Compiling a Gesture Recognition Application for a Low-Power Spatial Architecture

Phitchaya Mangpo Phothilimthana
University of California, Berkeley, USA
mangpo@eecs.berkeley.edu

Michael Schuldt
University of California, Berkeley, USA
schuldt@berkeley.edu

Rastislav Bodik
University of Washington, USA
bodik@cs.washington.edu

Abstract

Energy efficiency is one of the main performance goals when designing processors for embedded systems. Typically, the simpler the processor, the less energy it consumes. Thus, an ultra-low power multicore processor will, likely have very small distributed memory with a simple interconnect. To compile for such an architecture, a partitioning strategy that can tune between space and communication minimization is crucial to fit a program in its limited resources and achieve good performance. A careful program layout design is also critical. Aside fulfilling the space constraint, a compiler needs to be able to optimize for program latency to satisfy a certain timing requirement as well.

To satisfy all aforementioned constraints, we present a flexible code partitioning strategy and light-weight mechanisms to express parallelism and program layout. First, we compare two strategies for partitioning program structures and introduce a language construct to let programmers choose which strategies to use and when. The compiler then partitions program structures with a mix of both strategies. Second, we add supports for programmer-specified parallelism and program layout through imposing additional spatial constraints to the compiler. We evaluate our compiler by implementing an accelerometer-based gesture recognition application on GA144, a recent low-power minimalistic multicore architecture. When compared to MSP430, GA144 is overall 19x more energy-efficient and 23x faster when running this application. Without these inventions, this application would not be able to fit on GA144.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.4 [*Programming Languages*]: Processors–Compilers; I.2.2 [*Artificial Intelligent*]: Automatic Programming–Program transformation

Keywords Program Partitioning, Program Layout, Parallelism, Constraint Solving, Embedded Systems, Distributed Memory

1. Introduction

Energy requirements have been dictating simpler processor designs with more energy dedicated to computation and less to processor control. We imagine that future low-power embedded processors will be minimalistic. Multicore processors will likely have simple interconnects. A GreenArrays GA144 is a recent example of a low-power minimalistic spatial multicore architecture[1] [7]. It is likely the most energy-efficient commercially available processor [11]. Naturally, energy efficiency comes at the cost of low programmability; among many challenges of programming GA144, programs must be meticulously partitioned and laid out onto the physical cores such that code and data for each core can fit in its 144 bytes of memory.

Chlorophyll [15] is a language and a constraint-based compiler developed for GA144. To our knowledge, Chlorophyll is the only compiler that compiles from a high-level language to arrayForth, GA144 assembly. Chlorophyll programming model allows programmers to guide the compiler how to partition data and computations. The compiler relies on constraint solving and program synthesis to sidestep the laborious development of classical transformations and optimizations. We use Chlorophyll to compile a realistic embedded application for GA144. We select an accelerometer-based gesture recognition application as our case study because it requires both intensive computations and a relatively large data storage for gesture models. As a result, it is not typically run on an embedded device but offloaded to a more powerful computer or cloud. However, we believe that performing the classification locally is more energy-efficient. In order to make the application fit on GA144 and satisfy an additional timing requirement, we introduce the following extensions to the language and compiler.

In a very limited-resource environment such as a very small many-core, distributed-memory processor, a program partitioning strategy is very critical because it not only affects the performance of the application but, more importantly, determines whether the application can fit and run on the processor. In this paper, we are interested in strategies to partition program control flow constructs (i.e., program structures). The decision between communication and recomputation of non-control flow computations is another interesting question related to program partitioning, but this question is not in the scope of this paper. To improve Chlorophyll's strategy for partitioning control statements, we compare the original strategy to an alternative and identify their advantages in different scenarios. The *SPMD* (Single Program, Multiple Data) strategy, the original strategy, replicates control flow constructs in all relevant cores. The *actor* strategy avoids replicating control flow constructs by isolating program portions from the rest of the program control flow and dedicating a group of cores for each portion to execute the code upon receiving a request. The two strategies produce more efficient code—less communication and/or smaller code—in different scenarios. We identify when to use one or another and provide programmers with a construct for choosing between the two.

LCTES'16, June 13–14, 2016, Santa Barbara, CA, USA
© 2016 ACM. 978-1-4503-4316-9/16/06...$15.00
http://dx.doi.org/10.1145/2907950.2907962

[1] A *spatial architecture* is an architecture for which the compiler must assign data computations explicitly to its specific hardware resources.

In addition to having a good partitioning strategy, to satisfy both space and timing requirements, we also need a good program layout and parallelism. A constraint-based compiler should be able to generate a good program layout and parallelize a program, if the cost functions used by its constraint-solving procedures can precisely capture the architecture's performance model and all other relevant constraints. However, creating precise cost functions is difficult, and using precise cost functions is also often not scalable. On the other hand, abstract cost functions may lead to poor or infeasible solutions. In this paper, we show that we can work around the issue of abstract cost functions by using help from programmers to provide alternative constraints that are easier to deal with but provide the same effects. First, by simply providing a constraint that computations are placed onto distinct logical cores, programmers can control the computations to be executed in parallel. This annotation overcomes the limitation that the abstract cost model used by the partition type inference ignores program latency. Second, a constraint that pins a logical core to a physical core allows programmers to express program layouts, placing frequently communicating computations on nearby cores. This constraint compensates for the limitation that the abstract cost model used by the layout synthesizer ignores the code size constraint. We create new language constructs for programmers to impose these constraints.

Lastly, we describe our programming toolchain, demonstrate the benefits of having simulators at several levels of program intermediate representations, and report on the experience of developing the gesture recognition application. Our implementation of the application occupies a total of 82 cores of GA144. The application is 81%–91% accurate at classifying circle and flip-roll gestures. Compared to an implementation on MSP430, our GA144 implementation is overall 19x more energy-efficient and 23x faster. On the computationally expensive part of the application, GA144 is even more superior, 71x more energy-efficient. Without the new techniques we introduce, this application would not be able to run on GA144.

In sum, this paper makes the following contributions:

- Comparing SPMD and actor partitioning strategies for a very limited-resource environment and introducing a compilation mechanism to support a mix of both strategies (Section 3)
- Introducing a light-weight constraint-based mechanism to support parallelism and programmer-specified program layout (Section 4)
- Providing a toolchain for compiling and debugging GA144 programs (Section 5)
- Providing a low-power implementation of a gesture recognition application on GA144 (Sections 6 and 7)

2. Background

2.1 GreenArrays GA144

GA144 is a scalable embedded multicore architecture consisting of a 2D array of 18×8 cores [7]. Each core is identical to the others except on available I/O pins. Each core is an 18-bit Forth stack machine containing only a tiny amount of memory (64-word RAM and ROM) and two small circular stacks (one for data and one for return addresses). This forces programs and data structures to be partitioned over multiple cores. Each core runs asynchronously at 666 MIPS but can communicate with their four immediate neighbors using blocking reads and writes. GA144 consumes less energy per instruction than any other commercially available architectures [11]. It consumes 9x less energy and runs 11x faster than the TI MSP430 low-power microcontroller on a Finite Impulse Response benchmark [1].

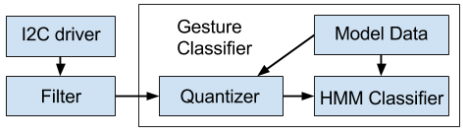


Figure 1. Accelerometer-based gesture recognition algorithm

2.2 Gesture Recognition Application

We select an accelerometer-based gesture recognition application as our case study, because this application becomes more and more popular as it provides an intuitive interaction from human to computer. Typically, the classification is not done locally on an embedded device that collects the data, because it is computationally expensive. However, we believe that performing the classification locally is more energy-efficient. We use the gesture recognition algorithm from Wiigee [16]. The main components of the algorithm, displayed in Figure 1, are a filter and a gesture classifier, which is composed of a quantizer and a Hidden Markov Model (HMM) classifier. The filter removes acceleration vectors[2] that do not significantly deviate from the gravitational acceleration or the previously accepted vector from the incoming stream of acceleration vectors. Vectors that are passed continue to the quantizer, which maps each input vector to a group number. The group number is found by searching for the closest vector to the input vector from the 14 centroid vectors. The set of centroid vectors are created during training using k-mean clustering. Given a group number, the HMM classifier of each gesture updates its belief state. The HMM model is created during training using the Baum-Welch algorithm. After a gesture completes, we obtain the probabilities from the different gesture classifiers, the gesture with the highest probability is the most likely gesture.

2.3 Original Chlorophyll Language and Compiler

Chlorophyll compiles a high-level program to GA144 assembly by decomposing the compilation problem into four main subproblems: partition type inference, layout and routing, code separation, and code generation.

2.3.1 Programming Model and Partitioning Strategy

The Chlorophyll language is a subset of C with *partition annotation*, which is used for assigning a *partition type* to data and operation. A partition represents a logical core. Every piece of data and operation lives in a partition, specified by its partition type. Programmers can specify partition annotations when they wish to partially or fully partition programs. For example, in this program:

```
int@0 mult(int x, int y) { return x * y; }
```

the programmer specifies that the return value of the function will be delivered at partition 0 but does not specify the partitions of variable `x`, `y`, and operation `*`. The programming model governs the compiler's partitioning strategy to put each constant, variable, array, operator in only one place. However, there are still multiple ways to partition control statements.

2.3.2 Partition Type Inference

The partition type synthesizer infers unannotated partition types, such that each partition fits into a core (occupying no more than 64 words) and minimizes a static over-approximation of the amount of messages sent between partitions. However, it ignores other performance counters including program latency.

[2] An acceleration vector consists of (x,y,z) accelerometer values.

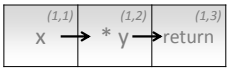
Given the example program, the compiler may infer (for a very tiny core):

```
int@0 mult(int@2 x, int@1 y) { return (x *@1 y); }
```

The inferred annotations indicate that when function `mult` is called, `x` is passed as an argument at partition 2, and `y` is passed as another argument at partition 1. The program body's annotations specify that the value of x at partition 2 is sent to partition 1, and is multiplied with the value of `y`. Finally, the result of the addition is sent to partition 0 as the function's return value.

2.3.3 Layout and Routing

The layout synthesizer then maps each program partition onto a physical core, minimizing communication cost, the sum of an over-approximated number of messages between every pair of cores times the distance between them. To make this problem tractable, the layout synthesizer does not take the code size constraint into account. It employs the Simulated Annealing algorithm to solve the layout problem and determines a communication route between each pair of cores by selecting an arbitrary shortest path. Given the fully-annotated program from the previous step, the figure below shows one possible solution from the layout synthesizer.



2.3.4 Code Separation

The separator splits a fully partitioned program into per-core program fragments and inserts send and receive instructions for communication. It splits the running example into:

```
// Core (1,1) core ID is (x,y) position on the chip
void mult(int x) { send(EAST, x); }
// Core (1,2)
void mult(int y) { send(EAST, read(WEST) * y); }
// Core (1,3)
int  mult()      { return read(WEST); }
```

Observe that each variable lives in only one core, but the function `mult` lives in all cores that have data or computations in the function.

2.3.5 Code Generation

The code generator first naively compiles each program fragment into machine code without any local machine-specific optimization. The code is then optimized with a superoptimizer [12], which searches the space of possible instruction sequences to find ones that are correct and fast or short.

3. Program Structure Partitioning Strategy

This section describes our flexible program structure partitioning in which the programmer controls the replication of control statements, trading communication efficiency for code size reduction. Note that this paper focuses on how control statements can be partitioned across multiple cores, not on how data and other computations should be partitioned; we apply the same strategy to partition data and computations from Chlorophyll. We first describe the two extreme strategies: the *actor model*, which replicates no code, and the *SPMD model*, which replicates all control statements. We then develop a compilation algorithm that replicates code based on function-level annotations for partitioning a program.

3.1 SPMD vs. Actor Partitioning Strategy

An invariant of our partitioning is that data and non-control computations are not replicated; they are always assigned to exactly one logical partition, which is mapped to one physical core. The programmer controls only replication of control statements (`if`s and loops), which determine the control condition, i.e., when a statement is executed. Empirically, replication of control statements across cores can eliminate messages whose sole purpose is to send the control condition; it may be less expensive to include on that core the control statement that computes the control condition.

The Actor Strategy. In the actor strategy, when a partition p_1 needs a value to be computed on another partition p_2, it sends a request message to p_2 and waits until the value is returned. On partition p_2, the computation is executed by an actor that is active only when responding to a request. While the actor strategy minimizes code duplication, it may incur more communication. Consider the program in which the function `f` executes on partition 2.

```
int@2 f(int@2 i) { ... }
int@1 x;
for (i from 0 to 100) x += f(i);
```

When `f` is an actor, each loop iteration requires three messages (the request to execute `f`, the argument, and the return value):

```
// Partition 1
for (i from 0 to 100)
  x += actor_call(f, i); // call f in the other partition

// Partition 2
port_execution_mode(); // execute f when requested
int f(int i) { ... }
```

Many messages from 1 to 2 can be eliminated by replicating the loop on partition 2, which is done by the the SPMD model.

SPMD Strategy. The SPMD model, introduced by Callahan and Kennedy [4], replicates all control flow constructs onto every partition, in the spirit of the Single-Program Multiple-Data model [10]. Each partition becomes independent in that it decides when to execute each statement, but naturally the replicated control flow constructs need to obtain their predicate values, and these may need to be communicated from the partition that computes the predicate. For loops with compile-time bounds, the condition is replicated and computed locally. For the example program above, the SPMD strategy will split the program into:

```
// Partition 1
for (i from 0 to 100) x += recv();

// Partition 2
int f(int i) { ... }
for (i from 0 to 100) send(f(i));
```

While this strategy duplicates control flow constructs, it can reduce communication significantly. In this program, only one message is sent per iteration. However, if the control flow of a program is complex, and the predicates of the control flow statements are not known at compile time, this strategy may require a lot of messages for sending the predicate values.

3.2 Language Extension for Controlling Replication

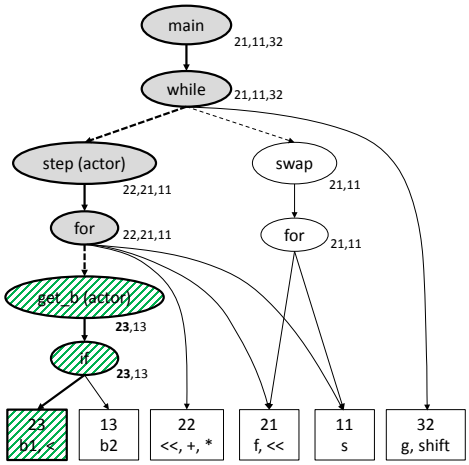
Rather than controlling replication of control statements in each individual partition, we control replication at the granularity of a function (a set of partitions or cores). The programmer specifies which functions should be compiled into actors, and the remaining function calls are invoked under replicated control flow. Programmers *actorize* a function by defining:

```
// Do not specify the requester and master actor
actor FUNC;
// Specify the requester and master actor
actor FUNC(REQUESTER => MASTER);
```

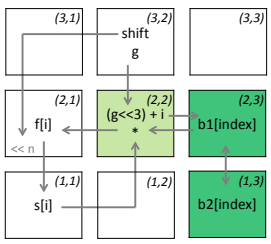
One of the actor partitions in the function is a dedicated *master actor* partition, and the rest are *subordinate actor* partitions. A *requester* partition is responsible for sending a remote execution

```
1  fix1_t@21 f[8];
2  fix1_t@11 s[8];
3  fix1_t@23 b1[32];
4  fix1_t@13 b2[32];
5
6  actor get_b(22=>23);
7  fix1_t@23 get_b(int@23 index) {
8    if (index <@23 32)
9      return b1[index];
10   else
11     return b2[index -@13 32];
12 }
13
14 actor step(32=>22);
15 void step(int@22 g) {
16   for (i from 0 to 8)
17     f[i] = s[i] *@22 get_b((g <<@22 3) +@22 i);
18 }
19
20 void swap(int@21 n) {
21   for (i from 0 to 8) s[i] = f[i] <<@21 n;
22 }
23
24 void main() {
25   while(1) {
26     int@32 g = ...; int@32 shift = ...;
27     step(g);
28     swap(shift);
29   }}
```

(a) Chlorophyll source. `blue`, `pink`, and `purple` highlight data and computations assigned to partition 22, 23, and 13 respectively. `fix1_t` is a fixed point data type with one bit for the integer part.



(b) CDG. An oval represents a control flow construct node. A rectangle represents a partition node, grouping operations and data that belong to the partition. The **entire highlighted path** is the *control flow slice* of partition 23. The **green striped-highlighted path** is its *relevant control flow*. Numbers attached to each node are the *relevant partitions* of the node.



(c) Layout and routing. Partition xy is mapped to physical core (x, y). **Dark** and **light** green denote actor cores of `get_b` and `step` respectively.

Figure 2. A simplified example program taken from the gesture recognition application

```
1  port_execution_mode(N);
2
3  void step(int g) {
4    for (i from 0 to 8) {
5      int s = recv(S);
6      // call actor get_b in p.23 (east)
7      int b = actor_call(E, get_b, (g << 3) + i);
8      send(W, s * b);
9    }
10 }
```

(a) A requester of the function `get_b`: partition 22 or core (2,2)

```
1  port_execution_mode(W); // wait for request from p.22
2
3  fix1_t b1[32];
4  fix1_t get_b(int index) {
5    int cond = index < 32;
6    send(S, cond);     // send condition to p.13
7    if (cond) {
8      return b1[index];
9    } else {
10     send(S, index);  // send index to p.13
11     return recv(S); // return value from p.13
12   }}
```

(b) A master actor of the function `get_b`: partition 23 or core (2,3)

```
1  fix1_t b2[32];
2
3  void get_b() {
4    // wait for condition from p.23 to start
5    if (recv(N)) { }
6    else {
7      // get index from p.23 and return value to p.23
8      send(N, b2[recv(N) - 32]);
9    }
10   get_b(); // loop back to the beginning
11 }
12
13 void main() { get_b(); }
```

(c) A subordinate actor of the function `get_b`: partition 13 or core (1,3)

Figure 3. Program fragments of partitions 22, 23, and 13 generated by the compiler when compiling the program in Figure 2. The compiler places `blue`, `pink`, and `purple` highlighted data and computations from Figure 2(a) in partitions 22, 23, and 13 respectively. `N`, `S`, `E`, and `W` stand for north, south, east, and west ports.

request to the master actor to invoke the function. The master actor in turn triggers subordinate actors to invoke their functions through data dependencies. More specifically, a subordinate actor waits for data to be used in its computations inside the function; the data essentially triggers the rest of the computations inside the function. Relying on these triggers, program fragments of actor partitions of a function `func` do not need to contain the control statements between the function `main` to the calls to `func`. If a partition does not belong to any actor function, it contains the control statements starting from `main`.

Consider the fully-annotated simplified snippet of code from the gesture recognition application in Figure 2(a). We actorize `get_b` by annotating `actor get_b(22=>23)`, specifying that partition 22 is a requester and 23 is a master actor. Since `get_b` also contains partition 13 in addition to partition 23, partition 13 becomes a subordinate actor. Figure 3 displays illustrative per-core program fragments generated by our compiler before being converted into machine code. Figures 3(b) and 3(c) display the program fragments of partition 23 and 13 respectively. Notice that both partitions do not have the control statements between `main` and the call to `get_b` (i.e., `while`, `step`, and `for`). Partition 23, the master actor, is in

the port execution mode (line 1), waiting for 22 to send a remote execution request to invoke `get_b`. When 23 finishes executing `get_b`, it goes back to the port execution mode waiting for the next request. Partition 23 triggers 13, the subordinate actor, to start computations in `get_b` by sending the predicate of `if` (data dependency). Partition 13 waits for this data (line 5) to start its task. When it finishes the task, it loops back to the beginning of the program to handle the next request. Figure 3(a) displays the partitioned program fragment at 22. Since it is the requester for `get_b`, it sends a remote execution request (line 7) to 23, the master actor. Partition 22 itself is an actor for another function `step`, so it is also in the port execution mode.

3.3 Compiling for Hybrid Strategy

3.3.1 Design Decisions

To support both partitioning strategies in the compiler, we make the following design decisions. First, we let programmers actorize at a function-granularity level. Among the constructs provided by Chlorophyll, a partition and a function are referenceable entities that can be considered as an actor, an entity that acts upon receiving a request. However, we believe that programmers prefer to reason about functionality of programs rather than reason about implementation details. Furthermore, if programmers let the compiler infer partition types for most parts of their programs, they will not know which partitions are responsible for which parts of the programs, so they cannot actorize their programs appropriately. Second, we let programmers make a decision if each function should be an actor or not. While it is possible to automate this decision, the automation is not the focus of this paper. Third, we associate an actor function with a requester and a master actor. To invoke an actor function, the requester sends a remote execution request to the master actor, which in turn triggers the subordinate actors through data dependencies. We could have made the master actor trigger the subordinate actors using explicit remote execution requests, the same way the requester triggers the master actor; however, using remote execution requests may incur a lot of communication, so we decide to utilize data dependencies instead. Nevertheless, we do not rely on the data dependency to invoke the master actor because we would like to support actorizing functions with no argument that perform I/O activities.

Restrictions Imposed by Our Decisions. Since a subordinate actor partition is invoked upon receiving any data from an expected neighbor port, it does not have a mechanism to distinguish between different requests for different tasks because the request does not identify the task to be invoked. Therefore, to simplify our implementation, we restrict a partition to be an actor for no more than one actor function. If a partition is in more than one actor functions, that partition will not be an actor for any function, so the partition will have the entire control flow from `main`. Furthermore, if a partition is an actor for a function, that partition cannot be used anywhere else outside the function, including for routing messages for any computation outside the function. Thus, too much actorization may cause the compilation to fail due to its implication on the routing restriction.

Advantages Over Low-Level Partitioning Control. The GreenArrays vendor provides a programming environment for GA144, called arrayForth. It allows programmers to explicitly write separate programs for individual cores with the flexibility to manually duplicate data, operations, and control statements, however, in a stack-based assembly language. Recall that Chlorophyll compiles to arrayForth. We can think of arrayForth as an MPI-style programming model. Initially, partitioning control flow statements in arrayForth may seem more intuitive than controlling the partitioning strategy in Chlorophyll because programmers maybe more familiar with MPI than SPMD and actor concepts. However, there are several advantages to program GA144 using Chlorophyll with the programmer-controlled hybrid partitioning strategy. First, both SPMD and actor partitioning strategies guarantee that the generated code is deadlock-free, unlike MPI. Although actorization may cause the compilation to fail because of too many constraints on communication routing, the failure happens at compile time, so programmers can fix their programs accordingly. Second, it is easier for a programmer to write a whole program in a sequential style than to write separate program fragments that run in parallel. Furthermore, using annotations to control the partitioning strategy enables programmers to easily explore different ways to partition program structures just by inserting or deleting actor annotations.

3.3.2 Where to Place Control Flow Constructs

We decide which partitions need copies of each control flow construct based on the *relevant control flow slice* of each program partition. In this section, we define the control flow slice and the relevant control flow slice. Then, we show how the compiler uses this information to separate a program into per-core program fragments.

Control Flow Slice. We define a *control flow slice* of a partition, based on the program slicing terminology [19], to consist of the control flow constructs in the slice of the source program with respect to the partition's data and computations as a *slicing criterion*. A slice of a program with respect to a slicing criterion can be determined from a Program Dependence Graph (PDG). We are interested in computing a control flow slice of a program with respect to a program partition, so we can use a Control Dependence Graph (CDG), a subgraph of PDG. Typically, a node in a CDG is either a control flow construct or a statement in the program. However, since our slicing criterion is in the level of a logical partition instead of a statement, we group statements, operations, and data that belong to the same partition together in one node, called a partition node. Thus, our CDG contains partition nodes instead of statement nodes. An edge represents a control dependency between nodes in a CDG. Figure 2(b) depicts the CDG of the program in Figure 2(a). Ovals represent control flow constructs. Rectangles represent logical partitions. Dashed edges indicate interprocedural control dependency (function calls). A control flow slice of a partition can be directly derived from a CDG. A control flow slice of a partition consists of all paths from the `main` node to the partition node. Figure 2(b) highlights the control flow slice of partition 23.

Actor and Its Relevant Control Flow Slice. With the pure SPMD strategy, a program fragment of a partition generated by the compiler will contain the data and computations assigned to that particular partition and the entire control flow slice of the partition. In contrast, as we described in Section 3.2, a partition of an actor function does not need to own the control flow constructs between `main` and the calls to that particular function because a requester partition is responsible for remotely invoking that function. We call the control flow constructs that an actor partition actually needs (excluding the constructs between `main` and the actor function) the *relevant control flow slice* of the partition.

Therefore, our task is to determine what the relevant control flow slice of each partition in the program is. First, we need to identify which partition is an actor for which function. Identifying an actor partition is not as straightforward as it seems because not all partitions inside an actor function are actors. To identify actor partitions, we perform a reachability analysis on a CDG. Partition p is an actor and belongs to actor function f, if f is the most *immediate* node in the CDG that *covers* p. Function f *covers* p if and only if there is no path from `main` to p when f is removed from the CDG. p can be covered by multiple actor functions. In such a case, p belongs to the most *immediate* actor function; there is no

Algorithm 1 Communication routing

```
Global variables: ActorsMap, AllActors, UsedCores
 1: function ROUTE(a, b)
 2:     Allow ← {}
 3:     actorFuncA ← getActorFunc(a)
 4:     actorFuncB ← getActorFunc(b)
 5:     if actorFuncA then
 6:         Allow ← Allow ∪ ActorsMap[actorFuncA]
 7:     if actorFuncB then
 8:         Allow ← Allow ∪ ActorsMap[actorFuncB]
 9:     Obstacles ← AllActors − Allow
10:     Path ← A*search(a, b, Obstacles)
11:     if actorFuncA and actorFuncA = actorFuncB then
12:         ActorsMap[actorFuncA] ←
13:             ActorsMap[actorFuncA] ∪ (Path − UsedCores)
14:     UsedCores ← UsedCores ∪ Path
15:     return Path
```

other actor function between the most immediate actor function and p. For example, in Figure 2(b), partition 23 is covered by both actor functions `get_b` and `step`, but `get_b` is the most immediate function to 23, so 23 belongs to `get_b`. Now consider 21, which is a partition inside the actor function `step`, but it is being used in the function `swap` outside the scope of `step`. According to the CDG, `step` does not cover 21. Thus, 21 is not an actor partition for step.

Once we identify actor partitions for all actor functions, we can compute the *relevant control flow slice* for every partition. Figure 2(b) highlights nodes in the relevant control flow slice of partition 23 with green stripes, which is its original control flow slice excluding the control flow constructs from `main` to `get_b`. We can also see that each program fragment of a partition in Figure 3 only contains the relevant control flow slice of that particular partition. Partition 22 only contains `step` and `for`. Partitions 23 and 13 only contain `get_b` and `if`.

Inversely, at each control flow construct, we can also compute a set of *relevant partitions*, partitions whose relevant control flow slices contain that particular control flow construct. Figure 2(b) labels each node in the CDG with its relevant partitions.

Routing Constraint and Algorithm. After the compiler maps logical partitions to physical cores, actor partitions become actor cores. This section describes how the routing algorithm works in the presence of actor cores.

Recall our restriction that if an actor partition is associated with an actor function f, the actor partition cannot contain code or data used anywhere outside f. During the code separation step, the compiler inserts communicate code between communicating cores; therefore, we have to make sure that for every actor core u in an actor function f, the compiler does not insert communication code associated with data sending outside f in u.

To ensure this property, we have to modify the routing algorithm. The original routing algorithm simply returns any shortest path between the two cores. Since there is no obstacle, finding a shortest path is as simple as moving along the x-axis to the target y-coordinate and then moving along the y-axis to the target x-coordinate. With the new restriction, the routing needs to be able to avoid obstacles. Algorithm 1 displays our new routing algorithm. Generally, when routing from core a to b, we need to avoid routing through any actor core. However, if a and b are actors of functions f_a and f_b respectively, then the communication path between a and b can go through other actor cores of both functions (lines 3–8). We use A* search algorithm to find a shortest path between a and b that avoids obstacles (actor cores from the other actor functions) (line 10). After we obtain the path, if a and b are both actors of the same function, we promote all cores along the path as actors of that function if they have never been used before (line 11-13).

Figure 2(c) displays the layout and routing of the running example in Figure 2(a). Assume that the layout synthesizer maps partition xy to physical core (x, y), which represents a coordinate on a 2D grid. When the compiler finds a path for sending the value of the variable `shift` from node (3,2) to (2,1), it avoids routing through actor cores (2,2), (2,3), and (1,3).

Code Separation. After routing, the compiler recomputes the CDG of the program. At this step, a partition node in the old CDG becomes a physical core node in the new CDG. We insert additional cores that are only responsible for routing data into the new CDG, for example, cores (3,1) and (1,2) in Figure 2(c), which are only used for routing data. We also add an edge connecting `main` node to core (3,1) node in the new CDG because the value of the variable `shift` is sent from core (3,2) to (2,1) through (3,1) in `main`. Similarly, we add an edge connecting `step` to core (1,2) node because the value of `s[i]` is sent from core (1,1) to (2,2) through (1,2) in the function `step`.

With the new CDG, we recompute relevant cores for each control flow construct. Afterwards, we separate the program AST into per-core program fragments. While traversing the program AST, we put each piece of data and computation into the assigned core. When we encounter a control flow construct, we place it in all of its relevant cores. When we encounter an actor function call, we insert a command to send a remote execution request in the requester core to be sent to the master actor core. We set all master actor cores to port execution mode, waiting for requests from the appropriate ports. Figure 3 displays some program fragments of the running example after the code separation step.

4. Extensions to Spatial Constraints

We add light-weight extensions to the compiler's spatial constraints to allow programmers to express parallelism and program layout.

4.1 Parallelism

Partition type, originally introduced as a mechanism to partition data and computations, can be used for expressing parallelism. If two operations have different partition types, they will be executed at two different cores and maybe executed at the same time in parallel, depending on data and control dependency. Consider the following program:

```
int@1 x;
int@2 y;
x = x +@1 1;
y = y -@2 1;
```

The increment of `x` and and decrement of `y` run in parallel in partition 1 and 2 respectively. The language also allows programmers to declare distributed arrays, which can be used to express data parallelism. For example,

```
// The first 16 elements are in partition 0.
// The last 16 elements are in partition 1.
int@{[0:16]=0, [16:32]=1} x[32];
for (i from 0 to 32)
  x[i] = x[i] +@place(x[i]) 1;
```

is separated to

```
// Partition 0
  int x[16];
  for (i from 0 to 16)
    x[i] = x[i] + 1;

// Partition 1
  int x[16];
  for (i from 16 to 32)
    x[i-16] = x[i-16] + 1;
```

Consequently, the two parts of the array are incremented in parallel.

4.1.1 Challenges of Automatic Parallelization

Automatically parallelizing programs is generally challenging. A very limited-resource constraint in a distributed-memory environment poses even more challenges to this problem. For example, in the gesture recognition application, we would like to update the belief states of two HMM classifiers in parallel. In this code:

```
hmm_step(acc,model1);
hmm_step(acc,model2);
```

the two `hmm_step`s do not run in parallel because only one set of partitions (hence, one set of cores) is responsible for executing the function. In order to make them run in parallel, the compiler needs to make two copies of `hmm_step`—e.g., `hmm_step1` and `hmm_step2`—and ensure that `hmm_step1` and `hmm_step2` do not use the same partitions. In fact, programmers can make these two functions run in parallel by themselves, but they will need to manually duplicate the function `hmm_step`. Additionally they need to explicitly assign every data and operation in the function to a partition to make sure that the two copies of `hmm_step` do not share any common partition. Doing this manually is highly error-prone and unproductive. Even then, the two functions may not run in parallel because communication routing may introduce dependency between the two functions.

Currently, the compiler does not support automatic parallelization for two reasons. First, the partition type inference may not infer partition type such that `hmm_step1` and `hmm_step2` will run in parallel. This is because the partition type inference only minimizes the amount of communication between cores and ignores the latency of the program. Second, even if the partition type inference is aware of program latency, it has to choose between minimizing for latency (consequently, obtaining parallelism) or resource usage (consequently, minimizing power consumption). These are conflicting goals because parallelism on GA144 requires more cores and memory. We believe that this kind of decision should be made by programmers and not by the compiler.

4.1.2 New Extension

Thus, we introduce an explicit parallel construct called *parallel module*, which can be used by programmers to express parallelism. Module and module instance are syntactically similar to class and object. For example, the program in Figure 4(a) uses the module construct to update the belief states of the HMM classifiers in parallel. Despite its syntactic similarity to class, module behaves like a macro. We insert a module expansion pass into the compiler, which expands the program in Figure 4(a) to the program in Figure 4(b), similar to a macro expansion. After the expansion, we obtain the program in the original Chlorophyll language.

To ensure parallelism, we have to modify the partition type inference and the routing algorithm. We add an extra constraint to the partition type inference to ensure that two module instances of the same module do not share any common partition types; this guarantees that the two module instances occupy two disjoint sets of partitions. We also modify the routing algorithm such that when cores a and b are in the same module instance, a communication path between a and b cannot pass any core in the other module instances of the same module. Once the algorithm finds a path between a and b, it adds cores along the path as members of a's and b's module instance as well.

We also support parallel map and reduce on distributed arrays. They are not used in the gesture recognition application, but they can be very useful for implementing data parallel programs. The compiler handles parallel map and reduce the same way it handles parallel module: desugaring into the original language and adding constraints to the partition type inference and the routing algorithm.

```
// Define module.
module Hmm(model_init) {
  fix1_t model[N] = model_init;
  fix1_t step(fit1_t[] acc) { ... }
}

// Create module instances.
hmm1 = new Hmm(model1);
hmm2 = new Hmm(model2);

// Call two different functions.
hmm1.step(acc);
hmm2.step(acc);
```

(a) Source code in Chlorophyll

```
// Expanded from module instance 1.
fix1_t hmm1_model[N] = model1;
fix1_t hmm1_step(fit1_t[] acc) { ... }

// Expanded from module instance 2.
fix1_t hmm2_model[N] = model2;
fix1_t hmm2_step(fit1_t[] acc) { ... }

hmm1_step(acc);
hmm2_step(acc);
```

(b) De-sugared code after module expansion

Figure 4. Example of a parallel HMM classification program using the module construct

4.2 Program Layout

The compiler assigns logical partitions to physical cores by using Simulated Annealing (SA) [5]. Specifically, given a set P of partitions, a set C of cores, a flow function $t : P \times P \to \mathbb{R}$, and a distance function $d : C \times C \to \mathbb{R}$, we want to find the assignment $a : P \to C$ that minimizes the following communication cost:

$$\sum_{p_1 \in P, p_2 \in P} t(p_1, p_2) \cdot d(a(p_1), a(p_2))$$

The flow is the number of messages between any two partitions, and the distance matrix stores the Manhattan distance between each pair of cores. The solution is a layout that minimizes the communication cost.

4.2.1 Challenges of Layout Synthesis

The cost function of the layout problem does not take other spatial constraints, such as code size, into account. As a result, the layout synthesizer may produce a program layout, in which some cores are heavily responsible for routing communication, and the final code does not fit in every core. The gesture recognition program requires a careful program layout design in order to fit on GA144. Unfortunately, because of the abstract cost function used by the layout synthesizer, the compiler cannot generate code that fits on GA144. One way to solve this issue is to make the cost function aware of the code size constraint. However, doing so is difficult.

4.2.2 New Extension

Instead, we compensate this limitation of the compiler by allowing programmers to design their own program layouts.

Pinning Individual Partition. We introduce a language construct to pin a logical partition to a physical core:

```
# PARTITION --> CORE
```

We modify the SA algorithm to understand this extra constraint. Initially, SA assigns partitions to cores randomly, and in every

round it randomly selects random pairs of cores and swaps the two partitions inside each pair. We modify SA such that when a programmer pins a partition p to a core c, it will initially put p in c and never swap the partition in c with a partition in any other core.

Pinning a Set of Partitions. We also support pinning a set of partitions to a set of cores by pinning a module instance. In the program in Figure 4(a), we can pin partitions in `hmm1` to be in cores (1,1), (1,2), (2,1), and (2,2) by:

```
hmm1 = new Hmm(model1)@{(1,1),(1,2),(2,1),(2,2)};
// or
hmm1 = new Hmm(model1)@REG((1,1),(2,2));
```

where `REG(BL,TR)` is an abbreviation for a set of cores covered by a rectangle whose bottom left is at `BL` and top right is at `TR`.

Inside a pinned module, we can pin individual partitions as well. For example, in this program:

```
module Hmm(model_init) {
  # 0 --> (0,1)
  fix1_t@0 model[N] = model_init;
  fix1_t@0 process(x,y,z) { ... }
}

hmm1 = new Hmm(model1)@REG((1,1),(2,2));
```

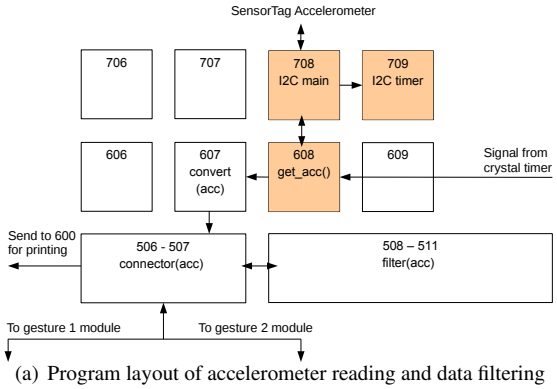
we specify that partitions inside `hmm1` should be placed at core (1,1), (1,2), (2,1), and (2,2), and specifically we want partition 0 in `hmm1` to be at (0,1) relative to the most bottom-left core of `hmm1`, so partition 0 is placed at core $(1+0, 1+1) = (1, 2)$.

5. Toolchain and Debugger

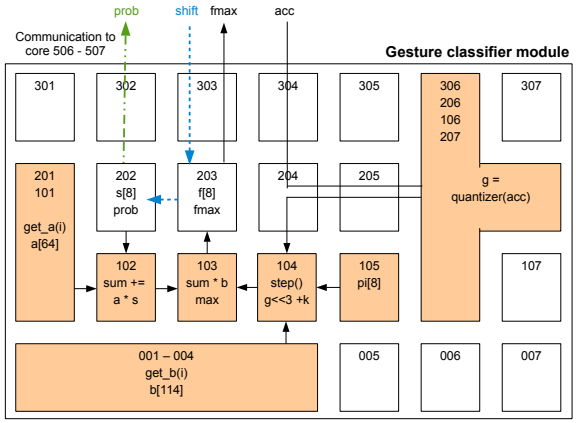
We have seen in practice that having multiple stages of testing is important and productive when programming for a complex hardware. For instance, a FPGA toolchain provides multiple simulations: behavioral simulation, functional simulation, timing simulation, and circuit verification [20]. Therefore, we develop a functional simulator, multicore simulator, and machine simulator for testing at different stages of the compilation.

First, the functional simulator allows programmers to test their algorithms without worrying about any implementation detail such as partitioning and layout. Since the core Chlorophyll language is a subset of C, we can easily generate a C program to be used for the functional simulation. Second, the multicore simulator allows us to doubly verify that the compiler indeed generates deadlock-free code. After the code separation step, we obtain per-core program fragments. At this stage, we utilize C++ pthread and mutex lock to produce the multicore simulator. Specifically, we create a thread to simulate a core running each program fragment. Each communication channel between two cores is represented by a variable with a lock to simulate blocking reads and writes. Third, the machine simulator interprets programs at the bit level, as if running on real hardware. It gives programmers many standard debugging methods to help eliminate bugs before running on real hardware, including breakpoints, state examination, and code execution. This has been especially useful when debugging problems with compiler's generated machine code. Other useful features include a support for multiple GA144 chips and a support for building testbeds to simulate I/O devices. When debugging multiple chips, their pins may be virtually wired together, and programs on different chips can be debugged in a single system.

This choice of having multiple simulators allows for simulating program execution at the abstraction level of the problem a programmer is trying to resolve.



(a) Program layout of accelerometer reading and data filtering



(b) Program layout of one gesture classifier. Black solid and blue dashed arrows denote data flow for updating the belief states `f` and `s` respectively in every classification round. A dotted-dashed green arrow denotes data flow for obtaining a final probability after 1000 rounds.

Figure 5. Program layout for the gesture recognition application. Each core is labeled with three digits. The first digit indicates the x-coordinate. The last two digits indicate the y-coordinate. Orange highlights cores that are actors.

6. Application Implementation

Figure 5 displays the program layout of our gesture recognition application on GA144. The accelerometer reading, filtering, and the connector of all components are located in the top part of the chip, as shown in Figure 5(a). This leaves the entire bottom portion of the chip for the gesture classifiers, but there is only room for two gesture classifiers, given their size. The two gesture classifiers have an identical layout shown in Figure 5(b).

6.1 Functional Implementation Details

Accelerometer Reading. The I2C implementation used for communicating with the accelerometer is based on GreenArrays' SensorTag application documentation [8]. A 32KHz software controlled crystal is used to clock the application to read the accelerometer at 200 Hz. The main I2C node 708 communicates with the accelerometer using node 709 to wait for clock edges. Node 608 passes the raw accelerometer register values to node 607, which converts the raw values into a proper fixed-point format and sends the converted values to the connector nodes 506 and 507.

Gesture Classifier. The connector nodes pass the accelerometer data to the filter function and the gesture classifiers. They also

gather the final probabilities from the gesture classifiers and send them to node 600 to be printed via a serial port.

There are three types of communication between the connector and a gesture classifier as illustrated with solid, dashed, and dotted-dashed arrows in Figure 5(b). The solid arrows depict the data flow for the main computation for each round of accelerometer reading. The main computation derives the group number of an acceleration vector and uses the group number along with the model data `a`, `b`, and `pi` to update the belief states `f` and `s`. Because of the accuracy limitation of an 18-bit fixed point arithmetic, we have to prevent the belief state values from getting too small by shifting the values left by some amount. To maintain correctness, all belief state values in all models must be shifted by the same amount. Thus, the main computation returns the largest value in its belief state `fmax` to the connector. Once the connector collects `fmax` values from all gesture classifiers, it determines the shifting value `shift`. The dashed arrows depicts the data flow for updating the belief state `s` to be equal to `f` shifted left by `shift` bits. After 1,000 rounds, we print out the final probabilities of all gestures and reset the classifiers. The dotted-dashed arrows depict the data flow for obtaining the final probabilities.

6.2 Utilizing New Compiler Extensions

Actorization. Actor functions are used throughout the program, as evidenced in Figure 5, which highlights actor cores. We use actors to both reduce communication and code size for I2C cores. The crystal timing node 713 sends a request to node 608 to read accelerometer data. Node 608, in turn, triggers node 708 to start the communication with the accelerometer. Node 708 also sends a request to node 709 to wait for the next clock edge and then sends a signal back. Within a gesture classifier module, we also use actors to reduce code size for cores that store large model data arrays or perform many computations. The use of actors inside the gesture classifier incurs more communication between cores, but it is crucial to make the application fit in a small distributed memory.

Parallel Module. To make two gesture classifiers run in parallel, we use the parallel module construct to create two gesture classifiers and place them on different regions of the chip: core 001–307 and core 008–314. Each module instance contains gesture-specific model data for a quantizer and an HMM classifier, which is parameterized during the creation of the module instance.

Programmer-Specified Program Layout. Last, we use the new language construct to specify the program layout displayed in Figure 5 to the compiler in order to make the code fits in GA144.

7. Experimental Results

In this section, we evaluate the impacts of the compiler extensions on the gesture recognition application. First, we evaluate the accuracy of the application running on GA144. Second, we evaluate the impact of being able to compile the application for GA144. Third, we evaluate the individual impact of each extension we introduce.

7.1 Classification Accuracy

In this experiment, we verified that the compiled application on GA144 is able to predict hand gestures accurately. We asked two participants to perform circle and flip-roll gestures, 11 times for each gesture. The prediction accuracies for the two participants were 90.91% and 80.82%. We obtained a similar prediction accuracy to Wiigee's (the original implementation that our application was based on), which ranges from 84% to 94% [16]. The demonstration of the application running on GA144 can be viewed at: `https://youtu.be/GD9lVm1ZyNQ`

7.2 GA144 vs. MSP430

Next, we evaluated the impact of being able to run the application on GA144. If running the application on other processors that do not require difficult programming partitioning and a careful program layout design were as good as running the application on GA144, we would not have to bother developing the compiler extensions we have introduced. For this purpose, we selected MSP430, a widely-used ultra low-power micro-controller to compare GA144 against.

7.2.1 Implementation for MSP430

We implemented the same application for MSP430F5529 with 128-kB flash, 8-kB RAM, and up to 25 MHz CPU speed. We used 16 bits to represent a fixed-point number instead of 18 bits as implemented on GA144. We interfaced the ADXL345 accelerometer to MSP430 via I2C protocol. Note that we used the SensorTag accelerometer for GA144. Although the accelerometers were different, we implemented the same I2C protocol on both GA144 and MSP430. Therefore, the two processors performed exactly the same activities to communicate with the accelerometers. The accelerometers were powered by a different energy source from the one that powered the processors, and we excluded the energy consumed by the accelerometers when comparing GA144 and MSP430. Therefore, we believe that our comparison was fair.

7.2.2 Experimental Results

We measured the energy consumption for one round of classification. In each round, the application read an (x,y,z) acceleration vector and updated the belief states of the two gesture classifiers if the input vector passed the filter. Reading the accelerometer required many I/O activities; whereas, filtering accelerometer values and updating the belief states required a lot of computations. Thus, we measured the energy consumptions of the two tasks separately to compare the performance of GA144 and MSP430 for the different types of usages. We powered GA144 with 1.8 V and MSP430 with 2.2 V, as these are the typical voltages. We powered the accelerometer from a different power source because we are only interested in energy consumption by the processors. We ran each task in a loop hundred to hundreds-of-thousand times to measure the average current drawn (using a multimeter with a microamp precision) and completion time by each processor.

Table 1 reports completion time and energy consumption of running one round of classification on GA144 and MSP430. Overall, GA144 was 18.8x more energy-efficient and 23.2x faster than MSP430. If we look at each task separately in Table 2, GA144 was excellent at performing a computationally heavy task: filtering and classification. It was three orders of magnitude faster and 71.1x more energy-efficient than MSP430. Recall that the application should run 200 rounds of classification in one second; each round takes 5 milliseconds. However, MSP430 took 60 milliseconds to update the belief states, if the accelerometer values passed the filter. Therefore, MSP430 was not able to run 200 rounds in one second consistently like GA.

On the other hand, MSP430 was better at the accelerometer reading task, 2.2x more energy-efficient than GA144. However, we believe that we can further optimize GA144 for this task. In our implementation, the main I2C core, which interacts with the accelerometer, waits for its I/O pin to become high by spinning in a loop. Therefore, the program can be optimized by avoiding this loop. Unfortunately, the main I2C core is completely full, and we need more space for this optimization. To make the optimized code fit in this core, the compiler will need advanced transformations that exploit transferring code (not just data) between cores.

Processor	Execution time per round		Energy consumption per round	
	absolute (ms)	relative to GA144	absolute (μJ)	relative to GA144
GA144	2.639	-	2.231	-
MSP430	61.346	23.2x	41.920	18.8x

Table 1. Total execution time and energy consumption per one round of classification

Processor	Accelerometer Reading				Filter & Classification			
	power (mW)	time (ms)	energy (μJ)	energy relative to GA144	power (mW)	time (ms)	energy (μJ)	energy relative to GA144
GA144	0.633	2.610	1.652	-	19.957	0.029	0.579	-
MSP430	0.565	1.346	0.760	0.46x	0.686	60.00	41.160	71.1x

Table 2. Energy consumption per each task per one round of classification

Features used	Number of cores	Overflowed cores	Biggest core (words)	Total words used
Actor + layout	82	0	64	2,609
No actor + layout	90	12	87	3,152
No actor + no layout	82	20	89	3,071

Table 3. Size of generated code. Each core can store up to 64 words of data and program.

7.3 Impacts of Compiler Extensions

Without the extensions we developed, we would not be able to run the application on GA144 because the code could not fit in its memory. Table 3 shows the impact of the new extensions on the sizes of generated programs. 'Actor' indicates actorizing some functions (yielding hybrid partitioning strategy), and 'layout' indicates specifying program layout. When we actorized program appropriately and specified the program layout design ('actor + layout'), the application fit on every core. However, when all functions were actors, the compiler failed to generate code because it could not find a feasible routing between every communicating pair of cores. Recall that actors impose additional constraints to the routing algorithm; the more actors, the more obstacles the routing algorithm has to avoid. When we did not actorize any function but specified the program layout ('no actor + layout'), the compiler successfully generated code, but 12 cores overflowed, and the total number of cores used was 90 instead 82. When we did not actorize any function and did not specify the program layout ('no actor + no layout'), 20 cores overflowed, and the biggest core overflowed by 25 words. When we actorized some functions but did not specify the program layout, the compiler failed to find feasible routing between some cores. We excluded the failed versions from the table. In summary, these results reveal that both hybrid partitioning strategy and programmer-specified layout are crucial for compiling code for a very limited-resource environment.

Furthermore, without the parallel module construct, we would have to duplicate the classification code and explicitly assign every data and computation to a partition to achieve parallelism, as explained in Section 4.1.1. Hence, the parallel module construct tremendously increased our productivity.

8. Future Work

There are several parts of the Chlorophyll compiler that can be be improved even further. First, we would like to make the compiler actorize programs automatically because this task may not be very intuitive to programmers. Second, we would like to improve the partitioning, layout, and routing algorithms to be smarter. As we discussed earlier, in order to fit the gesture recognition application on GA144, the partition type inference, layout synthesizer, and routing algorithm have to take the space occupied by communication code into account, which is very difficult. An alternative way is to iteratively improve the solutions. When the machine code generated from the compiler does not fit in some cores, the compiler compiles the program again. This time, it can learn from the previously generated code how much communication code may be inserted if it partitions the programs, mapping partitions to cores, and generate communication paths the way it has done before so that it can adjust its strategies accordingly. Lastly, the debugging support can be improved. Specifically, a tool that visualizes program layout, data location, and data routing will be extremely useful when programming a distributed computing system in such a small granularity.

9. Related Work

The SPMD program partitioning strategy was proposed by Callahan and Kennedy [4]. They pointed out that the partitioned program can be described as SPMD because in the most naively compiled code, every node executes the same program but performs computation on distinct data items. The Chlorophyll compiler splits the code more efficiently, similar to an ideal compiler described by Callahan and Kennedy, such that empty statements are removed. Many Distributed Fortran compilers apply this partitioning strategy with an *owner computes rule* to partition programs such that computations happen at the same place where the left-hand-side data element lives [3, 13].

The actor program partition strategy is similar to the strategy the X10 compiler uses for handling place change when programmers use the construct `at` to specify where the data and the computations inside the scope of `at` live and happen [18]. However, our language construct for actorization is very different from the X10 construct. As discussed in Section 3.3.1, since GA144 cores are very small, multiple cores may be required to perform one functional task. Therefore, we provide the construct that is suitable for actorizing a task performed by multiple cores. In contrast, X10 targets much bigger nodes, so a task can normally fit in one node. Thus, the `at` construct seems appropriate for X10's use cases. We borrow the name *actor* and its concept of reacting upon a request to perform a task from the actor model for a concurrent computation [9]. However, we use the actor concept to avoid duplicating control flow constructs instead of obtaining concurrency.

An existing constraint-based approach can solve partitioning, placement, and routing problems simultaneously using Integer Lin-

ear Programming (ILP) to map a computation DAG to a graph representing hardware's structure [14]. However, this technique cannot be applied to our partitioning and layout problems because it assumes a simple program control flow with no loops, as it targets scheduling problems at a finer granularity. Consequently, it does not address the problem of partitioning control statements.

Type systems have been used in many distributed programming languages to ensure properties of interest. For example, programmers can use a type system to infer the localization of expressions onto processors in synchronous dataflow programs [6]. Some are used to identify what portions of programs can be safely executed in parallel [2, 17]. Chlorophyll's partition type system differs from the type systems built purposefully for parallelism because it is originally designed for partitioning data and computations. However, partitioning and parallelism are inevitably related.

10. Conclusion

As energy efficiency forces processors to become simpler, compilers have to become smarter. Our work presents a method for making a compiler smarter through constraint solving, classical program transformations, and programmer-specified insights. We introduced new extensions to Chlorophyll to make complex applications run on a very small distributed-memory multicore processor and to allow programmers to express parallelism. First, we compared actor and SPMD partitioning strategies and extended the Chlorophyll language to allow programmers to control when to use which partitioning strategies. With this extension, the compiler can partition program structures using a mix of both strategies. Second, we added light-weight extensions to the compiler's spatial constraints to allow programmers to express parallelism and program layout. As a result, the gesture recognition application was able to fit and run on GA144. We demonstrated the benefit of being able to compile this application for GA144 by comparing GA144 against MSP430 and showed that we saved energy by 19 times when running on GA144.

Acknowledgments

We would like to thank Charley Shattuck and Greg Bailey from GreenArrays Inc and Rimas Avizienis for help on getting the application to run on GA144. We would like to thank Heather Levien for help on editing this paper. This work is supported in part by Qualcomm Innovation Fellowship, Microsoft Research Fellowship, Grants from NSF (CCF–1139138, CCF–1337415, and ACI–1535191), U.S. Department of Energy, Office of Science, Office of Basic Energy Sciences Energy Frontier Research Centers (FOA–0000619), and DARPA (FA8750–14–C–0011), as well as gifts from Google, Intel, Mozilla, Nokia, and Qualcomm.

References

[1] R. Avizienis and P. Ljung. Comparing the Energy Efficiency and Performance of the Texas Instrument MSP430 and the GreenArrays GA144 processors. Technical report, 2012.

[2] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *SC*, 2012.

[3] Z. Bozkus, A. Choudhary, T. Haupt, G. Fox, and S. Ranka. Compiling hpf for distributed memory mimd computers. In *The Interaction of Compilation Technology and Computer Architecture*. 1994.

[4] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*.

[5] D. T. Connolly. An improved annealing scheme for the QAP. *European Journal of Operational Research*, 1990.

[6] G. Delaval, A. Girault, and M. Pouzet. A type system for the automatic distribution of higher-order synchronous dataflow programs. In *LCTES*, 2008.

[7] GreenArrays. *Product Brief: GreenArrays Architecture*, 2010. URL `http://www.greenarraychips.com/home/documents/greg/PB002-100822-GA-Arch.pdf`.

[8] GreenArrays. *Appplication Note AB012: Controlling the TI SensorTag with the GA144*, 2013. URL `http://www.greenarraychips.com/home/documents/greg/AN012-130606-SENSORTAG.pdf`.

[9] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, 1973.

[10] A. H. Karp. Programming for parallelism. *Computer*, 20(5):43–57, May 1987.

[11] P. Ljung. Welcome to the dark side of computing, 2011. Presented at ParLab Summer Retreat, University of California, Berkeley.

[12] H. Massalin. Superoptimizer: a look at the smallest program. In *ASPLOS*, 1987.

[13] J. Merlin. Techniques for the automatic parallelisation of 'distributed fortran 90'. Technical Report SNARC 92-02, Southampton Novel Architecture Research Centre, 1992.

[14] T. Nowatzki, M. Sartin-Tarm, L. De Carli, K. Sankaralingam, C. Estan, and B. Robatmili. A general constraint-centric scheduling framework for spatial architectures. In *PLDI*, 2013.

[15] P. M. Phothilimthana, T. Jelvis, R. Shah, N. Totla, S. Chasins, and R. Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *PLDI*, 2014.

[16] T. Schlömer, B. Poppinga, N. Henze, and S. Boll. Gesture recognition with a wii controller. In *International Conference on Tangible and Embedded Interaction*, 2008.

[17] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken. Regent: A high-productivity programming language for hpc with logical regions. In *SC*, 2015.

[18] M. Takeuchi, Y. Makino, K. Kawachiya, H. Horii, T. Suzumura, T. Suganuma, and T. Onodera. Compiling x10 to java. In *ACM SIGPLAN X10 Workshop*, 2011.

[19] F. Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994.

[20] Xilinx. FPGA Design Flow Overview, 2008. URL `http://www.xilinx.com/itp/xilinx10/isehelp/ise_c_fpga_design_flow_overview.htm`.