MIT-CSAIL-TR-2007-017 March 14, 2007

Building Spatial Computers

Jonathan Bachrach and Jacob Beal

Building Spatial Computers

Jonathan Bachrach, Jacob Beal (MIT CSAIL)

1 Introduction

Spatial computing is an increasingly prevalent mode of computing, in which a program runs on a collection of devices spread through space whose ability to interact is strongly dependent on their geometry. Spatial computing arises in many domains: sensor networks, smart materials, swarm robotics, biofilms, and modular robotics, to name a few.

Programmability has been a major barrier to the deployment of spatial computing systems. We believe this is because present approaches, such as TinyOS[5] or Swarm[9], force the application programmer to describe the behavior of the devices rather than the behavior of the aggregate. Applications become entangled with the details of coordination and robustness, and as a result we see a plethora of niche solutions which often do not scale or compose well, let alone translate to other application domains.

We offer an alternate approach, in which the application programmer controls a single spatial computer which fills the space through which the devices are scattered. We accomplish this with two components. Proto Abstract Machine (PAM) is a machine model for a spatial computer, based on the *amorphous medium* abstraction and derived from the semantics of our language Proto. ProtoKernel is a distributed operating system which marshalls a collection of devices into a virtual spatial computer implementing PAM approximately. ProtoKernel has been demonstrated on platforms in three spatial computing domains: sensor networks, swarm robotics, and modular robotics.

This paper focusses on the challenges of implementing ProtoKernel given the resource scarcity endemic to spatial computing. We will briefly describe PAM and its advantages for programmability, then spend the bulk of the paper discussing how we handled challenges relating to execution, communication, and portability.

1.1 Related Work

Although there are many domain-specific programming models for spatial computers (Swarm[9], TinyOS[5], Paintable Computing[4], and CAs[7] to name a few) they all involve programming the behavior of the devices, rather than the behavior of the aggregate. A notable exception is CMost, the operating system for the CM-5[10], which allows operations on fields of devices, but assumes a fixed population of devices arranged in a grid.

The Proto language is described in [3], and its applicability to sensor networks in [1]. The amorphous medium abstraction was first proposed in [2].

2 Resource Scarcity

Spatial computers suffer from an ailment rare in computer science: a persistent resource scarcity not significantly affected by Moore's Law. The reason for this is simple: in addition to the familiar resources of memory, processing power, etc, a spatial computer has *expanse*, the area of the space filled with devices, and *density*, the number of devices per unit area.

Moore's Law doesn't just mean that computers get more powerful over time, it also means that computers of the same power become smaller, cheaper, and more energy-efficient. Smaller, cheaper, and more energy-efficient allows greater expanse and density, and proposed spatial computing applications often want computers with much larger expanse and density than is currently practical or economical. Thus, many generations of Moore's Law doubling could pass before an application sees a significant relaxation of the resource scarcity on individual devices.

The relative scarcity of resources on a spatial computer also differs significantly from a desktop machine, partly due to the nature of spatial computing applications and partly due to the devices commonly used in implementing them. In descending order of importance, these resource constraints are:

1. Memory and Storage—the amount per device is orders of magnitude lower than that of a desktop machine. For example, a Mica2 Mote has only 4K of RAM and 128K of flash memory.

2. Energy—devices are often powered by batteries or power harvesting. The rate of energy consumption thus translates directly into the lifespan or maintainence costs of a spatial computer.

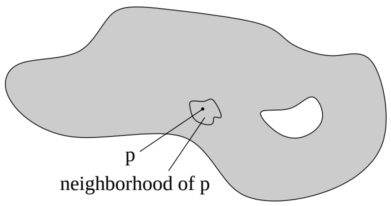

Figure 1: An amorphous medium is a space-filling material where every point is a device that computes using time-lagged state from a neighborhood of other nearby devices.

3. Bandwidth—although often superseded by memory or energy constraints, the long-distance coordination inherent in most spatial computing applications makes bandwidth an important consideration.

4. Processing Power—energy and bandwidth constraints usually slow down the speed of interaction enough that even an 8MHz processor is more than sufficient for most applications.

There are, of course, exceptions from platform to platform. This common order, however, will drive our implementation decisions.

3 Proto Abstract Machine

Our spatial computer abstraction is derived from the notion of an *amorphous medium*, a continuous space-filling material where every point is a computational device. Nearby devices share state—each device has some neighborhood in which it can access the state of other devices. Information propagates through the medium at a fixed rate, so neighbor state is time-lagged proportional to the distance it has travelled.

We cannot, of course, build an amorphous medium, but we can approximate it. Thinking about a continuous material will help us with both robustness and portability, since many differences between platforms and changes during execution can be subsumed into the single, simpler issue of approximation accuracy.

Accordingly, PAM begins by defining the *expanse* of a spatial computer to be the continuous region of space that it occupies. Data on the computer comes in the form of fields, which assign a value to every point in the computer and evolve over time. Finally, a computation is specified as a network of instructions that manipulates streams of field values.

For example, Figure 2 shows a program with six instructions that reads a light sensor and integrates

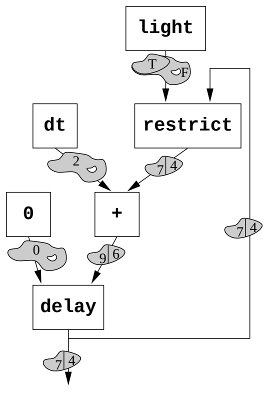

Figure 2: An example PAM program with six instructions, reporting how long each part of the computer has been detecting light.

time using a feedback loop to report how long each part of the computer has been detecting light. This model is closely related to the semantics of the Proto language for spatial computing; for more detailed explanation and examples of programming such a computer, see [3] or [1].

Why this dataflow model, rather than the familiar linear sequence of instructions on a serial machine? The fundamental reason is that information takes time to get from place to place within a spatial computer. This makes it impractical to synchronize execution across a computer of non-trivial expanse: the streaming dataflow model allows different parts of the computer to do different things, and synchronize weakly through the propagation of data across the expanse.

The instruction set is mostly composed of simple pointwise operations like literals and arithmetic. These are of little interest, as they translate directly to equivalent operations on individual devices. A few critical instructions, however, satisfy the special needs of spatial computing. These are:

- `restrict`, which limits the area in which a computation is evaluated to effect branching.
- `delay`, which time-shifts values, allowing state to be established in terms of feedback loops.
- three groups of neighborhood operators: one selects sets of neighborhood values (e.g. `nbr-range`, `nbr-val`), another operates on those sets, and the third summarizes neighborhoods down into a single value (e.g. `integral`, `exists`).

The job of ProtoKernel will be to implement the

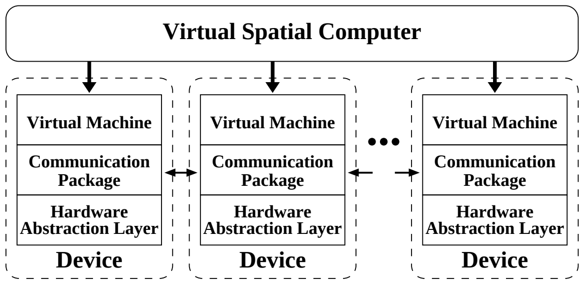

Figure 3: A ProtoKernel virtual spatial computer is a distributed system generated by a collection of devices running the ProtoKernel microkernel, which provides a virtual machine, neighbor communication package, and hardware abstraction layer.

dataflow network and these instructions using a microkernel which fits into individual devices (Figure 3).

4 Execution

The primary challenge for ProtoKernel to execute PAM code is fitting it into available memory. We accomplish this with a compiler that serializes the dataflow graph for execution on a virtual machine.

The virtual machine is a simple stack machine that executes scripts of 1-byte instructions, manipulating data tagged as one of several types—scalars, tuples, functions, and dead values. Execution happens in regular rounds, with one invocation of the script per round. The services needed during execution—sensing, actuation, communication, and power management—are handled by other components of ProtoKernel. Note that threading and memory management are not on that list: memory management is handled statically by the compiler, and computation is relatively plentiful enough to make threading unnecessary.

The compiler we have built is for Proto, a LISP-like functional language in which simple expressions are trivially translatable into PAM dataflow graphs. The compiler infers data types as it transforms code to graph, then simplifies the graph by inlining, folding constant sub-expressions, and removing empty structures. The compiler then walks the graph, translating it into a script that computes a single round of execution. In the process, `restrict` instructions become branch code, `delay` instructions have state storage allocated, and neighborhood operators export values (See Section 5).

During the walk the compiler determines the maximum sizes of stack, globals, state, and neighborhood data, and constructs loading and linking code. Because available memory varies from platform to platform, this stage requires a platform target, which sets the available memory and number of potential neighbors, allowing the compiler to determine statically whether a script is small enough to run.

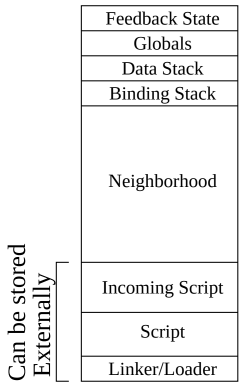

Figure 4: Memory is allocated into many different sections, with most going to the neighborhood. The script sections could be moved to external storage to save memory.

When a script is loaded onto a device, the link/load code is executed to allocate memory, initialize the program, and return a pointer to the start of the script; when a script is unloaded, exit code releases the memory. Memory is allocated into many different sections (See Figure 4) with the majority generally dedicated to the neighborhood. At present, all of these are kept in RAM but the script sections could be moved to slower storage.

5 Communication

Two actions that span space must be approximated by communication in the network of devices: loading programs onto the computer and instructions that use neighbor state. In both cases, communication is implemented in terms of local broadcast.

5.1 Loading Programs

A user should only need to touch a single device in order to load a program into the entire spatial computer. We implement this capability with a viral programming mechanism similar to those in [8] and [6].

Compiled programs are versioned and broken up into packets for transmission. Each device then broadcasts script digests stating which script packets is has, and sends script packets when a neighbor needs a packet it has. The compiled program also carries with it all of the necessary information for allocating memory and starting the computation running. This process is expensive in communication and power consumption, but occurs infrequently.

5.2 Neighborhood Support

Neighborhood operations are supported by a neighborhood communication module that maintains a best-effort table of its neighbors and any information that might be needed from them, similar to that in [4] and others. Neighborhood operations then walk through the table, calculating from the information it contains and combining it incrementally into an approximate summary value.

The operating system maintains the table by gathering export values during each round of execution. These are then transmitted halfway between rounds, in order to minimize the skew in data propagation rate due to phase differences between devices. Meanwhile, receipt of packets proceeds in the background.

Within the neighborhood memory section, structures encode neighbors' ID, timeout, position, timestamp, and data values. Because of memory considerations, a limited number of neighbor structures are allowed. This number is fixed for each platform, so that the behavior of code is not affected by the context in which it is called. Optimal handling of excess neighbors is a topic for future research. At present, neighbors are taken on a first come first serve basis and discarded if they have not been heard from within a timeout period.

The neighborhood mechanism also regulates power by an exponential backoff in frequency of transmission when data is not changing. The gradual backoff makes it less likely that dropped packets will will cause a neighbor to mistakenly drop a slowly transmitting device. Since broadcast is a significant fraction of power consumption, this backoff can save large amounts of power during periods of stability.

6 Portability

Much of our portability comes from the approximation of PAM by the virtual machine. The rest comes from a thin hardware abstraction layer in ProtoKernel whose device interface provides sensing, actuation, communication, and geometry. This allows the same code to run on different platforms and makes it easier to test code in simulation.

Sensors and actuators are implemented as instructions that call a platform-specific handler for the device, with default results provided for unavailable devices. The handler for actuators must resolve conflicts for multiple actuations during a single round; at present, we have only used lowest-and-rightmost precedence, though any method is acceptable as long as it is consistent across platforms.

The device interface also allows debugging support through virtual sensors and actuators: at present, there is a probe device that exposes values, a peek/poke interface that manipulates sensor and geometry information, and an interface for breakpoints and communication tracing.

Neighborhood support depends on communication and geometry. Communication simply provides the local broadcast interface needed for neighborhood support. Geometric information, in the form of relative coordinates, area, time lag, etc, is derived from whatever localization hardware the platform provides. This can be as crude as communication range, or as sophisticated as precise rangefinding and global coordinates.

At present, ProtoKernel supports three platforms besides the simulator.

Mica2 Motes ProtoKernel runs on Mica2 Motes [5] on top of TinyOS. The Mica2 has an 16MHz 8-bit processor, 4K of RAM, and 128K of flash memory. The hardware abstraction layer uses TinyOS's radio communication stack for transceiving data and script in 32-byte packets at 19.2kbps, and its timer, sensor, and actuation modules for implementing the rest of the hardware layer. In this implementation, we assumed the presence of a global positioning service and send global coordinates to neighbors who then calculate relative coordinates.

SwarmBots ProtoKernel runs on McLurkin and iRobot's swarm robots [8] on top of their embedded threaded operating system. Resources are plentiful: 40MHz 32-bit processor, 648K of RAM, 3MB of flash memory, and 125Kbps infrared communication that also provides ranging and direction information. The operating system provides its own incompatible neighborhood system, which we use as a blackboard system to simulate local broadcast.

Topobo The Topobo [11] modular robotics platform, in which devices are physically connected using passive linkage components connected from one motor to a join point on another device. This platform has the least resources: a 16MHz 8-bit processor, 2K RAM, 32K of flash memory, four 9600kbps wired connections to its neighbors, and a user interface of a button and two LEDs on each device. There is no native Topobo operating system, so the ProtoKernel implementation includes low-level modules to drive

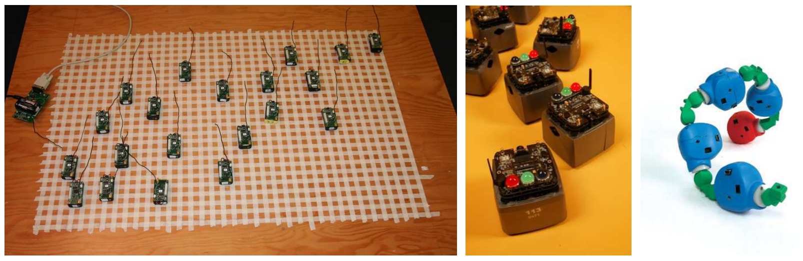
(a) Mica2 Motes (b) SwarmBots (c) Topobo

Figure 5: ProtoKernel currently supports platforms for sensor networks (a), swarm robotics (b), and modular robotics (c). (Photo credit: (b) James McLurkin and Swaine Photography, (c) Hayes Raffle and Amanda Parkes)

the motor, run the user interface, and communicate packets. Geometric information is completely available only through connectivity.

7 Conclusion

Spatial computing poses special challenges to programmability due to problems of coordination, robustness, and persistent resource scarcity. The problems of robustness and coordination can be simplified by writing programs for PAM, a spatial computer model based on the amorphous medium abstraction. This spatial computer is then implemented approximately by ProtoKernel and a compiler that transforms programs to minimize resource consumption.

This approach allows us to run ProtoKernel on platforms with severe resource limitations in different spatial computing domains: Mica2 Motes for sensor networks, SwarmBots for swarm robotics, and Topobo for modular robotics. In future work, we look to expand the number of platforms supported and further improve resource usage with techniques such as compressed numbers and application-specific instruction sets.

8 Acknowledgements

This work was funded under NSF grant CCF-0621897. We would like to thank James McLurkin, Hayes Raffle, Anthony Grue, and Mark Tobenkin for their help in getting Proto to run on the SwarmBots, Topobo, and Mote platforms respectively.

References

[1] J. Bachrach and J. Beal. Programming a sensor network as an amorphous medium. In *DCOSS 2006 Posters*, June 2006.

[2] J. Beal. Programming an amorphous computational medium. In *Unconventional Programming Paradigms International Workshop*, 2004.

[3] J. Beal and J. Bachrach. Infrastructure for engineered emergence in sensor/actuator networks. *IEEE Intelligent Systems*, pages 10–19, March/April 2006.

[4] W. Butera. *Programming a Paintable Computer*. PhD thesis, MIT, 2002.

[5] J. Hill, R. Szewcyk, A. Woo, D. Culler, S. Hollar, and K. Pister. System architecture directions for networked sensors. In *ASPLOS*, November 2000.

[6] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code progagation and maintenance in wireless sensor networks. In *NSDI*, 2004.

[7] N. Margolus. CAM-8: A computer architecture based on cellular automata. In *Pattern Formation and Lattice-Gas Automata*, 1993.

[8] J. McLurkin. Stupid robot tricks: A behavior-based distributed algorithm library for programming swarms of robots. Master's thesis, MIT, 2004.

[9] N. Minar, R. Burkhart, C. Langton, and M. Askenazi. The swarm simulation system, a toolkit for building multi-agent simulations. Technical Report Working Paper 96-06-042, Santa Fe Institute, 1996.

[10] J. Palmer and J. G.L. Steele. Connection machine model cm-5 system overview. In *Fourth Symposium on the Frontiers of Massively Parallel Computation*, 1992.

[11] H. Raffle, A. Parkes, and H. Ishii. Topobo: A constructive assembly system with kinetic memory. *CHI*, 2004.