A Modular and Extensible Macroprogramming Compiler

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

ABSTRACT

Translation of a macroprogram to node-level microprograms is a complex and challenging task for a compiler. Developing of a robust macroprogramming compiler framework poses a unique challenge because users need to create optimized function decompositions, in a modular way, such that they can be composed with other compiler functionality such as optimization and debugging. This differs from most compilers because they typically employ a fixed set of code optimizations for a particular target platform. We present a macroprogramming compiler for MacroLab which has a modular architecture that provides users with a two layer interface: one to tackle user created optimized function decompositions and another to handle low-level compiler modifications. Our results show that the introduction of new decompositions and compiler optimizations can be accomplished with a small number of lines of code and typically written in a matter of hours.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization, Retargetable compilers*; D.2.11 [**Software Architectures**]: *Domain-specific architectures*

General Terms

Design, Languages, Performance

Keywords

Compiler, Function Library, Macroprogramming, Wireless Embedded Networks

1. INTRODUCTION

A *compiler* is responsible for transforming source code into a machine-level binary for a particular target platform, typically a processor or micro-controller. A *macroprogram* compiler (Figure 1) translates a program describing network-wide operations into node-level actions. It operates in much the same way as a traditional compiler except that the target platform is a system of independent devices. Wireless embedded networks have traditionally relied on programmers to define node-level actions in order to program a system. A macroprogramming compiler allows the end-user to write a high-level application (macroprogram) and it will convert it into relevant node-level actions.

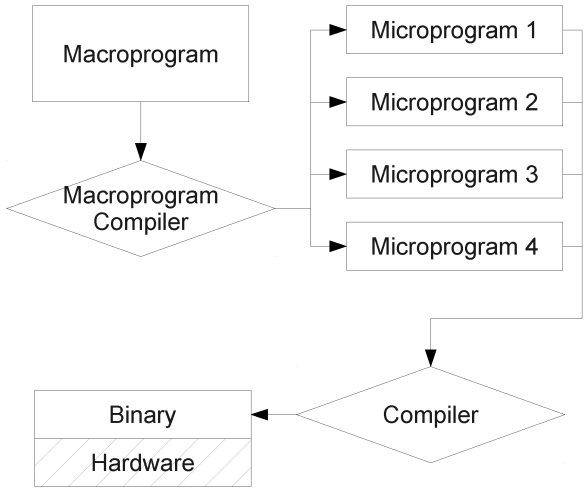

Figure 1. A typical macroprogramming compiler. An application is broken down into different versions which are compiler by another compiler into a binary that runs on the hardware.

The development of a robust macroprogramming compiler framework poses a unique challenge because users need to create optimized function decompositions, in a modular way, so that they can still be composed with other compiler functionality such as optimization and debugging. This is a contrast to most compilers because they typically employ a fixed set of code optimizations for a particular target platform. Since our target platform is a wireless embedded network based in the physical world, the type of optimizations and function decompositions will vary according to the environment.

In this paper, we present a macroprogramming compiler for MacroLab [7] which has a modular architecture and provides users with a two layer interface to tackle a need for user created, optimized function decompositions that work well with varying compiler optimizations. This approach can be generalized for other programming abstractions. At a high-level, a user specifies optimized function decompositions as part of a *function library*. This library contains

SESENA '10, May 3 2010, Cape Town, South Africa

implementations of common macroprogram functions and can be extended to support any user-defined functionality. The low-level interface allows modification and addition of compiler optimizations or stages. This two interface design allows the compiler to apply existing optimizations such as adding debugging support, allowing heterogeneous hardware configurations, or optimizing for a specific topology, in the presence of new and unknown distributed operations. By separating out the optimized distributed function decompositions from low-level compiler optimizations, we create an expandable and modular compiler architecture.

The remainder of this paper is structured as follows. First, we discuss how distributed operations are added to the function library in order to increase functionality. Examples serve to illustrate how easy it is to create these functions. Next, the compiler structure is discussed along with an example that shows how the pieces are put together. Finally, each of these sections is evaluated as a series of case studies that illustrate the effectiveness and extensibility of our approach.

2. ADDING DISTRIBUTED OPERATIONS

Our compiler relies on a *function library* which contains all implementations of functions used to write a macroprogram. This is similar to a traditional function library except that each function has multiple implementations. For instance, the calculation of the maximum light value in the network can occur using in-network aggregation or at a centralized location such as a base station. The function library contains both implementations of *max*. The compiler uses a technique similar to *polymorphism* to decide among various implementations of a function based on the expected input and output type. In the case of macroprograms, the input and output types refer to how the data is stored, for instance distributed across the network or at a centralized location. The function library deviates from polymorphism by allowing different implementations to have the same signature. In this case, the compiler decides based on the topology which frees the programmer from having worry about data movement.

The function library is configured as a directory structure (Figure 2) where the first level of folders specify the names of functions. These are the available for use in the macroprogram. Within each directory there are either files or subdirectories. First, files following the form $func*.m$ specify macrocode functions and contain macrocode that is inserted at the appropriate time during compilation. A second option is a directory that following the form, $func*$. Contained within these directories are node- and base-specific implementations. A file *sig.m* contains the function signature which specifies a function name, typed input parameters, and a typed return result. This file is used by the compiler to determine when to apply code located within the directory. Unlike macrocode, the rest of the function contains untyped code (microcode) and will be converted into a binary at the end of the compilation process.

Structuring the function library as a set of files and directories makes it easy for the user to extend its functionality. New functions can be added by introducing new directories and filling them with the appropriate files: allowing the introduction of both macrocode and microcode. If the programmer desires, the function library can be customized for particular network scenario or modified to ensure that the current deployment will execute efficiently. Over time, our function library will grow and the capabilities of systems built in MacroLab will increase.

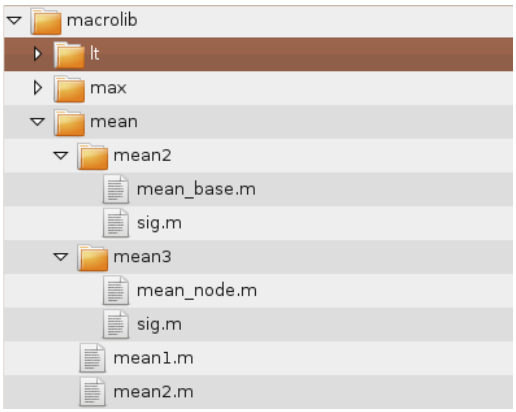

Figure 2. Function-library directory structure: First level directories specify function names while second level directories and files provide implementations.

To illustrate the capabilities of our function library, we will examine three different implementations of the *mean* operation. First, we will look at a macrocode example. Next, base-station specific code will be examined. Finally, node-specific microcode is shown.

Macrocode – We will examine each mechanism starting with macrocode specific functionality. For example, this code

```
1 function centralized result = mean(distributed
      value)
2 centralized centVect = assign (distributed value)
3 centralized result = mean(centralized centVect)
```

is a macrocode implementation of the *mean* operation. The first line specifies the return type, function name, and all parameters with associated types. In this case, there is a single distributed value. Line 2 assigns the distributed vector to a centralized vector which causes messages to be sent between nodes. The last line computes the mean and returns the result in a centralized form. Both *assign* (line 2) and *mean* (line 3) are not primitive functions and refer to code located within the library. The *mean* operation in line 3 is not a case of self-referencing code because the data type of the input parameter is different.

Base-station code – Another category of code that is stored within the function library is base-station specific microcode. For example, this code

```
1 function centralized result = mean(centralized
      value)
2 temp = [];
3 for i = 1:length(value)
4   temp(end+1) = value(i).data;
5 end
6 tempMean = mean(temp);
7 result = MacroElement;
8 result.data = tempMean;
9 result.nodeID = -1;
```

illustrates a *mean* operation; however, it is written as Matlab code. This is because our base-station is based on a Matlab interface. The only typed variables are located in the function definition and are for identification of the function signature. Types are removed in the final version of the

code through an in-lining process.

Node-specific code – The final category of code within the function library is node-specific microcode. It is written as a subset of Matlab called Embedded Matlab (EML) and can be transformed by existing compilers into C-code that we utilize on motes. The same *mean* operation is shown

```
1  function result = mean(value)
2  result = zeros(1,2,'uint16');
3  tempMean = uint16(0);
4  count = uint16(1);
5  tempMean = value(1,2);
6  for y = 2:MAX_NUM_NODES
7    if (value(y,1)>0)
8      tempMean = tempMean + value(y,2);
9      count = count + uint16(1)
10   end
11 end
12 myID = uint16(0);
13 myID = eml.ceval('getID');
14 result(1,2) = tempMean/count;
15 result(1,1) = myID;
```

and unlike the prior examples, it contains no type information within the header. All of this information is contained within the *sig.m* file. This version of the function operates over a *neighbor* vector as an input parameter (value) and produces a *distributed* value (result).

Having a function library that contains all three types of implementations allows the compiler to decide on a data representation for a particular scenario. This flexibility is how application efficiency is achieved. The compiler has the ability to adjust these data representation to work around particular scenarios where functionality is missing from the library.

3. ADDING COMPILER FUNCTIONALITY

The first version of the MacroLab compiler was implemented as a simple code translator in the manner of early macroprogramming systems. This enabled development of a compiler in a few lines of code (1470). This was a proof of concept implementation. Version 1 has the downside of being very difficult to extend as separate translation rules have to be written for all possible source to binary mappings. This resulted in it supporting a very small class of applications.

In version 2, we extended the compiler (7437 LOC) in the spirit of CIL [11]. This enabled easier code modification through CST modifications which increased the class of applications the compiler could support. It also allowed the compiler to be extended easily. Version 3 of the compiler cleaned up the source code by defining interfaces between modules in a consistent manner so that the compiler could be released for others to use and modify. This resulted in a reduction in code size (6390 LOC) as certain modules became more reusable. Clear interface definitions allowed the concept of stages where extensions to the compiler could be made by the addition of a stage instead of having to modify the code in various modules. For instance, the compiler was extended to support debugging by porting our existing code into a single stage with fewer than 50 lines of code in approximately 3 hours. The majority of this time was spent understanding the new infrastructure and figuring out how to utilize the stage system.

The compiler is composed of *stages*: classes that perform an action relevant to compilation. A stage could read information from files, construct an Abstract Syntax Tree (AST), manipulate an AST, or create new files. Each stage inherits from a base class and is identified by a unique name. A stage has a requirements list composed of the names of stage on which it depends. The code executed in each stage is defined in an *execute* method. The following code is an example of a stage that replaces all binary operations with their respective function-call implementations:

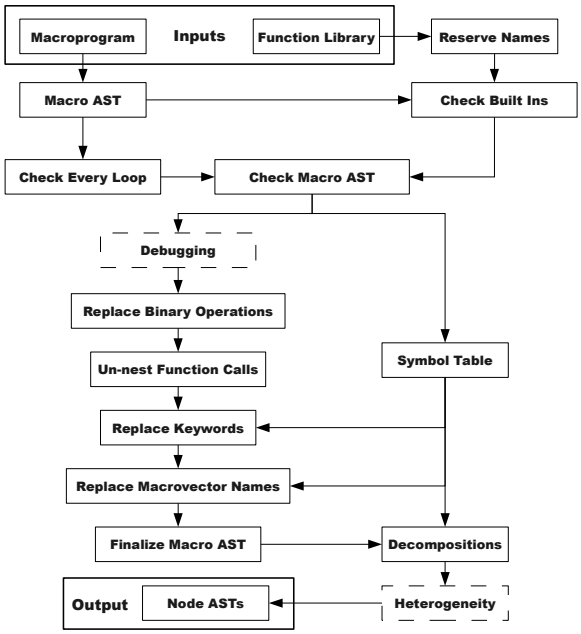

Figure 3. Block diagram of the compiler stages. It includes all of the inputs along with the stages used to convert macrocode into node specific microcode. The stages Debugging and Heterogeneity, denoted by dashed lines, are new additions to the compiler and are described later.

```
1 class BinaryOps(stage):
2     name = REPLACED_BINOPS
3     requires = set([CHECKED_MACRO_AST])
4
5     def execute(self, resources):
6         macroast = resources[MACRO_AST]
7         visitor = BinOpVisitor()
8         walkast(macroast, visitor)
9         return True
```

A compiler is implemented by wiring together stages to form a *configuration* (Figure 3). Implementing compilers as a configuration of dependent stages allows their capabilities to be dynamically changed by including or removing stages. For instance, debugging can be enabled by adding its stage using a command-line switch.

A dependence graph is constructed by reading a configuration for stage dependencies and executed in a breadth-first manner. This allows as many stages as possible to be executed in the event of a stage failure, allowing the failure to be localized more precisely.

A common operation in the *execute* method is the manipulation of AST nodes. The compiler framework makes AST manipulations easier by providing an *AST walker* that can be used to traverse an AST. It uses an extensible class named *visitor* that has methods to manipulate all possible AST nodes. Classes derived from *visitor* can override these methods in order to process nodes encountered by the

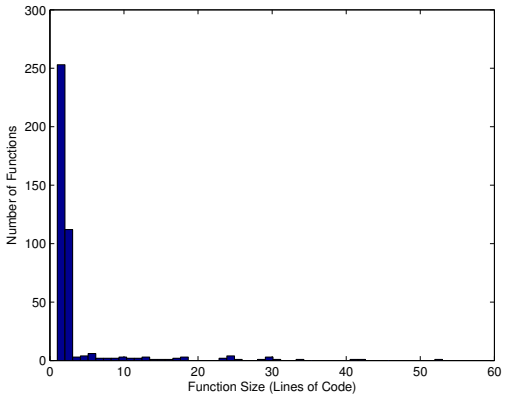

Figure 4. Most functions in the library require less than 5 lines of code and only a small fraction require more than 20-30 lines.

walker. The compiler framework also provides the ability to search for AST sub-trees using a *structured search*.

The structured search is carried out by defining the subtree to be found. For instance, the following structure defines a search for all string constants in the code:

```
1 t = colonop({'type':'CONST',
2               'ctype':'STRING',
3               'mvals':{'val':'value'}})
```

To illustrate the capabilities of our compiler framework we show how the visitor class and stages are used to translate binary operations. Many stages are relatively easy to construct as a series of modifications to an AST. The first stage during the compilation process is to change all binary operations, such as `A = B + C`, into function calls, such as `A = add(B, C)`, allowing the compiler to utilize user provided function implementations. This stage instantiates a visitor class named *Binary Operation Visitor*, then traverses the AST using the AST walker in order to modify all binary operations using the overloaded visitor.

4. EVALUATION

Our evaluation of adding distributed functions and compiler functionality is expressed as a series of case studies. First, we will show how to add new methods into the function library and examine the three major types of functions that can be written. Second, we explore the addition of two new modules into the compiler infrastructure: debugging and heterogeneity support. These modules were each written in less than a day. Each case study will showcase the ease with which extensions have been made to the infrastructure.

4.1 Adding New Distributed Operations

Because of the many ways data can be represented within a wireless embedded network, operations can be carried out in different ways. In some cases, it makes sense to write macrocode in order to transform the data storage location into something that can be processed by microcode. Other cases allow for the microcode to be written directly. The typing associated with each implementation allows the compiler to determine the appropriate implementation.

Figure 4 shows the distribution of functions in the library in terms of lines of code. The average size of existing functions is five lines of code. The most complicated function of the 124 implemented is 53 lines of code with most functions having four different implementations.

Function Name	Lines of Code	Type
mean1	3	Macrocode
mean2	3	Macrocode
mean3	10	Node code
mean4	15	Base code
find1	27	Node code
find2	42	Node code
find3	30	Base code
assign1	23	Node code
assign2	3	Base code
contourf1	34	Base code
cameradisplay1	22	Base code

Table 1. Lines of code and type of a representative sample of functions in the function library.

Stage	Lines of Code
AST Walker	337
Visitor	65
Matched Search	60
Reserved Names	10
Macro AST	7
Check Built Ins	12
Check Every Loop	8
Check Macro AST	5
Debugging	46
Replace Binary Operations	8
Un-nest Function Calls	8
Symbol Table	7
Replace Keywords	28
Replace Macrovector Names	10
Finalize Macro AST	12
Decompositions	27

Table 2. Number of lines of code for key pieces of the compiler infrastructure and stages.

Table 1 show some example functions in the library. The `mean` function, for instance, has four implementations depending on input and output types.

4.2 Adding New Compiler Functionality

We show how the compiler was extended with two case studies. The first case study describes how the compiler was extended to support the MDB [15] debugger. The second case study describes how the compiler, which initially supported only homogeneous networks, was extended to support heterogeneous devices.

Table 2 lists the sizes of some of the major functions in the compiler infrastructure and the sizes of the stages that composed the MacroLab compiler. The lines of code illustrate the module complexity.

4.2.1 Debugging

While we implemented support for the MDB [15] debugger, the compiler can be extended to support any other debugger equally easily. Modifications to the compiler include debugging support which involves inserting functions into existing application code in order to cause debugging logic to occur.

MDB is a replay debugger which requires logs to be collected of all variable writes and all executed lines of code.

This is accomplished by inserting a new statement between all existing lines in the AST. These lines follow a simple template that tells the system to store what the last written variable is or the current line number (obtained from the annotations to the AST). The inserted line of code is a function in the library and later, during the code generation phase, appropriate microcode is inserted to properly log the data.

This approach is not restricted to logging data and more general debugging approaches are possible. Instead of just storing the data to flash, the inserted function could include options to control the execution of the application. Enabling traditional debugging commands such as *start*,*stop*, *step*, *breakpoint* and *conditional breakpoints*. The compiler inserts an appropriate trampoline to transition execution into the debugging code.

Debugging support is a highly desired feature for distributed systems and especially important in wireless embedded network systems. The large number of devices and numerous data sources necessitate a set of debugging tools. Compiler support for debugging is key when making code transformations. Managing and mapping code transformations makes it easy to incorporate debugging support. We foresee may other debugging techniques for the future of macroprogramming systems and a compiler that easily allows the inclusion of these techniques will promote their rapid development and introduction into existing systems.

4.2.2 Heterogeneity

One of the challenges with compiling macroprograms is handling heterogeneous hardware capabilities, which is a separate issue from handling different hardware types. Handling this is a job for the microprogram to binary compiler which is outside the scope of our macroprogram compiler. The issue of heterogeneous hardware involves having different sensor and actuator configurations on nodes. It would be erroneous to actuate a temperature sensor on a device that only can sense light values or try to actuate a camera movement on a node without a camera.

A capability list specifies the types of hardware contained on each of the platforms. For example, this file

```
<node>
  <name>Room Monitor</name>
  <sensors>
    <sensor>temperature</sensor>
    <sensor>humidity</sensor>
    <sensor>light</sensor>
  </sensors>
  <actuators>
    <actuator>lightcontrol</actuator>
  </actuators>
</node>
```

specifies a sensor node called *Room Monitor* which contains three sensors, *temperature*, *humidity*, and *light*. It also contains an actuator, *lightcontrol*, which controls the room lights. A capability list is needed for each type of hardware configuration that will be deployed for an application. The compiler produces a set of microcode files for each configuration.

We implemented another stage that reads in the capability lists and processes an AST which generates appropriate code for each configuration. First, we will discuss the general process for how code is eliminated followed by all of the necessary pieces to maintain proper code execution.

Capability lists are utilized as a way to eliminate or restrict which portions of code are placed on each device. We start by processing a user specified macroprogram and eliminate all unnecessary lines of code in order to support the hardware. First, the AST is processed to remove references to hardware (sensors or actuators) not specified in the capability list. This is easily done because our programming abstraction requires the user to create sensor or actuator vectors and utilize the hardware names as part of a configuration. Each line of the AST is processed and if the right-hand side contains any variables that had previously been removed or are not available because of a hardware configuration, it is removed and the left-hand side identifier is recorded. This process is repeated until all possible variables and lines are eliminated and we are left with code containing no missing dependencies or invalid hardware accesses.

This prototype implementation is not without limitations. We are currently working on implementing support the remaining pieces necessary for heterogeneous devices. A number of stages are required beyond a simple line removal algorithm. A control flow graph is necessary and allows the compiler to see how each variable is used. This is complicated because of the way control flow is transfered between separate micro-ASTs. We are concerned about how the program executes as it runs on each node and the system as a whole.

Once a control flow graph is constructed, dependency analysis will be used to eliminate variables that are not part of an actuation. In our case, an actuation is any event that utilizes a variable in order to perform an action. In our macroprogramming abstraction, we are looking for the set of variables that determine actuator events or sent messages.

Unlike most distributed systems, we are constantly dealing with many different hardware platforms and configurations as a targeted system. A macroprogram will be written as if all devices contained all hardware. It is the responsibility of the compiler to properly break down the code and only place relevant pieces on devices.

The compiler presented in [7] supports only homogeneous networks where all the nodes have the same capabilities. Since then it has been extended to support heterogeneous networks. A new stage was introduced to support heterogeneous networks. This stage consists of about 100 lines of code written in a day. This is a 1.6% increase in the number of lines of code. We expect that full support of heterogeneity will require approximately 1000 lines of code for handling the control transfers and actuation events along with a dependency analysis.

5. RELATED WORK

A number of macroprogramming systems have been proposed over the last few years [1, 5, 9, 10, 12, 16–18]. They provide various abstractions for networks of devices ranging from a database, group, or data streams. Most of these macroprogramming systems rely on compilers to translate the source code into node-level binaries and the compilation strategies adopted have gotten more complicated as time goes on. For instance, early macroprogramming systems such as Marionette [19] and Region Streams [13] used simple code translating scripts while newer systems such as Pleiades [8] and MacroLab [7] use multi-stage compilers that perform advanced program analysis. As these compilers

have become more complicated, they have also become more difficult to modify and extend. Another system only targets node configurations for a data-driven applications [14]. This system only handles node configurations and requires a programmer to provide the necessary imperative processing code. To the best of our knowledge no other compiler provides the ease of extensibility at multiple levels of the compilation process that our compiler does. We believe this is an important feature of any compiler that is targeted towards the rapidly changing field of wireless embedded networks.

In addition to the macroprogramming systems that have been proposed for wireless embedded networks, the field of high performance computing has yielded a number of compilers that allow programmers to write imperative programs which distributed across multiple processors [2–4, 6]. These compilers differ from macroprogramming compilers by relying on programmers to annotate code to indicate how data and programs should be distributed across devices. While this maybe acceptable for a limited number of homogeneous processors, it is much more difficult on a wireless embedded network composed of large number of heterogeneous devices.

6. CONCLUSION

Macroprogramming languages make programming wireless embedded networks easier by allowing a programmer to define an application in terms of a high-level programming abstraction and relying on a compiler to translate into node-level binaries. In addition to shielding the programmer from the low-level details, the high-level abstraction of macroprogramming languages provide a compiler the flexibility to optimize how an application executes on a network. Having an easily extensible compiler allows end-users to add new optimizations for their particular deployments or introduce optimizations have not been implemented.

In this paper, we present a modular macroprogramming compiler that is easy to extend. Extensions are possible at two levels: (i) the addition of distributed functions, and (ii) the addition of compiler functionality such as support for debugging. Most end-users would use the compiler as it is but some may want to include additional functions. For instance, controlling a new hardware device such as a camera or additional implementations of a particular function are two such examples. Our compiler makes it easy to make changes by writing a few lines of code in a set of files and saving them in a structured directory. Other users may want to change the actual behavior of the compiler, by introducing new features such as static code analysis and our compiler allows this to be done through stages and associated functionality.

7. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 0845761.

8. REFERENCES

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

[2] P. Banerjee, J. A. Chandy, M. Gupta, E. W. H. IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The paradigm compiler for distributed-memory multicomputers. *Computer*, 28(10):37–47, 1995.

[3] S. Benkner. Vfc: the vienna fortran compiler. *Sci. Program.*, 7(1):67–81, Jan. 1999.

[4] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with polaris. *Computer*, 29(12):78–82, 1996.

[5] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairos. In *Distributed Computing in Sensor Systems*, pages 126–140. 2005.

[6] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29(12):84–89, 1996.

[7] T. Hnat, T. Sookoor, P. Hooimeijer, W. Weimer, and K. Whitehouse. Macrolab: a vector-based macroprogramming framework for cyber-physical systems. In *SenSys*, 2008.

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

[9] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30:122–173, 2005.

[10] G. Mainland, G. Morrisett, and M. Welsh. Flask: staged functional programming for sensor networks. *SIGPLAN Not.*, 2008.

[11] G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. *Lecture Notes in Computer Science*, pages 213–228, 2002.

[12] R. Newton, G. Morrisett, and M. Welsh. The regiment macroprogramming system. In *IPSN*. ACM Press New York, NY, USA, 2007.

[13] R. Newton and M. Welsh. Region streams: functional macroprogramming for sensor networks. In *DMSN '04: Proceeedings of the 1st international workshop on Data management for sensor networks*, pages 78–87, New York, NY, USA, August 2004. ACM Press.

[14] A. Pathak, L. Mottola, A. Bakshi, V. Prasanna, and G. Picco. A compilation framework for macroprogramming networked sensors. *Lecture Notes in Computer Science*, 4549:189, 2007.

[15] T. Sookoor, T. Hnat, P. Hooimeijer, W. Weimer, and K. Whitehouse. Macrodebugging: Providing abstract views of system state. In *SenSys*, 2009.

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

[17] K. Whitehouse, J. Liu, and F. Zhao. Semantic streams: a framework for composable inference over sensor data. In *EWSN*, 2006.

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

[19] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: using rpc for interactive development and debugging of wireless embedded networks. In *IPSN*, 2006.