't cross back).

3 Profile & Partition

The WaveScript compiler, implemented in the Scheme language, can profile stream graphs by executing them directly within Scheme during compilation (using sample input traces). This produces platform-independent data rates, but cannot determine execution time on embedded platforms. For this purpose, we employ a separate profiling phase on the device itself, or on a cycle-accurate simulator for its microprocessor.

First, the partitioner determines what operators might possibly run on the embedded platform, discounting those that are pinned to the server, but including movable operators together with those that are pinned to the node. The code generator emits code for this partition, inserting timing statements at the beginning and end of each operator's work function, and at emit statements, which represent yield points or control transfers downstream.

The partition is then executed on simulated or real hardware. The inserted timing statements print output to a debug channel read by the compiler. For example, we execute instrumented TinyOS programs either on TMote Sky motes or by using the MSPsim simulator[2]. In either case, timestamps are sent through a real or virtual USB serial port, where they are collected by the compiler.

For most platforms, the above timestamping method is sufficient. That is, the only relevant information for partitioning is how long each operator takes to execute on that platform (and therefore, given an input data rate, the percent CPU consumed by the operator). For TinyOS, some additional profiling is necessary. To support subdividing tasks into smaller pieces, we must be able to perform a reverse mapping between points in time (during an operator's execution) and points in the operator's code. Ideally, for operator splitting purposes, we would recover a full execution trace, annotating each atomic instruction with a clock cycle. Such information, however, would be prohibitively expensive to collect. We have found it is sufficient to instead simply time stamp the beginning and end of each `for` or `while` loop, and count loop iterations. As most time is spent within loops, and loops generally perform identical computations repeatedly, this enables us to roughly subdivide execution of an operator into a specified number of slices.

After profiling, control transfers to the partitioner. The movable subgraph of operators has already been determined. Next, the partitioner formulates the partitioning problem in terms of this subgraph, and invokes an external solver (described in Section 4) to identify the optimal partition. The program graph is repartitioned along the new boundary, and code generation proceeds, including generating communication code for cut edges (e.g., code to marshal and unmarshal data structures). Also, after profiling and partitioning, the compiler generates a visualization summarizing the results for the user. The visualization, produced using the well-known GraphViz tool from AT&T Research, uses colorization to represent profiling results (cool to hot) and shapes to indicate which operators were assigned to the node partition.

4 Partitioning Algorithms

In this section, we describe Wishbone's algorithms to partition the dataflow graph. We consider a directed acyclic graph (DAG) whose vertices are stream operators and whose edges are streams, with edge weights representing bandwidth and vertex weights representing CPU utilization or memory footprint. We only include vertices that can move across the node-server partition; i.e., the movable subset. The server is assumed to have infinite computational power compared to the embedded nodes, which is a close approximation of reality.

The partitioning problem is to find a cut of the graph such that vertices on one side of the cut reside on the nodes and vertices on the other side reside on the server. The bandwidth of a given cut is measured as the sum of the bandwidths of the edges in the cut. An example problem is shown in Figure 3.

Unfortunately, existing tools for graph partitioning are not a good fit for this problem. Tools like METIS [12] or Zoltan [7] are designed for partitioning large scientific codes for parallel simulation. These are heuristic solutions that generally seek to create a fixed number of

[2] We also tried Simics and msp430-gdb for simulation, but MSPsim was the easiest to use. Note that TOSSIM is not appropriate for performance modeling.

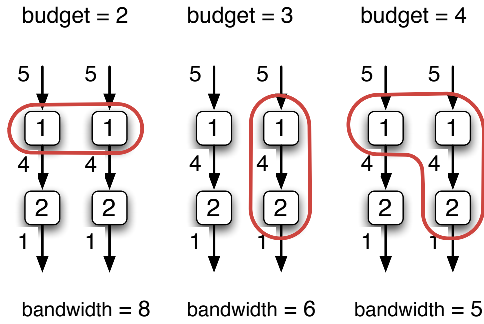


Figure 3: Simple motivating example. Vertices are labeled with CPU consumed, edges with bandwidth. The optimal mote partition is selected in red. This partitioning can change unpredictably, for example between a horizontal and vertical partitioning, with only a small change in the CPU budget.

balanced graph partitions while minimizing cut edges. Newer tools like Zoltan support unbalanced partitions, but with a specified ratios, not allowing unlimited and unspecified capacity to the server partition. Further, they expect a single weight on each edge and each vertex. They cannot support a situation where the cost of a vertex *changes* depending on the partition is it placed in. This is the situation we're faced with: diverse hardware platforms that not only have varying capacities, but for which the *relative* cost of operators varies (for example, due to a missing floating point unit).

We may also consider traditional task scheduling algorithms as a candidate solution to our partitioning problem. These algorithms assign a directed graph of tasks to processors, attempting to minimize the total execution time. The most popular heuristics for this class of problem are variants of *list scheduling*, where tasks are prioritized according to some metric and then added one at a time to the working schedule. But there are three major differences between this classic problem and our own. First, task-scheduling does not directly fit the nondeterministic dataflow model, as no conditional control flow is allowed at the task level—all tasks execute exactly once. Second, task-scheduling is not designed for vastly unequal node capabilities. Finally, schedule length is not the appropriate metric for streaming systems. Schedule length would optimize for *latency*: how fast can the system process one data element. Rather, we wish to optimize for throughput, which is akin to scheduling for a task-graph repeated ad infinitum.

Thus we have developed a different approach. Our technique first preprocesses the graph to reduce the partition search space. Then it constructs a problem formulation based on the desired objective function and calls an external ILP solver. By default, Wishbone currently uses the minimum-cost cut subject to not exceeding the CPU resources of the embedded node or the network capacity of the channel. Cost here is defined as a linear combination of CPU and network usage, $\alpha{\cdot}CPU{+}\beta{\cdot}Net$ (which can be a proxy for energy usage). Therefore we set four numbers for each platform: the CPU/Network resource limits, and coefficients α, β. The user may override these quantities to direct the optimization process.

4.1 Preprocessing

The graph preprocessing step precedes the actual partitioning step. The goal of the preprocessing step is to eliminate edges that could never be viable cut-points. Consider an operator u that feeds another operator v such that the bandwidth from v is the same or higher than the bandwidth on the output stream from u. A partition with a cut-point on the v's output stream can always be improved by moving the cut-point to the stream $u \rightarrow v$; the bandwidth does not increase, but the load on the embedded node decreases (v moves to the server). Thus, any operator that is data-expanding or data-neutral may be merged with its downstream operator(s) for the purposes of the partitioning algorithm, reducing the search space without eliminating optimal solutions.

4.2 Optimal Partitionings

It is well-known that optimal graph partitioning is NP-complete [8]. Despite the intrinsic difficulty of the problem, the problem proves tractable for the graphs seen in realistic applications. Our pre-processing heuristic reduces the problem size enough to allow an ILP solver to solve it exactly within a few seconds to minutes.

4.2.1 Integer Linear Programming (ILP)

Let $G = (V, E)$ be the directed acyclic graph (DAG) of stream operators. For all $v \in V$, the compute cost on the node is given by $c_v > 0$ and the communication (radio) cost is given by r_{uv} for all edges $(u, v) \in E$. One might think of the compute cost in units of MHz (megahertz of CPU required to process a sample and keep up with the sampling rate), and the bandwidth cost in kilobits/s consumed by the data going over the radio. Adding additional constraints for RAM usage (assuming static allocation) or code storage is straightforward in this formulation, but we do not do it here. For each of these costs we can use either mean or peak load (profiling computes both). Because our applications have predictable rates, we use mean load here. Peak loads might be more appropriate in applications characterized by "bursty" rates.

The DAG G contains a set of terminal *source* vertices S, and *sink* vertices T, that have no inward and outward edges, respectively, and where $S, T \subset V$. As noted above, we construct G from the original operator graph such that these boundary vertices are pinned—all the sources must remain on the embedded node; all sinks on the server. Recall that the partitioning problem is to find a single cut of G that assigns vertices to the nodes

and server. We can think of the graph G as corresponding to the server and a single node, but vertices assigned to the node partition are instantiated on all physical nodes in the system.

We encode a partitioning using a set of indicator variables $f_v \in \{0, 1\}$ for all v in V. If $f_v = 1$, then operator v resides on the node; otherwise, it resides on the server. The pinning constraints are:

$$\begin{array}{l} (\forall u \in S)\ f_u = 1 \\ (\forall v \in T)\ f_v = 0 \\ (\forall v)\ f_v \in \{0, 1\}\ . \end{array} \quad (1)$$

Next, we constrain the sum of node CPU costs to be less than some total budget C.

$$cpu \leq C \quad \texttt{where} \quad cpu = \sum_{v \in V} f_v c_v \quad (2)$$

A simple expression for the total cut bandwidth is $\sum_{(u,v) \in E} (f_u - f_v)^2 r_{uv}$. (Because $f_v \in \{0, 1\}$, the square evaluates to 1 when the edge (u, v) is cut and to 0 if it is not; $|f_u - f_v|$ gives the same values.) However, we prefer to formulate the integer programming problem as one with a linear rather than quadratic objective function, so that standard ILP techniques can be used.

We can convert the quadratic objective function to a linear one by introducing two variables per edge, e_{uv} and e'_{uv}, which are subject to the following constraints:

$$\begin{array}{l} (\forall (u, v) \in E)\ e_{uv} \geq 0 \\ (\forall (u, v) \in E)\ e'_{uv} \geq 0 \\ (\forall (u, v) \in E)\ f_u - f_v + e_{uv} \geq 0 \\ (\forall (u, v) \in E)\ f_v - f_u + e'_{uv} \geq 0\ . \end{array} \quad (3)$$

The intuition here is that when the edge (u, v) is not cut (i.e., u and v are in the same partition), we would like e_{uv} and e'_{uv} to both be zero. When u and v are in different partitions, we would like a non-zero cost to be associated with that edge; the constraints above ensure that the cost is at least 1 unit, because $f_u - f_v$ is -1 when u is on the server and v on the embedded node. These observations allow us to formulate the bandwidth of the cut, cap that bandwidth, and define the objective function in terms of both CPU and network load.

$$net < N \quad \texttt{where} \quad net = \left(\sum_{(u,v) \in E} (e_{uv} + e'_{uv}) r_{uv} \right) \quad (4)$$

$$\texttt{objective}: \quad \min \left(\alpha \ \ cpu + \beta \ \ net \right) \quad (5)$$

Any optimal solution of (5) subject to (1), (2), (3), and (4) will have $e_{uv} + e'_{uv}$ equal to 1 if the edge is cut and to 0 otherwise. Thus, we have shown how to express our partitioning problem as an integer programming problem with a linear objective function, $2|E| + |V|$ variables (only $|V|$ of which are explicitly constrained to be integers), and at most $4|E| + |V| + 1$ equality or inequality constraints.

We could use a standard ILP solver on the formulation described above, but a further improvement is possible if we restrict the data flow to not cross back and forth between node and server, as described in Section 2.1.2. On the positive side, the restriction reduces the size of the partitioning problem, which speeds up its solution.

With the above restriction, we can then flip all edges going from server to node for the purpose of partitioning (the communication cost would be the same under our model). With all edges pointed towards the server, and only one crossing of the network allowed, another set of constraints now apply:

$$(\forall (u, v) \in E)\ f_u - f_v \geq 0 \quad (6)$$

With (6) the network load quantity simplifies:

$$net = \left(\sum_{(u,v) \in E} (f_u - f_v) r_{uv} \right) . \quad (7)$$

This formulation eliminates the e_{uv} and e'_{uv} variables, simplifying the optimization problem. We now have only $|V|$ variables and at most $|E| + |V| + 1$ constraints. We have chosen this restricted formulation for our current, prototype implementation, primarily because the per-platform code generators don't yet support arbitrary back-and-forth communication between node and server. We use an off-the-shelf integer programming solver, `lp_solve`[3], to minimize (7) subject to (1) and (2).

We note that the restriction of unidirectional data flow does preclude cases when sinks are pinned to embedded nodes (e.g., actuators or feedback in the signal processing). It also prevents a good partition when a high-bandwidth stream is merged with a heavily-processed stream. In the latter case, the merging must be done on the node due to the high-bandwidth stream, but the expensive processing of the other stream should be performed on the server. In our applications so far, we have found our restriction to be a good compromise between provable optimality and speed of finding a partition.

4.3 Data Rate as a Free Variable

It is possible that the partitioning algorithm will not be able to find a cut that satisfies all of the constraints (i.e., there may be no way to "fit" the program on the embedded nodes.) In this situation we wish to find the maximum data rates for input sources that *will* support a viable partitioning. The algorithm given above cannot directly treat data rate as a free variable. Even if CPU and

[3] `lp_solve` was developed by Michel Berkelaar, Kjell Eikland, and Peter Notebaert. It uses branch-and-bound to solve integer-constrained problems, like ours, and the Simplex algorithm to solve linear programming problems.

network load varied linearly with data rate, the resulting optimization problem would be non-linear. However, it turns out to be inexpensive to perform the search over data-rates as an *outer loop* that on each iteration calls the partitioning algorithm.

This is because in most applications, CPU and network load increase monotonically with input data rate. If there is a viable partition when scaling input data rates by a factor X, then any factor $Y < X$ will also have a viable partitioning. Thus Wishbone simply does a binary search over data rates to find the maximum rate at which the partitioning algorithm returns a valid partition. As long as we are not over-saturating the network such that sending fewer packets actually result in more data being successfully received, this maximum sustainable rate will be the best rate to pick to maximize outputs (throughput) of the data flow graph. We will re-examine this assumption in Section 7.

5 Wishbone Platform Backends

In this section, we describe three new WaveScript code generators we built for Wishbone, which are described here for the first time. These support ANSI C, NesC/TinyOS and JavaME.

5.1 Code Generation: ANSI C and JavaME

In contrast with the original WaveScript C++ backend (and XStream runtime engine), our current C code-generator produces simple, single threaded code in which each operator becomes a function definition. Passing data via `emit` becomes a function call, and the system does a depth-first traversal of the stream graph. The generated code requires virtually no runtime and is easily portable. This C backend is used to execute the server-side portion of a partitioned program, as well as the node-side portion on Unix-like embedded platforms that run C, such as the iPhone (jailbroken), Gumstix, or Meraki.

Generating code for JavaME also straightforward, as Java provides a high level programming environment that abstracts hardware management. The basic mapping between the languages is the same as in the C backend. Operators become functions, and an entire graph traversal is a chain of function calls. Some minor problems arise due to Java's limited set of numeric types.

5.2 Code Generation: TinyOS 2.0

Supporting TinyOS 2.0 is much more challenging. The difficulties are both due to the extreme resource constraints of TinyOS motes (typically less than 10 KB of RAM and 100 KB of ROM), and to the restricted concurrency model of TinyOS (tasks must be be relatively short-lived and non blocking; all IO must be performed with split-phase asynchronous calls). Also, program objects be serialized and split into small network packets. Wishbone's support for TinyOS demonstrates its ability to use platforms with severe resource restrictions and unusual concurrency models.

Our prototype does not currently support WaveScript's dynamic memory management in code running on motes. We may support it in the future, but it remains to be seen whether this style of programming can be made effective for extremely resource constrained devices. Instead, we enforce that all operators assigned to motes use only statically allocated storage in our applications.

The most difficult issue in mapping a high-level language onto TinyOS is handling the TinyOS concurrency model. All code executes in either *task* or *interrupt* context, with only a single, non-preemptive task running at a time. Wishbone simply maps each operator onto a task. Each data element that arrives on a source operator, for example a sensor sample or an array of samples, will result in a depth-first traversal of the operator graph (executed as a series of posted tasks). This graph traversal is not re-entrant. Instead, the runtime buffers data at the source operators until the current graph traversal finishes.

This simple design raises several issues. First, generated TinyOS tasks must be neither too short nor too long. Tasks with very short durations incur unnecessary overhead, and tasks that run too long degrade system performance by starving important system tasks (for example, sending network messages). Second, the best method for transferring data items between operators is no longer obvious. In the basic C backend, we simply issue a function call to the downstream operator, wait for it to complete, and then continue computation. We cannot use this method under TinyOS, where it would force us to perform an entire traversal of the graph in a single very long task execution. But the obvious alternative also presents problems: executing an operator in its entirety before any downstream operators would require a queue to buffer all output elements of the current operator.

The full details of TinyOS code generation are beyond the scope of this paper. In short, the WaveScript compiler can convert programs programs into a cooperative multi-tasking form (via a CPS conversion). This serves two purposes: every call to `emit` can serve as a yield point, causing the task to yield to its downstream operator in a depth-first fashion (with no queues), which in turn will re-post the upstream operator upon completing the traversal. Second, based on profiling data, additional yield points can be inserted to "split" tasks to adjust granularity for system health.

6 Applications

We evaluate Wishbone in terms of two experimental applications: acoustic speech detection and EEG-based seizure onset detection. Both of these applications exercise Wishbone's capability to automatically partition a

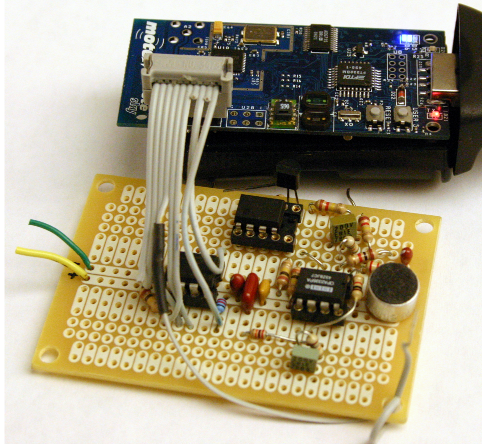


Figure 4: Custom audio board attached to a TMote Sky.

single high-level program into components that run over a network containing sensor nodes and a server or "base station". Neither of these applications is in itself novel. In both cases we ported existing implementations from Matlab and C to Wishbone and verified that the results matched the original implementations.

6.1 Application: Seizure Onset Detection

We used Wishbone to implement a patient-specific seizure onset detection algorithm [20]. The application was previously implemented in C++, but by porting it to Wishbone/WaveScript we enabled its embedded/distributed operation, while reducing the amount of code by a factor of four without loss of performance.

The algorithm is designed to be used in a system for detecting seizures outside a clinical environment. In this application, a user would wear a monitoring cap that typically consists of 16 to 22 channels. Data from the cap is processed by a low-power portable device.

The algorithm we employ [21] samples data from 22 channels at 256 samples per second. Each sample is 16-bits wide. For each channel, we divide the stream into 2 second windows. When a seizure occurs, oscillatory waves below 20 Hz appear in the EEG signal. To extract these patterns, the algorithm looks for energy in certain frequency bands.

To extract the energy information, we first filter each channel by using a polyphase wavelet decomposition. We use a repeated filtering structure to perform the decomposition. The filtering structure first extracts the odd and even portions of the signal, passes each signal through a 4-tap FIR filter, then adds the two signals together. Depending on the values of the coefficients in the filter, we either perform a low-pass or high-pass filtering operation. This structure is cascaded through 7-levels, with the high frequency signals from the last three levels used to compute the energy in those signals. Note that at each level, the amount of data is halved.

As a final step, all features from all channels, 66 in total, are combined into a single vector which is input into a patient-specific support vector machine (SVM). The SVM detects whether or not each window contains epileptiform activity. After three consecutive positive windows have been detected, a seizure is declared.

There are multiple places where Wishbone can partition this algorithm. If the entire application fits on the embedded node, then the data stream is reduced to only a feature vector—an enormous data reduction. But data is also reduced by each stage of processing on each channel, offering many intermediate points which are profitable to consider.

6.2 Acoustic Speech Detection

We used Wishbone to build a speech detection application that uses sampled audio to detect the presence of a person who is speaking near a sensor. The ultimate goal of such an application would be to perform speaker *identification* using a distributed network of microphones. For example, such a system could potentially be used to locate missing children in a museum by their voice, or to implement various security applications.

However, in our current work we are only concerned with speech *detection*, a precursor to the problem of speaker identification. In particular, our goal is to reduce the volume of data required to achieve speaker identification, by eliminating segments of data that probably do not contain speech and by summarizing the speech data through feature extraction.

Our implementation of speech detection and data reduction is based on Mel Frequency Cepstral Coefficients (MFCC), following the approach of prior work in the area. Recent work by Martin, et al. has shown that clustering analysis of MFCCs can be used to implement robust speech detection [14]. Another article by Saastamoinen, et al. describes an implementation of speaker identification on smartphones, based on applying learning algorithms to MFCC feature sets [19]. Based on this prior work, we chose to exercise our system using an implementation of MFCC feature extraction.

6.2.1 Mel Frequency Cepstral Coefficients

Mel Frequency Cepstral Coefficients (MFCC) are the most commonly used features in speech recognition algorithms. The MFCC feature stream represents a significant data reduction relative to the raw data stream.

To compute MFCCs, we first compute the spectrum of the signal, and then summarize it using a bank of overlapping filters that approximates the resolution of human aural perception. By discarding some of the data that is less relevant to human perception, the output of the filter bank represents a 4X data reduction relative to the original raw data. We then convert this reduced-

resolution spectrum from a linear to a log spectrum. Using a log spectrum makes it easier to separate convolutional components such as the excitation applied to the vocal tract and the impulse response of a reverberant environment, because transforms that are multiplicative in a linear spectrum are additive in a log spectrum.

Finally, we compute the MFCCs as the first 13 coefficients of the Discrete Cosine Transform (DCT) of this reduced log-spectrum. By analyzing the spectrum of a spectrum, the distribution of frequencies can be characterized at a variety of scales [6, 5].

6.2.2 Trade-offs in MFCC Extraction

The high level goal of Wishbone is to explore how a complex application written in a single high level language can be efficiently and easily distributed across a network of devices and support many different platforms. As such, the MFCC application presents an interesting challenge because for sensors with very limited resources there appears to be no perfect solution; rather, using Wishbone the application designer can explore different trade-offs in application performance.

These trade-offs arise because this algorithm squeezes a resource-limited device between two insoluble problems: not only is the network capacity insufficient to forward all the raw data back to a central point, but the CPU resources are also insufficient to extract the MFCCs in real time. If the application has any partitioning that fits the resource constraints, then the goal of Wishbone is to select the best partition, for example, lowest cost in terms of energy. If the application does not *fit* at its ideal data rate, ultimately, some data will be dropped on some target platforms. The objective in this case is to find a partitioning that minimizes this loss and therefore maximizes the *throughput*: the amount of input data successfully processed rather than dropped at the input sources or in the network.

6.2.3 Implementing Audio Capture

Some platforms, such as the iPhone and embedded-Linux platforms (such as the Gumstix), provide a complete and reliable hardware and software audio capture mechanism. On other platforms, including both TMotes and J2ME phones, capturing audio is more challenging.

On TMotes, we used a custom-built audio board to acquire audio. The board uses an electret microphone, four opamp stages, a programmable-gain amplifier , and a 2.5 V voltage reference. We have found that when the microphone was powered directly by the analog supply of the TMote, the audio board performed well when the mote was only acquiring audio, but was very noisy when the mote was communicating. The communication causes a slight modulation of the supply voltage, which gets amplified into significant noise. Using a separately regulated supply for the microphone removed this noise. The anti-aliasing filter is a simple RC filter; to better reject aliasing, the TMote samples at a high rate and applies a digital low-pass filter (filtering and decimating a 32 Ks/s stream down to 8 Ks/s works well). The amplified and filtered audio signal is presented to an ADC pin of the TMote's microcontroller, which has 12 bits of resolution. We use TinyOS 2.0 `ReadStream<uint16_t>` interface to the ADC, which uses double buffering to deliver arrays of samples to the application.

Phones naturally have built-in microphones and microphone amplifiers, but we have nonetheless encountered a number of problems using them as audio sensors. Many J2ME phones support the Mobile Media API (JSR-135), which may allow a program to record audio, video, and take photographs. Support for JSR-135 does not automatically imply support for audio or video recording or for taking snapshots. Even when audio recording is supported, the API permits only batch recording to an array or file (rather than a continuous stream) resulting in gaps.

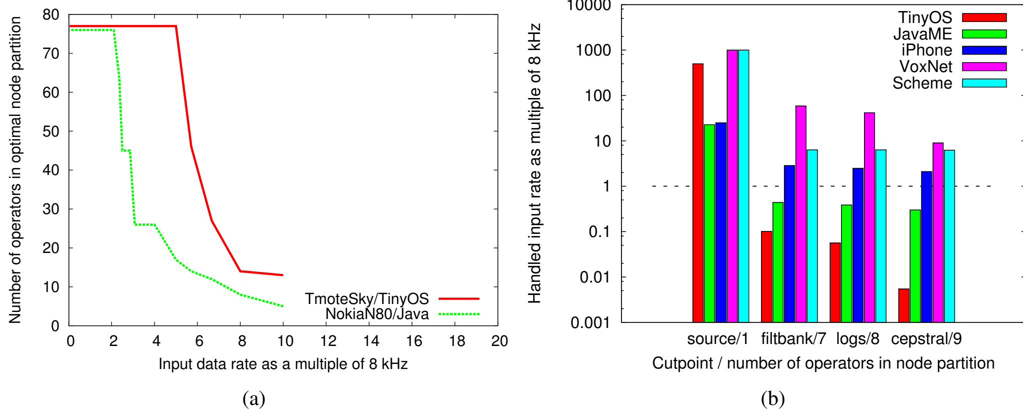
We ran into a bug on the Nokia N80: after recording audio segments for about 20 minutes, the JVM would crash. Other Nokia phones with the same operating system (Symbian S60 3rd Edition) exhibited the same bug. We worked around this bug using a simple Python script that runs on the phone and accepts requests to record audio or take a photograph through a TCP connection, returning the captured data also via TCP. The J2ME program acquires audio by sending a request to this Python script, which can record indefinitely without crashing.

The J2ME partition of the Wishbone program uses TCP to stream partially processed results to the server. When the J2ME connects, the phone asks the user to choose an IP access point; we normally use a WiFi connection, but the user can also choose a cellular IP connection. With any of these communication methods, dependence on user interaction presents a practical barrier to using phones in an autonomous sensor network. Yet these software limitations are incidental rather than fundamental, and should not pose a long-term problem.

7 Evaluation

In this section we evaluate the Wishbone system on the EEG and speech detection applications we discussed in Section 6. We focus on two key questions:

1. Can Wishbone efficiently select the best partitioning for a real application, across a range of hardware devices and data rates?
2. In an overload situation, can Wishbone effectively predict the effects of load-shedding and recommend a "good" partitioning?



(a) (b)

Figure 5: Relationship between partitioning and compute-bound sustainable data rates. On the left (a), a subset of the EEG application (one channel). The X axis shows a required data rate, the Y axis the number of operators in computed optimal node partition. On the right (b), the speaker detection application; we flip the axes due to the small number of viable cut-points. For each viable cut-point, we show the maximum data-rate supported on each hardware platform.

7.1 EEG Application

Our EEG application provides an opportunity to explore the scaling capability of our partitioning method. In particular, we look at our worst case scenario—partitioning all 22-channels (1412 operators). As the CPU budget increases, the optimal strategy for bandwidth reduction is to move more channels to the nodes. On our lower-power platforms, not all the channels can be processed on one node. The graph in Figure 5(a) shows partitioning results only for the *first* of 22 channels, where we vary the input data rate on the X axis and measure the number of operators that "fit" on different platforms. We ran `lp_solve` to derive a partitioning 2100 times, linearly varying the data rate to cover everything from "everything fits easily" to "nothing fits". To remove confounding factors, the objective function was configured to minimize network bandwidth subject to not exceeding CPU capacity ($\alpha = 0, \beta = 1$): that is, allow the CPU to be fully utilized (but not over-utilized). As we increased the data rate (moving right), fewer operators can fit within the CPU bounds on the node (moving down). The sloping lines show that every stage of processing yields data reductions.

The distribution of resulting execution times are depicted as two CDFs in Figure 6, where the x axis shows execution time in seconds, on a log scale. The top curve in Figure 6 shows that even for this large graph, `lp_solve` always found the optimal solution in under 90 seconds. The typical case was much better: 95 percent of the executions reached optimality in under 10 seconds. While this shows that an optimal solution is typically discovered in a reasonable length of time, that solution is not necessarily *known* to be optimal. If the solver is used to prove optimality, both worst and typical case runtimes become much longer, as shown by the lower CDF curve

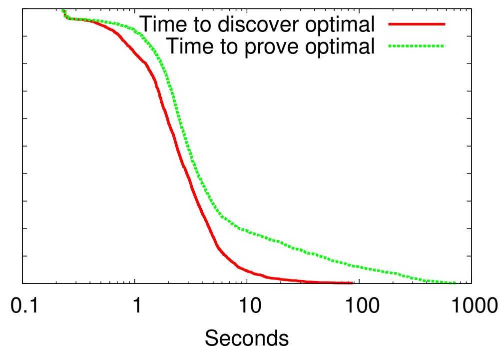


Figure 6: CDF of the time required for `lp_solve` to reach an optimal partitioning for the full EEG application (1412 operators), invoked 2100 times with data rates. The higher curve shows the execution time at which an optimal solution was found, while the lower curve shows the execution time required to prove that the solution is optimal. Execution times are from a 3.2 GHz Intel Xeon.

(yet still under 12 minutes). To address this, we can use an approximate lower bound to establish a termination condition based on estimating how close we are to the optimal solution.

7.2 Speech Detection Application

The speech detection application is a linear pipeline of only a dozen operators. Thus the optimization process for picking a cut point should be trivial—a brute force testing of all cut points will suffice. Nevertheless, this application's simplicity makes it easy to visualize and study, and the fact that the data rate it needs to process all data is unsustainable for TinyOS devices provides an op-

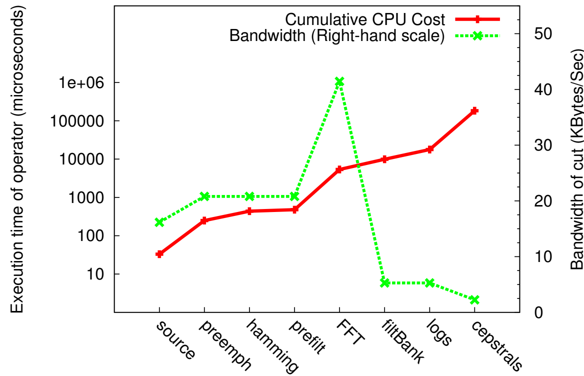


Figure 7: Data is reduced by processing, lowering bandwidth requirements, but increasing CPU requirements.

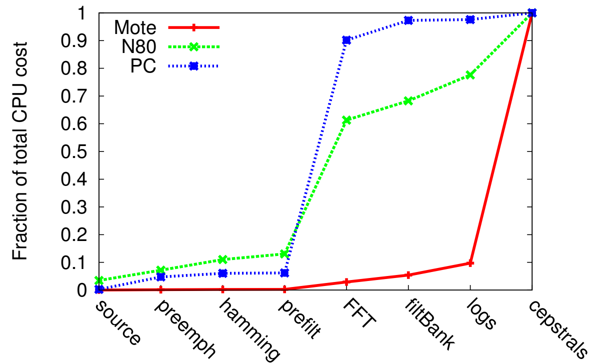


Figure 8: Normalized cumulative CPU usage for different platforms. Relative execution costs of operators vary greatly on the tested systems.

portunity to examine the other side of Wishbone's usage: what to do when the application doesn't fit.

In applying Wishbone to the development process for our speech detection application, we were able to quickly assess the performance on several different platforms. Figure 7 is a detailed visualization of the performance trade-offs, showing only the profiling results for TMote Sky (a TinyOS platform). In this figure, the X axis represents the linear pipeline of operators, and the Y axis represent profiling results. Each vertical impulse represents the number of microseconds of CPU time consumed by that operator per frame (left scale), while the line represents the number of bytes per second output by that operator. It is easy to visualize the trade-off between CPU cost and data rate. Each point on the X-axis represents a potential graph cut, where the sum of the red bars to the left provides the processing time per frame.

Thus, we see that the MFCC dataflow has multiple data-reducing steps. The algorithm must natively process 40 frames per second in real time, or one frame every 25 ms. The initial frame is 400 bytes; after applying the filter bank the frame data is reduced to 128 bytes, using 250 ms of processing time; after applying the DCT, the frame data is further reduced to 52 bytes, but using a total of 2 s of processing time. This structure means that although no split point can fit the application on the TMote at the full rate, we can achieve different CPU/bandwidth trade-offs by selecting different split points. Selecting a bad partitioning can result in retrieving no data, and the best "working" partition provides 20 times more data than the worst. Figure 5(b) shows an axes-flipped version of Figure 5(a): predicted data-rate as a function of the partition point. Only viable (data reducing) cutpoints are shown. Bars falling under the horizontal line indicate that the platform cannot be expected to keep up with the full (8 kHz) data rate.

As expected, the TMote is the worst performing platform, with the Nokia N80 performing only about twice as fast—surprisingly poor performance given that the N80 has a 32-bit processor running at 55X the clock rate of the TMote. This is due to the poor performance of the JVM implementation. The 412 MHz iPhone platform using GCC performed 3X worse than the 400 MHz Gumstix-based Linux platform; we believe that this is due to the frequency scaling of the processing kicking in to conserve power.

We can also visualize the relative performance of different operators across different platforms. For each platform processing the complete operator graph, Figure 8 shows the fraction of time consumed by each operator. If the time required for each operator scaled linearly with the overall speed of the platform, all three lines would be identical. However, the plot clearly shows that the different capabilities of the platforms result in very different relative operator costs. For example, on the TMote, floating point operations, which are used heavily in the `cepstrals` operator, are particularly slow. This shows that a model that assumes the relative costs of operators are the same on all platforms would mis-estimate costs by over an order of magnitude.

7.3 Wishbone Deployment

To validate the quality of the partitions selected by Wishbone, we deployed the speech detection application on a testbed of 20 TMote Sky nodes. We also used this deployment to validate the specific performance predictions that Wishbone makes using profiling data (e.g., if a combination of operators were predicted to use 15% CPU, did they?).

7.3.1 Network Profiling

The first step in deploying Wishbone is to profile the network topology in the deployment environment. It is important to note that simply changing the network size changes the available per-node bandwidth and thus requires re-profiling of the network and re-partitioning of the application. We run a portable WaveScript program that measures the goodput from each node in the network. This tool sends packets from all nodes at an iden-

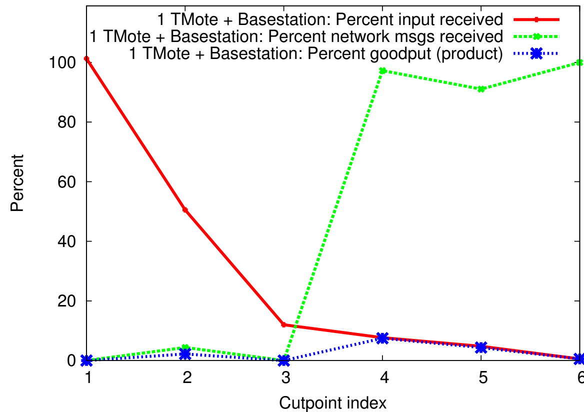


Figure 9: Loss rate measurements for a single TMote plus basestation across different partitionings. Lines show the percentage of input data processed, the percentage of network messages received, and the product of these: the goodput.

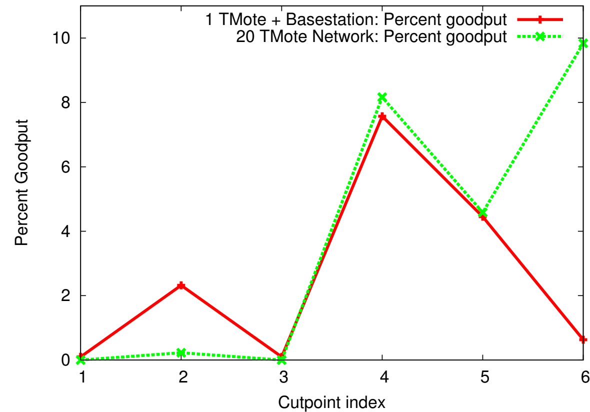


Figure 10: Goodput rates for a single TMote and for a network of 20 TMotes, over different partitionings when running on our TMote testbed.

tical rate, which gradually increases. For our 20 node testbed the resulting network profile is typical for TMote Sky devices: each node has a baseline packet drop rate that stays steady over a range of sending rates, and then at some point drops off dramatically as the network becomes excessively congested. Our profiling tool takes as input a target reception rate (e.g. 90%), and returns a maximum send rate (in msgs/sec and bytes/sec) that the network can maintain. For the range of sending rates within this upper bound the assumption mentioned in 4.3 holds—attempting to send more data does not result in fewer actual bytes of data received. Thus we are free to maximize the data rate within the upper bound provided by the network profiling tool, and thereby maximize total application throughput. This enables us to use binary search to find the the maximum sustainable data rate when we are in an overload situation.

To empirically verify that our computed partitions are optimal, we established a ground truth by exhaustively running the speech detection application at every cut point on our testbed. Figures 9 and 10 show the results for six relevant cutpoints, both for a single node network (testing an individual radio channel) and for the full 20 node TMote network. Wishbone counts missed input events and dropped network messages on a per-node basis. The relevant performance metric is the percentage of sample data that was fully processed to produce output. This is roughly the product of the fraction of data processed at sensor inputs, and the fraction of network messages that were successfully received.

Figure 9 shows the input event loss and network loss for the single TMote case, as well as the resulting goodput. On a single mote, the data rate is so high at early cutpoints that it drives the network reception rate to zero. At later cutpoints too much computation is done at the node and the CPU is busy for long periods, missing input events. In the middle, even a underpowered TMote can process 10% of sample windows. This is equivalent to polling for human speech four times a second—a reasonably useful configuration.

Figure 10 compares the goodput achieved with a single TMote and basestation to the case of a network of 20 TMotes. For the case of a single TMote, peak throughput rate occurs at the 4th cut point (filterbank), while for the whole TMote network in aggregate, peak throughput occurs at the 6th and final cut point (cepstral). As expected, the throughput line for the single mote tracks the whole line closely until cut point six. For a high-data rate application with no in-network aggregation, a many node network is limited by the same bottleneck as a network of only one node: the single link at the root of the routing tree. At the final cut point, the problem becomes compute bound and the aggregate power of the 20 TMote network makes it more potent than the single node.

We also ran the same test on an a Meraki Mini based on a low-end MIPS processor. While the Meraki has relatively little CPU power—only around 15 times that of the TMote—it has a WiFi radio interface with at least 10x higher bandwidth. Thus for the Meraki the optimal partitioning falls at cut point 1: send the raw data directly back to the server.

Having determined the optimal partitioning in our real deployment, we can now compare it to the recommendation of our partitioning algorithm. Doing this is slightly complex as the algorithm does not model message loss; instead, it keeps bandwidth usage under the user-supplied upper bound (using binary search to find the highest rate at which partitioning is possible), and minimizes the objective function. In the real network, lost packets may cause the actual delivered bandwidth to be somewhat less than expected by the profiler. Yet if we stay within a regime where network loss is roughly constant, this will reduce absolute performance but not change the optimal cut-point.

In this case, binary search found that the highest data rate for which a partition was possible (respecting network and CPU limits) was at 3 input events per second (with each event corresponding to a window of 200 audio samples). The optimal partitioning at that data rate[4] was in fact cut point 4, right after filterbank, as in the empirical data. Likewise, the computed partitions for the 20 node TMote network and single node Meraki test matched their empirical peaks, which gives us some confidence in the validity of the model.

In the future, we would like to further refine the precision of our CPU and network cost predictions. To use our ILP formulation we necessarily assume that both costs are additive—two operators using 10% CPU will together use 20%, and don't account for operating system overheads or processor involvement in network communication. For example, on the Gumstix ARM-linux platform the entire speaker detection application was predicted to use 11.5% CPU based on profiling data. When measured, the application used 15% CPU. Ideally we would like to take an automated approach to determining these scaling factors.

8 Related Work

First we overview other systems that, like Wishbone, automatically partition programs—either dynamically or statically—to run on multiple devices. Generally speaking, Wishbone differs from these existing systems by using a profile-driven approach to automatically derive a partitioning, as well as its support for diverse platforms.

The Pleiades/Kairos systems [13] statically partition a centralized C-like program into a collection of node-level nesC programs that run on motes. Pleiades is primarily concerned with the correct synchronization of shared state between nodes, including consistency, serializability, and deadlocks. Wishbone, in contrast, is concerned with high-rate shared-nothing data processing applications, where all nodes run the same code. Because Wishbone programs are composed of a series of discrete dataflow operators that repeatedly process streaming data, they are amenable to our profile-based approach for cost estimation. Finally, by constraining ourselves to a single cut point, we can generate optimal partitionings quickly, whereas Pleiades uses a heuristic partitioning approach to generate a number of cut points.

Triage [3] is a related system for "microservers" that act as gateways in sensor network applications. Triage's focus is on power conservation on such servers by using a lower-power device to wake a higher-power device based on a profile of expected power consumption and utility of data coming in over the sensor network. However, it does not attempt to automatically partition programs across the two device classes as Wishbone does.

In stream processing there has been substantial work looking at the problem of migrating operators at runtime [2, 18]. Dynamic partitioning is valuable in environments with variable network bandwidth, unpredictable load, but also comes with serious downsides in terms of runtime overheads. Also, by focusing on static partitioning, Wishbone is able to provide feedback to users at compile time about whether their program will "fit" their sensor platform and hardware configuration.

There has been related work in the context of traditional, non-sensor related distributed systems. For example, the Coign [11] system automatically partitions binary applications written using the Microsoft COM framework across several machines, with the goal of minimizing communication bandwidth. Like Wishbone, it uses a profile-driven approach. Unlike Wishbone, Coign does not formulate partitioning as an optimization problem, and only targets Windows PCs. Neubauer and Thiemann [15] present a similar framework for partitioning client-server programs. Automatic partitioning is also widely-used in high-performance computing, where it is usually applied to some underlying mesh, and in automatic layout of circuits. Finally, several systems, including JESSICA2 [25], MagnetOS [4], and cJVM [1], implement distributed Java virtual machines that appear as a single system. These systems must use runtime methods to load-balance threads between machines. The overheads on communication and synchronization are typically high, and only applications with a high ratio of computation to communication will scale effectively.

Tenet [9] proposes a two-tiered architecture with programs decomposed across sensors and a centralized server, much as in Wishbone. The VanGo system [10], which is related to Tenet, proposes a framework for building high data rate signal processing applications in sensor networks, similar to the applications that inspired our work on Wishbone. But VanGo is constrained to a linear chain of filters, does not support automatic partitioning, and runs only TinyOS code.

Marionette [24] and SpatialViews [17] use static partitioning of programs between sensor nodes and a server that is explicitly under the control of the programmer. These systems work by allowing users to invoke predefined handlers (written in, for example, nesC) from a high-level centralized program that runs on a server, but neither offers automatic partitioning.

Abstract Regions [22] and Hood [23] enable operations over clusters of nodes (or "regions") rather than single sensors. They allow data from multiple nodes to be combined and processed, but are targeted at coordinating sensors rather than stream processing.

[4] In this case with $\alpha = 0, \beta = 1$, although the linear combination in the objective function is not particularly important when we are maximizing data rate because we are saturating either CPU or bandwidth

9 Future Work/Conclusions

The model presented in this paper enables communication between embedded endpoints and a central server. But it would be straightforward to extend our model with a basic form of in-network aggregation: namely, tree-based aggregation that happens at every node in the network, useful, for example, for taking average sensor readings. This communication pattern would be exposed as a "reduce" operator that would reside in the logical node partition, but would implicitly take its input not just from streams within the local node, but from child nodes routing through it in an aggregation tree. The partitioning algorithm remains the same. If the reduce operator is assigned to the embedded node, aggregation happens in-network, otherwise all data is sent to the server.

Also, while our prototype implementation only supports networks of one type of node, the model can also handle certain kinds of mixed networks. A single logical node partition can take on different physical partitions at different nodes. This is accomplished simply by running the partitioning algorithm once for each type of node. The server would need to be engineered to deal with receiving results from the network at various stages of partial processing. In the future, mixed partitions may be desirable even for homogeneous networks. Varying wireless link quality can create a situation where each node should partitioned differently.

A more radical change would extend the model with multiple logical partitions corresponding to categories of devices. This opens up several design choices; for example, what communication relationship should the logical partitions should have? We have verified that we can use an ILP approach for a restricted *three tier* network architecture. (Motes communicate only to microservers, and microservers to the central server.) But going further would require revisiting the partitioning algorithm.

Acknowledgements: This work was supported by the NSF under grants 0520032, 0720079, and 0754662.

References

[1] Y. Aridor, M. Factor, and A. Teperman. cjvm: A single system image of a jvm on a cluster. In *Proc. ICPP*, 1999.

[2] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-Based Load Management in Federated Distributed Systems. In *Proc. USENIX NSDI*, San Francisco, CA, Mar. 2004.

[3] N. Banerjee, J. Sorber, M. D. Corner, S. Rollins, and D. Ganesan. Triage: balancing energy and quality of service in a microserver. In *MobiSys*, 2007.

[4] R. Barr, J. C. Bicket, D. S. Dantas, B. Du, T. W. D. Kim, B. Zhou, and E. G. Sirer. On the need for system-level support for ad hoc and sensor networks. *SIGOPS Operating Systems Review*, 36(2), 2002.

[5] R. A. Cole, J. Mariani, H. Uszkoreit, A. Zaenen, and V. Zue. Survey of the state of the art in human language technology, 1995.

[6] S. Davis and P. Mermelstein. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *IEEE Trans. on ASSP*, 28:357–366, 1980.

[7] E. B. et al. *Zoltan 3.0: Data Management Services for Parallel Applications; User's Guide*. Sandia National Laboratories, 2007. Tech. Report SAND2007-4749W.

[8] M. Garey, D. Johnson, , and L. Stockmeyer. Some simplified NP complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[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*, 2006.

[10] 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. In *SenSys*, pages 279–292, 2006.

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

[12] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20:359–392, 1998.

[13] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan. Reliable and efficient programming abstractions for wireless sensor networks. In *Proc. PLDI*, 2007.

[14] A. Martin, D. Charlet, and L. Mauuary. Robust speech/non-speech detection using LDA applied to MFCC. In *IEEE Intl. Conference on Acoustics, Speech, and Signal Processing*, pages 237–240, 2001.

[15] M. Neubauer and P. Thiemann. From sequential programs to multi-tier applications by program transformation. In *Proc. PLDI*, 2005.

[16] R. R. Newton, L. D. Girod, M. B. Craig, S. R. Madden, and J. G. Morrisett. Design and evaluation of a compiler for embedded stream programs. In *Proc. LCTES*, 2008.

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

[18] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.

[19] J. Saastamoinen, E. Karpov, V. Hautamki, and P. Frnti. Accuracy of MFCC-Based speaker recognition in series 60 device. *EURASIP Journal on Applied Signal Processing*, (17):2816–2827, 2005.

[20] A. Shoeb et al. Patient-Specific Seizure Onset. *Epilepsy and Behavior*, 5(4):483–498, 2004.

[21] A. Shoeb et al. Detecting Seizure Onset in the Ambulatory Setting: Demonstrating Feasibility. In *IEEE EMBS 2005*, September 2005.

[22] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *NSDI*, 2004.

[23] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *Proc. Mobisys*, 2004.

[24] 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 *Proc. IPSN*, 2006.

[25] W. Zhu, C.-L. Wang, and F. C. M. Lau. Jessica2: A distributed java virtual machine with transparent thread migration support. In *Proc. CLUSTER*, page 381, 2002.