Chlorophyll: Synthesis-Aided Compiler for Low-Power Spatial Architectures

Blind for review

Abstract

We develop Chlorophyll, a synthesis-aided programming model and compiler for GreenArrays 144, an extremely minimalist low-power spatial architecture that requires partitioning the program into fragments of no more than 256 instructions and 64 words of data. This processor is 100x more energy efficient than its competitors, but it can currently be programmed only by using a low-level stack-based language.

Chlorophyll allows programmers to provide human insight by specifying partial partitioning of data and computation. The compiler relies on synthesis, sidestepping the need to develop classical optimizations, which may be challenging given the unusual architecture. To scale synthesis to real problems, we decompose it into smaller synthesis subproblems—partitioning, layout, and code generation. We show that the synthesized programs are no more than 80% slower than highly optimized expert-written programs and faster than programs produced by a heuristic, non-synthesizing version of our compiler.

Categories and Subject Descriptors E.2.1 [*Software notations and tools*]: General programming languages—Language features, Compilers; K.2.2 [*Parallel computing methodologies*]: Parallel programming languages

Keywords Program Synthesis, Spatial Architectures

1. Introduction

Energy requirements have been dictating simpler processor implementations, with more energy for computation and less for processor control. Simplicity is already the norm in low-power systems, where the 32-bit ARM dominates the phone computer class [32]; the 16-bit TI 430MSP is a typical example of a low-power embedded controller; and the even simpler 8-bit Atmel AVR controller powers Arduino [1]. Naturally, energy efficiency comes at the cost of computing capability.

The GreenArrays (GA144) processor is a recent example of a low-power spatial processor, composed of many simple replicated processing elements [13]. Likely the most energy-efficient commercially available processor, it consumes 9 times less energy and simultaneously runs 11 times faster than TI MSP430, the low-power microcontroller, in the finite impulse response benchmark [2]. As expected, the energy-efficiency of GA144 comes at the cost of programmability. Programs must be meticulously partitioned and laid out onto the physcial cores.

Future low-power many-core processors will likely be similarly minimalistic, going to the extremes in offloading low-level programming tasks to the programmer or the compiler. They will likely have many cores, simple interconnects, and radically different ISAs. In this paper, we introduce a new programming model and a synthesis-based compiler for such processors. We use GA144 as our primary hardware target because its extremely minimalistic architecture stresses the demands on the programming tool chain—if we can build a synthesizer for this processor, we are likely able to do so for other low-power processors.

1.1 GreenArrays Low-Power Spatial Processors

GA144 is a stack-based, 18-bit processor consisting of 144 independent cores, with no clock and no shared memory [13, 14]. It consumes the smallest amount of energy per instruction compared to other commercially available architectures [23]. Many GA144 applications have been developed, which are implemented manually in a low-level language. Using the low-level language to program the chip comes with many challenges.

First, each core can communicate only with its neighbors, using blocking reads and writes. There are no message buffers. To communicate with distant cores, the programmer must intersperse communication code into the computation code of a core, carefully avoiding deadlocks and race conditions. Second, each core contains only a tiny memory and two circular stacks (one for storing data and one for storing return addresses), with less than 100 18-bit words in total per core. The program and data structures must thus be partitioned into fine-grain tasks. For instance, MD5 hash needs to be partitioned across 10 cores on GA144, even when heavily optimized [15]. Third, GA144 is an 18-bit architecture. Wider bitwidths must be handled in software. Last, the machine code of GA144 is a stack-based language [14]. Most programmers have been used to implement quality stack code.

1.2 Challenges and Our Solutions

Our new programming model and compiler make an important step towards overcoming the following challenges.

First, classical compilers may not be able to bridge the abstraction gap of low-power parallelism. When the desire for energy efficiency sacrifices programmability features of the hardware—such as hardware-controlled caches—the compiler must bridge a greater abstraction gap. This problem cannot be easily addressed by classical compilation for two reasons: (i) it may take a decade to build a mature compiler with optimizations for the target hardware; and (ii) low-power architectures will be under active investigation for a while, representing a moving target for the compiler developer, delaying compiler development.

Second, programmers prefer control over hardware but at a high level. To optimize their programs, programmers can decide how to partition data structures and code but do not want to deal with low level details of the communication code the partitioning induces. We develop a programming model in which they can selectively partition key data structures and code, leaving the remaining partitioning as well as communication code generation to the synthesizer.

Third, applying program synthesis to large problems may not scale. Algorithms developed for program synthesis operate on whole programs (or whole program fragments but not on decompositions of larger programs) [16, 29, 31]. In order to scale synthesis to large programs, problems must be decomposed into smaller synthesis problems. Our approach is to break the problem down into logical sub-problems: program partitioning, layout and routing, and optimized generation (and code separation which is a classical compilation problem). The resulting synthesis-aided compiler uses a suitable solver for each subproblem.

In summary, we make the following contributions:

- We develop a programming model that allows the programmer to optionally partition some data and code. The model allows fine-grain partitioning, including slicing integers across multiple cores.
- We design and evaluate a compiler that solves three consecutive synthesis problems. Our design shows how to decompose synthesis to solve a practical synthesis problem.
- We build a compiler that generates optimized code without manually implemented optimizations.
- Ours is the first compiler for the extremely minimalist GreenArrays architecture, with performance within 1.8x of manually written programs. Without our compiler, the only option to run on GA144 is an interpreter with several orders of magnitude slowdown, and it would negate all energy benefits.

2. Overview

Our proposed programming model provides flexibility to partition data and computation. Programmers can optionally specify partitions of data and operations by annotating them with *partition types* when they have good insight into how to partition programs. To experiment with a different partitioning scheme, they only need to change the partition annotations without touching other parts of the program. The communication is implicit, eliminating the need for programmers to handle it correctly.

The problem of compiling a high-level program into machine code is decomposed into 4 main subproblems: partitioning, layout and routing, code separation (a classical compilation technique), and code generation.

Step 1 (partition) The input to this step is a source program with partial partition annotations. The compiler partitions the input high-level program by completing the annotations that the programmer has not specified. Annotations define the logical core in which code and data resides. For example, the input source code

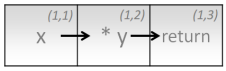
```
int@0 mult(int x, int y) { return x * y; }
```

conveys that the programmer wants the function to deliver the return value to core 0. Programs are partitioned such that each partitioned program piece fits in a core and that a static over-approximation of messages between partitions is minimized. Assuming that each core is very tiny and has only 16 words of memory (rather than 64 words in reality), the output from the partitioner is the following program.

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

We implement a partitioning synthesizer that minimizes the number of messages. It is a constraint solver that not only reasons about the number of messages but also accounts for the size of communication code.

Step 2 (layout) The layout synthesizer maps program partitions onto actual cores such that a better approximation of communication cost is minimized. For each pair of cores, the communication path (routing) is determined. This layout problem can be viewed as a well-known Quadratic Assignment Program (QAP) which can be solved exactly or approximately. We choose the Simulated Annealing algorithm for our layout synthesizer because it is one of the fastest techniques that produces nearly optimal solution. Given the partitioned $mult$ function from the previous step, the figure below is the result of this step.



Step 3 (code separation) The separator splits the original program AST into multiple per-core ASTs that communicate with each other using sends and receives. We guarantee that the resulting separated programs are deadlock-free by enforcing the no instruction reordering rule within each core. Given the partitioned program and layout from step 1 and 2, we obtain the program fragments shown below.

```
// 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); }
```

Step 4 (code generation) The code generator naïvely compiles each per-core AST into machine code without optimizations. A synthesizer searches the space of all instruction sequences to find a correct and fast (or short) sequence. Although the superoptimizer is allowed to reorder evaluations, it preserves the order of send and receive operations. This restriction is enough to prevent deadlocks. The compiler applies a sliding window technique to adaptively merge small code sequences into bigger ones and input it to the synthesizer. The compiler keeps a database of optimized code it has found, resulting in quick compilation of programs that have been previously compiled.

Partitioning, layout, and code generation for radical architectures are difficult problems for traditional compilers. However, we show that each of them admits more natural solutions using synthesis techniques. The rest of the paper is organized as follows. Section 3 shows how to obtain a partitioning synthesizer by implementing an interpreter to calculate number of communications and a partition space checker. Section 4 describes building a layout synthesizer using existing layout algorithms. Section 5 describes implementation of a simple program transformation for code separation. Section 6 shows how to obtain optimized code without implementing the optimizations explicitly. Finally, Section 7 presents evaluation, Section 8 describes related work, we conclude in Section 9.

3. Programming Model for Partitioning

Chlorophyll language is designed to ease reasoning about code partitioning and to obviate explicit communication code. We extend the type system with a *partition type* which enables the *partitioning synthesizer* to optimally infer unspecified partitions. In this section, we introduce Chlorophyll language, its type system, and the partitioning process.

3.1 Language Overview

Chlorophyll syntax is a subset of C with *partition annotation* for specifying the partitions data and operation live in. In order to make fine-grain partitioning possible, we track the partition of every piece of data and operation using these annotations. Figure 1(a) shows `LeftRotate` implemented in Chlorophyll. On line 15, we assign the partition of variable r to 6 by annotating its declaration. On line 12, we assign the partitions of distributed array x such that $x[0]$ to $x[31]$ live in partition 0, and the rest in partition 1. On line 18, operation $+$ is assigned to partition 6. On line 17, operation $-$ is assigned to $place(z[i])$; when $0 < i < 32$, operation $-$ at partition 4 is executed, and when $32 < i < 64$, operation $-$ at partition 5 is executed. Note how most of the data and operations in the program are left unannotated, but they will be automatically inferred by the partitioning synthesizer.

3.2 Programming Constructs and Space

Constants, variables, arrays, operators, and statements all take up space in memory. Most operations and statements take up a constant amount of memory, so we can estimate the space occupied with a simple lookup table. However we have to handle control flow constructs, function calls and arrays specially.

Control Flow Constructs (for, while, and if-else) When the body of a control flow construct is spread across many partitions, called

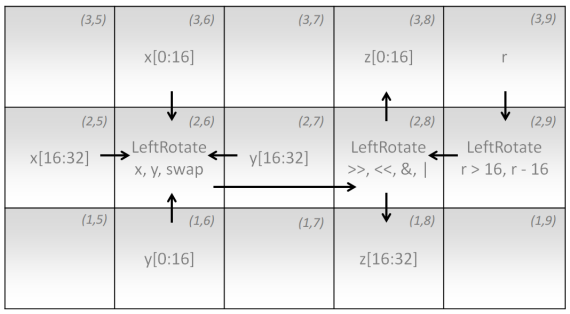
```
1  int leftrotate(int x, int y, int r) {
2    if(r > 16) {
3      int swap = x;
4      x = y;
5      y = swap;
6      r = r - 16;
7    }
8    return ((y >> (16 - r)) | (x << r)) & 65535;
9  }
10
11 void main() {
12   int@{[0:32]=0,[32:64]=1} x[64];
13   int@{[0:32]=2,[32:64]=3} y[64];
14   int@{[0:32]=4,[32:64]=5} z[64];
15   // x[0] to x[31] live at partition 0,
16   // x[32] to x[63] live at partition 1, and so on.
17
18   int@6 r = 0;
19   for (i from 0 to 64) {
20     z[i] = leftrotate(x[i],y[i],r) -@place(z[i]) 1;
21     r = r +@6 1; // + happens at partition 6.
22     if (r > 32) r = 0;
23   }
24 }
```

(a) Input source code written in Chlorophyll.

```
int @7 leftrotate(int@8 x, int@8 y, int@7 r) {
  if(r >@8 16) {
    int@8 swap = x;
    x = y;
    y = swap;
    r = r -@8 16;
  }
  return ((y >>@7 (16 -@7 r)) |@7 (x <<@7 r)) &@7 65535;
}

void main() {
  int@{[0:32]=0,[32:64]=1} x[64];
  int@{[0:32]=2,[32:64]=3} y[64];
  int@{[0:32]=4,[32:64]=5} z[64];
  int@6 r = 0;
  for (i from 0 to 32) {
    z[i] = leftrotate(x[i]!8,y[i]!8,r!7)!4 -@4 1; // !8 means send to partition 8
    r = r +@6 1;
    if (r >@6 32) r = 0;
  }
  for (i from 32 to 64) {
    z[i] = leftrotate(x[i]!8,y[i]!8,r!7)!5 -@5 1;
    r = r +@6 1;
    if (r >@6 32) r = 0;
  }
}
```

(b) Output from partitioner when memory is 64 words.



(c) Output from layout synthesizer.

```
void leftrotate(int x,int y) {
  if(read(E)) {
    int swap = x;
    x = y;
    y = swap;
  }
  write(E,y);
  write(E,x);
}

void main() {
  for(i from 0 to 32) {
    leftrotate(read(N), read(S));
  }

  for(i from 32 to 64) {
    leftrotate(read(W), read(E));
  }
}
```

(d) Program at core (2,6) after code separation.

Figure 1. Example program written in Chlorophyll, and intermediate results from partitioning, layout, and code separation steps.

body partitions, the control flow construct needs to be placed in all of these partitions as well. The result of the condition expression, for example, *x +@2 y*, is at partition 2. This evaluated value is sent to all the body partitions, each of which in turn use the received value as its condition. Chlorophyll only allows the loop bound expressions $e1$ and $e2$ in *for (i from e1 to e2)* {...} to be constants. This restriction allows Chlorophyll to produce more efficient code. More specifically, each of the body partitions has and uses its own copy of i. This reduces the number of communications between partitions. However, there is no restriction on conditions of while loops.

Function A function's partitions are defined by the partitions of variables and operations inside it. Consider a program with a function call inside a control flow construct. The control flow construct needs to be placed in every partition containing any operation and variable inside the (control flow) construct as well as inside the function body.

Array An array is categorized as:

- ***Non-distributed array*** only lives in one partition. The index to this type of array has to live at the same partition as the array itself.
- ***Distributed array*** lives in multiple partitions. For example, array x, y, and z in `LeftRotate` are distributed arrays. This type of array can be indexed by affine expressions of surrounding loop variables and constants. Accessing this type of array requires no communication because the indexes are expressions of loop variables which live in all body partitions. Chlorophyll currently only supports distributed arrays that are indexed by this type of expression.

Currently, Chlorophyll does not handle recursive calls, multidimensional arrays, and array accesses whose index expressions use non-loop variables.

3.3 Partition Type and Typing Rules

Partition types can be inferred by the partitioning synthesizer or specified by the programmer. We present a simplified version of our complete type system to convey the core idea. Types in Chlorophyll can be expressed as follows:

$\tau :=$ `τ@ρ` $\quad\quad$ $\tau :=$ `val | int | void`
$\rho :=$ N `| any |` ρ_{dist} $\quad\quad$ $\rho_{dist} :=$ $\{(N,)^+\}$
$N :=$ `natural number`

Our type consists of data type τ and partition type ρ. For simplicity, our data types only include `int` and `void`. ρ_{dist} is a type of distributed array. The typing rules shown in Figure 2 (omitting trivial rules due to space limitation) essentially enforce that operands and operators are in the same partition. *Partition subtype* rule says that an expression with partition type '`any`' can be used when any partition type '`N`' is expected.

$$\frac{}{\text{val} < \text{int}}\ [\text{basic subtype}] \qquad \frac{}{\text{any} < \text{N}}\ [\text{partition subtype}]$$

$$\frac{\tau_1 < \tau_2 \qquad \rho_1 < \rho_2}{\tau_1@\rho_1 < \tau_2@\rho_2}\ [\text{subtype}] \qquad \frac{}{\Gamma \vdash n : \text{val@any}}\ [\text{const}]$$

$$\frac{x : \tau@\rho \in \Gamma}{\Gamma \vdash x : \tau@\rho}\ [\text{variable}] \qquad \frac{i : \text{val@any} \in \Gamma}{\Gamma \vdash x : \text{val@any}}\ [\text{iterator}]$$

$$\frac{\Gamma x : \tau@\{\rho\} \in \Gamma}{\Gamma \vdash x : \tau@\{\rho\}}\ [\text{array}] \qquad \frac{\Gamma x : \tau@\{\rho_1, \rho_2, ..., \rho_n\} \in \Gamma}{\Gamma \vdash x : \tau@\{\rho_1, \rho_2, ..., \rho_n\}}\ [\text{dist array}]$$

$$\frac{\Gamma \vdash e_1 : \tau_1@\rho_1 \quad \Gamma \vdash e_2 : \tau_2@\rho_2 \quad \tau_1@\rho_1 < \tau@\rho \quad \tau_2@\rho_2 < \tau@\rho}{\Gamma \vdash e_1 \ \ \text{op}@\rho\ e_2 : \tau@\rho}\ [\text{op}]$$

$$\frac{\Gamma \vdash x : \tau@\{\rho\} \qquad \Gamma \vdash e : \tau_e@\rho_e \qquad \tau_e@\rho_e < \text{int}@\rho}{\Gamma \vdash x[e] : \tau@\rho}\ [\text{access array}]$$

$$\frac{\Gamma \vdash x : \tau@\{\rho_1, ..., \rho_n\} \qquad \Gamma \vdash e : \text{val@any} \qquad e \downarrow v}{\Gamma \vdash x[e] : \tau@\rho_v}\ [\text{access dist-array}]$$

$$\frac{\Gamma \vdash e : \tau@\rho_1}{\Gamma \vdash e!\rho_2 : \tau@\rho_2}\ [\text{send}]$$

Figure 2. Typing rules

In *access dist-array* rule, the type checker needs to evaluate e during compile time. This is possible because our type system ensures that the index to a distributed array is an expression of loop variables and constants, and the language restricts that the loop bounds are finite. Thus, we can break a loop that iterates over a distributed array into multiple loops such that each of them accesses a chunk of array that lives at only one partition. For example, the loop in `LeftRotate` is broken into two loops: one iterating from 0 to 32 and another from 32 to 64. This process is done by *loop fission* procedure, described later, before type checking.

! is an operation for sending data from one partition to another. It is the only operation that accepts an operand whose partition type does not have to be subtype of the output partition type. The compiler automatically generates this operator during type checking and inferring, so programmers are not required to insert any ! in the source code.

3.4 Partitioning Process

The partitioning synthesizer is constructed from 1) the communication interpreter and 2) the partition space check.

3.4.1 Interpretation for Number of Communications

Let $Comm(P, \sigma, x)$ be a function that counts number of communications between partitions given a program P with complete annotated partitions Φ and a concrete input x. $MaxComm(P, \sigma)$ = $\max_{x \in Input} Comm(P, \sigma, x)$, where $Input$ is a set of all valid inputs to the program, under the assumption that while loop is executed a certain number of times (100 in our current implementation). $MaxComm$ essentially computes the maximum number of communications considering all program paths. $MaxComm$ interpretation of most expressions and statements is straightforward; the communications count is equal to sum of its components' counts. Loops multiply the count. ! is the only statement that contributes to new communication counts.

3.4.2 Partition Space Checking

Operations and statements all take up space in the partitions they belong to. If-else and loop take up space in the body partitions. Each unique partition in the body partitions reduces its unused space only once for each control statement, even if the same partition appears more than once inside the body. Communication operations such as read and write also occupy space. Given a program with complete partition annotations, the partition space checker can compute how much space is used in each partition. The compiler only accepts the program if the occupied space in every partition is not more than the amount of memory available in a core.

3.4.3 Partitioning and Loop Fission Synthesizer

Partitioning Synthesizer In this section, we show how to use the communication interpretation and partition space checking to automatically infer partition annotations when unspecified.

We implemented the interpreter and the partition space checker using Rosette, a tool for building a light-weight synthesizer [33]. We represent a specified partition annotation as a concrete value and an unspecified partition annotation as a symbolic variable. Given a fully annotated program (one with all concrete partitions), the result from the interpretation is a concrete value, and the partition space checker just verifies if the memory constraint holds. Given a partially annotated or unannotated program (program with some or all symbolic partitions), the result from the interpretation is a formula in terms of the symbolic variables, and the partition space check becomes a constraint depending on symbolic variables.

Once we obtain a formula of communication count with an additional partition space constraint, we query Rosette's back-end solver to find an assignment to the symbolic partitions such that the space constraint holds. If the solver returns with a solution, we can further optimize communication count by asking the solver the same question but with the extra constraint that the count has to be less than the best one found so far.

Loop Fission Synthesizer Since the traditional way to perform loop fission, using polyhedral analysis, is not easy to implement, we implemented the loop fission synthesizer using Rosette similar to how we implement the partitioning synthesizer. Consider this prefixsum program.

```
int@{[0:5]=0,[5:10]=1} x[10];
for (i from 1 to 10)  x[i] = x[i] + x[i-1];
```

We first duplicate the loop into k loops and replace the loop bounds with symbolic values. Let k be 3 in this particular example. Let the first loop iterate over i from a_0 to b_0, the second loop from a_1 to b_1, and so on. We need to check that $a_0 = 1, b_2 = 10$, $a_{i+1} = b_i$, and every $x[i]$ under the loop bounds belongs to one partition as well as $x[i-1]$. We implement the checker as if the bounds are concrete. When the bounds are unknown, they become symbolic values, and the checking conditions become constraints given to the solver. Finally, the solver outputs one feasible solution for loop bounds. In this particular example, the output is

```
for (i from 1 to 5)
  x[i] = x[i] + x[i-1]; // x[i] at 0, x[i-1] at 0
for (i from 5 to 6)
  x[i] = x[i] + x[i-1]; // x[i] at 0, x[i-1] at 1
for (i from 6 to 10)
  x[i] = x[i] + x[i-1]; // x[i] at 1, x[i-1] at 1
```

The final output is the solution with the smallest possible k.

3.5 Example and Rationale

Figure 1(b) shows the result after partitioning the program in Figure 1(a) when memory per core is 64 words. Notice that ! is automatically inserted into the program. In bigger processors where each core has 128 words of memory, the function `LeftRotate` can fit entirely on one core, so this step will annotate everything within the function with the same partition.

While the reader may question why we need annotation at all, compilers should take advantage of the fact that programmers generally have a good idea of how to partition data and operations. We designed this tool with the philosophy that the programmer and the compiler have different strengths and that we should let them

spend time on the tasks they excel at. Allowing programmers to provide their insight also enables this partition approach to scale.

4. Layout

In this step, we map code partitions to physical cores by framing this *layout* problem as an instance of QAP, which can be described as follows.

Given a set F of facilities, a set L of locations, a traffic function $t : F \times F \to \mathbb{R}$, and a distance function $d : L \times L \to \mathbb{R}$, find the assignment $a : F \to L$ that minimizes the following cost function:

$$\sum_{f_1 \in F, f_2 \in F} t(f_1, f_2) \cdot d(a(f_1), a(f_2))$$

Our partition location problem can be viewed as an instance of QAP. Code partitions can be viewed as facilities, and the number of messages sent between each pair of code partitions is the flow between partitions. The distance matrix can be constructed from the Manhattan distance between each pair of cores. The solution to the QAP problem on these graphs is the assignment of partitions to cores that minimizes the total number of port reads during program execution.

This QAP problem can be solved by many techniques such as Branch and Bound search with pruning [21], Simulated Annealing (SA) [8], Ant System [10], and Tabu Search [30]. According to our preliminary experiment, the SA algorithm takes the least amount of time and generates the best solutions, which are often optimal.

Thus, we integrate the SA module as the layout synthesizer in our compiler. First, the compiler generates a flow graph f by adding more flow units to the graph when operation `!s` are encountered during the AST traversal. Second, the compiler calls an existing SA program, giving the flow graph as input to obtain the layout. In the final step, the compiler builds a routing table by selecting any shortest path for each pair of cores. The layout and routing of program in Figure 1(b) is shown in Figure 1(c).

5. Code Separation

In this step, the AST is separated into multiple ASTs communicating with each other using send and receives operations. We choose this communication scheme because GA does not support shared memory, and the processors can only communicate with their neighbors using synchronous reads and writes. In order to guarantee deadlock-freedom, we preserve the order of evaluations within each core (same order as the original AST). The rest of this section explains in detail how we perform code separation for different program constructs.

Basic Statements A program without control flow, function, and array is relatively simple to separate. While post-order traversing the AST, place sub-expressions at the partitions they belong to according to the partition types, and append the statements including communication code preserving the original order. For example, consider the program

```
int@3 x = (1 +@2 2) *@3 (3 +@1 4);
```

Partition 1, 2, and 3 map to cores (0,1), (0,2), and (0,3) arranged from west to east. The result from the separation is

```
partition 1:  write(E, 3 + 4);
partition 2:  write(E, 1 + 2); write(E, read(W));
partition 3:  int x = read(W) * read(W);
```

`E` and `W` represent east and west ports respectively. Notice that there is an implicit parallelism in this program where `1 + 2` and `3 + 4` are executed in parallel.

Function Invoking a function call in the original program corresponds to invoking multiple function calls at all the cores the function resides on. For instance, in this program

```
int@3 f(int@1 x, int@2 y) { return x +@2 y; }
int@3 x = f(1,2);
```

`f` is split across partition 1, 2, and 3 with the same layout as the previous example. When `f` is invoked, `f` at all 3 partitions are invoked as follows:

```
partition 1:  void f(int x) { send(E, x); }
              f(1);
partition 2:  void f(int y) { send(E, read(W) + y); }
              f(2);
partition 3:  int f() { return read(W); }
              int x = f();
```

Array The distributed array is stored in multiple cores. It is the main source of parallelism in our current programming model. For example,

```
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 program can run on the two different parts of the array in parallel.

Figure 1(d) shows the `LeftRotate` program at core (2,6) given the layout and routing shown in Figure 1(c).

6. Code Generation Using Modular Superoptimization

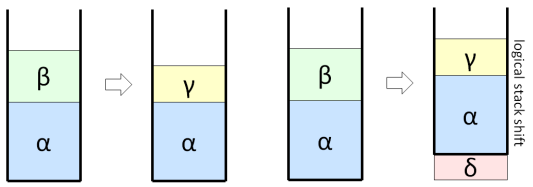
This section addresses the problem of translating single-core programs obtained in previous phases into optimized machine code. We solve the problem with an algorithm for modular superoptimization.

Typically, generation of optimized machine code is carried out with a bottom-up algorithm that optimally selects instruction sequences, performing local optimization along the way [11]. A bottom-up algorithm is well-suited for applications in which the optimizations are known and tend to be local, and in which we can determine all of the valid ways to generate code. This rewrite-based approach for instruction selection seems difficult to adopt for our target machine because it is unclear how to design rewrite rules sufficient for discovering the nonlocal optimizations leading to unusual instruction sequences that exploit special hardware features, such as the bounded-size cyclic data stack.

We sidestep the problem of having to come up with rules by replacing derivation of programs with a search for an optimized program in the space of candidate programs. One such approach is called superoptimization [16, 19, 24, 29]. It searches a space of all instruction sequences, verifying these candidate programs behaviorally against a few tests or a reference implementation, e.g., a naïvely generated code. If the candidate space includes a (correct) program that exploits the desired hardware-specific optimization, then the search-based code generator will find this program.

Superoptimization thus presents an attractive procedure for generation of optimal code for unusual hardware: (1) generate naïve code, to be used as a behavioral specification, using a simple code generator; and (2) synthesize optimal code equivalent to the specification using a superoptimizer. Unfortunately, superoptimizers scale to only about 25 instruction-long sequences [16, 19, 29], which is less than the size of basic blocks in programs which range from 1 to 100 instructions.

We found that it is non-trivial to apply superoptimization in our problem domain for two reasons:



(a) Basic stack specification (b) Improved stack specification

Figure 3. Specification on data stack.

- An obvious solution to scale superoptimization is to modularize by breaking down large code sequences (specifications) into smaller ones, superoptimizing the small segments, and then composing the optimal segments. However, arbitrarily chosen segment boundaries may break optimization opportunities.
- A straightforward method for specifying the input-output behavior of the program segments prevents some hardware-specific optimizations. For example, it may reject a segment that leaves garbage values on the stack when, in fact, legal to do so.

First, we explain the naïve code generator and terminology in Section 6.1. We then present the solutions to these two problems in the next two subsections. Finally, in Section 6.4, we describe our approach to encoding the space of candidates as a set of constraints.

6.1 Naïve Code Generation and Terminology

Chlorophyll naïve code generator translates each per-core high level program into an IR that preserves the program structure such as if-else and loop. The straight-line machine code is stored in many small units called *superoptimizable units*. A superoptimizable unit corresponds to one operation in the high-level program, therefore, containing a few instructions. Contiguous superoptimizable units can be merged into a longer sequence called *superoptimizable segment*.

We define a state of the machine as a collection of data stack, return stack, memory, and special registers. Sequences of instructions P and P' change the state of the machine from S to T and T' respectively. We define $P \equiv P'$ if $InterestRegion(T) \equiv InterestRegion(T')$. $InterestRegion$ extracts values that reside in the region of interest specific to that particular superoptimizable segment. Usually, the region of interest consists of entire memory, return stack, and data stack. Thus, the naïve code generator also need to collect the information on $InterestRegion$ for every superoptimizable unit. $InterestRegion$ of a segment can be calculated from $InterestRegion$s of its constituting units. Since we do not support recursion, it is possible to statically determine the depth of the stack at any point of the program. Since the physical stacks are bounded, our compiler rejects program that overflows either data or return stack at any point.

6.2 Specifications for Modular Superoptimization

We specify the input-output behavior of a segment using a sequence of instructions P and its $InterestRegion$. In this section, we will focus on the constraint on data stack since data stack is used for performing every kind of computation and may be used for storing data.

Instruction sequence P changes data stack from $\alpha|\beta$ to $\alpha|\gamma$ as shown in Figure 3(a). α is a region that contains intermediate values that will be used later. β is a region that needs to be removed from the stack, and γ is a region that needs to be added to the stack. P' is equivalent to P if it also produces $\alpha|\gamma$, and their stack pointers are pointing at the same location.

However, this specification is too strict, preventing some optimizations. For instance, consider the following example in Figure 4

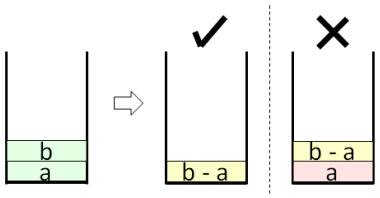


Figure 4. Basic specification rejects an instruction sequence that leaves a at the bottom of the stack.

when α is empty, and we want $b-a$ on top of the stack. The shortest sequence of instruction that has this behavior is 8 instruction long with the last 3 instructions that drop a remaining garbage value— a in this case— from the second entry from the top of the stack. Since α is empty, it is, in fact, legal to leave a at the bottom of the stack saving space of 3 instructions. However, this basic specification rejects the shorter sequence because its output data stack is $a|\alpha|b-a$ not $\alpha|b-a$.

We modify the specification such that instruction sequence P changes data stack $\alpha|\beta$ to $\alpha|\gamma$ as shown in Figure 3(b). P' is equivalent to P if it produces $\delta|\alpha|\gamma$ without any constraint on the stack pointer, where δ can be empty. Because GA stacks are circular, leaving garbage items at the bottom of the stack is essentially shifting the logical stack upward. This specification allows both upward and downward logical stack shifts.

6.3 Sliding Windows

Sliding windows technique adaptively merge superoptimizable units into a superoptimizable segment and give it as an input to the superoptimizer. This maximizes the opportunity of superoptimizing the program. Given a sequence of superoptimizable units, the sliding window proceeds as follows.

1. Starting from the sequence with an empty superoptimizable segment, merge superoptimizable units into the segment until the number of instructions are more than the upperbound.
2. Superoptimize the segment for optimal solution according to the cost function (e.g. segment length, or estimated runtime).
3. If a valid program is found, append the program to the global output, remove the merged superoptimizable units from the sequence, and repeat 1. If no better implementation is found, append only the first unit to the global output, remove the first unit from the sequence, and repeat 1. If timeout, unmerge the last unit, and repeat 2.
4. The process is done when there is no more superoptimizable unit in the sequence.

Alternatively, dynamic programming used in peephole superoptimization [4] can be applied for producing even result, but it requires much longer time than does the sliding windows technique. Dynamic programming is appropriate for the peephole superoptimization because the window size is only up to 3 instructions, while our window size is up to 16 instructions.

Sliding windows and dynamic programming technique only work when superoptimizable units are significantly smaller than superoptimizable segments. If our program behavior specification was defined by higher level IR that abstract stacks away instead of naïve machine code, we would not be able to apply this sliding windows technique.

6.4 Program Encoding

The state of a program at each step consists of two registers, data stack, return stack, memory, and stack pointers. Since each core can communicate to its four neighbors, we represent the data that the core receives and sends using incoming and outgoing channels, which are ordered lists of (data, neighbor port) pairs. Hence,

the program state needs to include incoming and outgoing channel pointers to indicate which data and port the core expect to receive and send at a particular point of the program. We also use communication channels to preserve the order of sends and receives to prevent deadlock. Each stack is represented by one large bitvector as well as memory and channels because the SMT solver we use (Z3 [9]) can handle large bitvector much faster than array of integers or an array of bitvectors. Each instruction in a program converts the old state into a new state. Therefore, we use static single assignment (SSA) form for the state of the program. We encode the formula of each instruction using a switch statement that alters the state of the program according to the value of the instruction.

Address Space Compression

Address space compression is necessary to make superoptimization scale. Each core in GA chip can store up to 64 18-bit words of data and instructions in memory. The generated code assigns each variable a unique location in memory; an array with 32 entries, thus, occupies 32 words of memory. When the formula generator translates program to formula, it discards the free memory space and only represents memory just big enough to contain all variables. The smaller the memory, the smaller the search space.

Arrays occupy a lot of memory space, but most of the time, an array is accessed with symbolic index during superoptimization. If the index to the array is an expression of one or more variables, the index is symbolic because it depends on the values of those variables. With this observation, before superoptimizing the program, we compress the memory of the input program by representing an array using 2 words of memory and modifying the variable and array addresses throughout the program accordingly. After we get a valid optimal output program, we decompress the output program, and ask the verifier if the decompressed output program is indeed the same as the original input program. Note that the verification is much faster than the synthesis, so we can verify programs with full address space in a reasonable amount of time.

7. Evaluation

In this section, we perform various experiments by running programs on the GA144 chip to verify our hypothesis that using synthesis provides advantages over the traditional way.

Hypothesis 1 *Partitioning synthesizer, layout synthesizer, superoptimizer, and sliding windows technique help generate faster programs than alternative techniques.*

We conduct experiments to measure the effectiveness of each component. First, to assess the performance of the partitioning synthesizer, we implement a heuristic partitioner that greedily merges an unknown partition into another known or unknown partition of a sufficiently small size when there is communication between the two. Second, to assess the performance of the layout synthesizer, we compare the default layout synthesizer that takes communication counts between partitions into account with the modified version that assumes the communication count of every communicating pair is equal to 1. Third, we compare performance of programs generated with and without superoptimization. Last, we compare sliding windows against fixed windows. For each benchmark, 5 different versions of the program are generated: (a) with sliding-windows superoptimization, partitioning synthesizer, and layout synthesizer (***sliding s+p+l***), (b) with fixed-windows superoptimization, partitioning synthesizer, and layout synthesizer (***fixed s+p+l***), (c) with no superoptimization, partitioning synthesizer, and layout synthesizer (***ns+p+l***), (d) with no superoptimization, heuristic partitioner, and layout synthesizer (***ns+hp+l***), and (e) with no superoptimization, heuristic partitioner, and imprecise layout synthesizer (***ns+hp+il***).

We run 5 benchmarks in this experiment. *Prefixsum* sequentially computes prefixsum of a distributed array that spans 10 cores. *SSD* computes 38-bit sum of squared distance between two distributed 18-bit arrays of size 160, each of which spans across 4 cores. SSDs of different chunks of array can be computed in parallel since there is no dependency between them. *Convolution* performs 1D convolution on a 4-cores distributed array with kernel's width equal to 5 in parallel. The program first fills in the ghost regions to eliminate loop dependency before the main convolution computation starts. *Sqrt* computes 16-bit square root of 32-bit input. *Sin-Cos* computes $cos(x)$ and $sin(x)$.

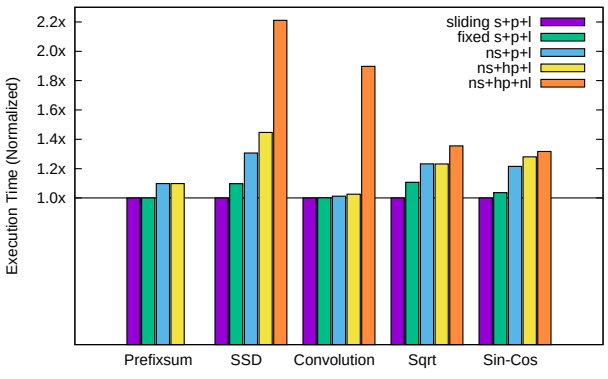


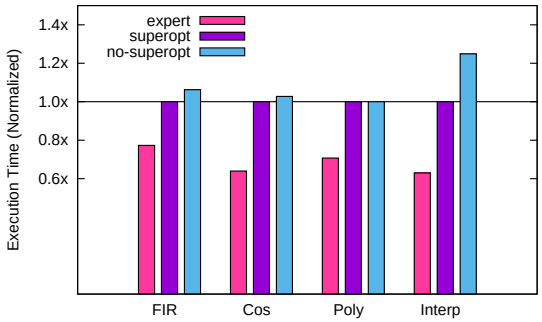
Figure 5. Execution time of multicore benchmarks normalized to program generated by the complete synthesizing compiler.

The execution time result shown in Figure 5 confirms our hypothesis. First, comparing *ns+p+l* (third bar) vs. *ns+hp+l* (forth bar) shows that partitioning synthesizer offers 5% on average and up to 11% speedup over heuristic partitioner. Second, comparing *ns+hp+l* (forth bar) vs. *ns+hp+il* (fifth bar) shows that more precise layout is crucial, providing 1.8x speed up on Convolution. When the layout synthesizer does not take communication frequency into account, it fails to group the heavily communicating cores right next to each other; as a result, the communication paths of different parallel groups share common cores preventing those groups from running in parallel. In Prefixsum, the imprecise layout generates program that is too big. Third, comparing *sliding s+p+l* (first bar) vs. *ns+p+l* (third bar) shows that superoptimization gives 15% on average and up to 30% speedup over programs generated without superoptimization. Finally, comparing *sliding s+p+l* (first bar) vs. *fixed s+p+l* (second bar) shows that program generated with sliding windows superoptimization is on average 4% (up to 11%) faster than programs generated with fixed windows.

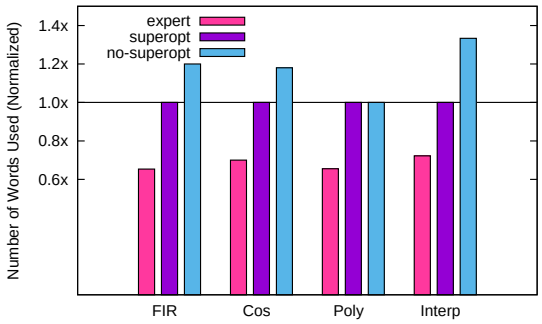
Hypothesis 2 *Partitioning synthesizer is more robust than the heuristic one.*

In this experiment, we test whether the partitioning synthesizer always performs better than the heuristic partitioner. The previous experiment already shows that the partitioning synthesizer generates no slower programs on all 5 benchmarks. In this experiment, we look at the number of cores the programs occupy, on the same set of benchmarks. We find that synthesis is more robust since in 3 out of 5 benchmarks, it generates programs that require significantly fewer cores (using 50-72% of the number of cores used by the heuristic).

Furthermore, the heuristic partitioner does not account for communication code, since counting communication without double counting is very complicated in the heuristic algorithm. Therefore, we set the threshold limit of space of each core by scaling the actual available space with a factor k in the heuristic partitioner. The higher the scaling factor, the fewer cores it uses. However, the maximum feasible k— while generating code that still fits in cores— for different programs varies ($k = 0.8$ on SSD and $k = 0.4$ on Sqrt). Thus, heuristic algorithm requires parameter tuning specific to each program, while synthesis does not.



(a) Execution time



(b) Space

Figure 6. Single-core benchmarks.

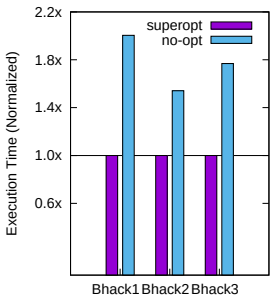
Hypothesis 3 *Generated programs with synthesis are comparable to the highly-optimized expert-written programs.*

We compare execution time and program size between highly-optimized programs written by GA144 developers, generated programs with superoptimization, and generated programs without superoptimization. Single-core expert-written programs that we have access to are *FIR*, applying 16th-order discrete-time finite impulse response filter on a sequence of samples, *Cos*, computing cosine, *Polynomial*, evaluating a polynomial using Horner's method given the coefficients and an input, and *Interp*, performing linear interpolation on input data given a sequence of reference points.

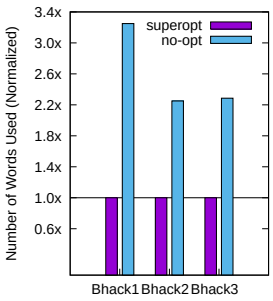
Figure 6 shows that our generated programs are 46% slower and 47% longer than the experts' on average, and the superoptimizer helps improve the running time by 8% and reduce the program length 18% compared to no superoptimization on average.

Besides single-core benchmarks, the only multicore application written by experts that we can compare against is the MD5 hash. The other applications published on the GreenArrays website, including SRAM control cluster, programmable DMA channel, and dynamic message routing, require interaction with GA virtual machine and specific I/O instructions for accessing external memory that Chlorophyll does not support. In this benchmark, we compute the hash value of a random message with one million characters. The sequence of characters is streamed into the computing cores while the hash value is being computed.

Given partition annotations for all arrays and variables, the partitioning synthesizer times out, while the heuristic partitioner fails to produce a program that fits in memory. We manually obtain partition annotations with the assistance of the synthesizer. We first ignore other functions except main by commenting out their bodies. After we solve main, we include other functions back in one by one. Finally, we refine the partitioning by examining the machine code and further breaking or combining partitions just by changing the partition annotations. Therefore, we can generate code for different partitioning (without superoptimization) in a very short amount of time. This process can, in fact, be done automatically. However, we have not yet integrated this technique into our current tool chain.



(a) Execution time



(b) Space

Figure 7. Bithack benchmarks.

We generate two versions of MD5. First, we partition the program such that the generated unsuperoptimized code is slightly bigger than memory, but the excess is small enough that the final superoptimized code still fits. We generate the second version that fits on cores without superoptimization. The generated program with superoptimization is 13% faster than the one without superoptimization, while using 6 fewer cores. Compared to the experts', it is only 77% slower and uses 2x more cores. This result confirms that our generated programs are comparable with the experts' not only on small programs but also on a real application.

Hypothesis 4 *Superoptimizer can discover optimizations that traditional compilers may not.*

We implement a few small programs taken from the book *Hacker's Delight*: *Bithack 1*, $x-(x\&y)$, *Bithack 2*, $\sim(x-y)$, and *Bithack 3*, $(x\oplus y)\oplus(x\&y)$. We hypothesize that the superoptimizer should be able to discover the bit-hack tricks automatically. Figure 7 shows that superoptimization provides 1.8x speedup and 2.6x code length reduction on average. The superoptimizer indeed successfully discovers $x\&\sim y$, $\sim x+y$, and $(x\&\sim y)+y$ as the faster implementations for the three benchmarks respectively. Investigating generated programs in many benchmarks, we find that the superoptimizer can discover various strength reductions and clever ways to manipulate data and return stacks. It also automatically performs register allocation and CSE within program segments, and exploits special instructions that do not exist in common ISAs. Hence, the superoptimizer can discover unlimited number of optimizations specific to the machine, while the optimizing compiler can only perform limited number of optimizations implemented by the compiler developers.

Hypothesis 5 *Chlorophyll increases programmer productivity and offers the ability to explore different implementations quickly to obtain one with satisfying performance.*

A graduate student spent one summer testing the performance of GA144 and TI MSP430 micro-controller. He managed to learn the GA assembly-like stack-based programming language. However, he was able implement only 2 benchmarks: FIR and a simple pedometer application [2]. In contrast, with our compiler, we can implement 5 different FIR implementations within an afternoon. Figure 8 shows the running time of 3 different implementations of FIR: sequential FIR-1, parallel FIR-2 on 2 cores, and parallel FIR-4 on 4 cores, as well as the experts'. Parallel FIR-4 is 1.8x faster than the experts' with the cost of more cores. Hence, programmers can use our tool to productively test different implementations and exploit parallelism to get the fastest implementation. Although superoptimization is slower, we can still test our implementations quickly by not running the superoptimization to get a rough estimate of performance.

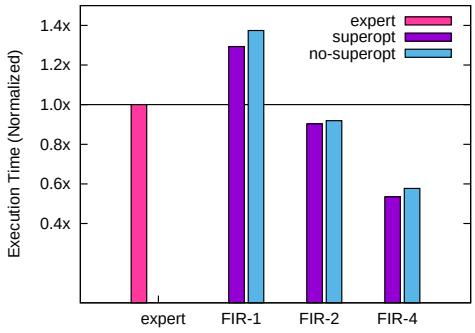


Figure 8. FIR benchmark.

Benchmarks	# of Given Cores	Loop Fiss	Part	Layout	Superopt
Prefixsum	64	3	36	24	10.78
SSD	64	12	225	24	4.46
Convolution	64	23	122	24	8.39
Sqrt	16	0	566	7	3.60
Sin-Cos	16	2	527	7	6.31
MD5	64	7	N/A	24	10.43

Figure 10. Compile time of multicore benchmarks. Time is in seconds except superoptimize time is in hours. The compiler runs on 8 cores machine, so it superoptimizes up to 8 independent instances in parallel. Layout time only depends on the number of available cores given. Heuristic partitioner takes less than one second to generate a solution.

Hypothesis 6 *The compiler can be improved by providing more human insight to the synthesizers.*

The GA instruction set does not include division, but expert-written integer division code is provided in ROM, so programmers can conveniently call that function. A faster division can be implemented when a divisor is known. More specifically, $x/k = (k_1 * x) >> k_2$ where k_1 and k_2 are magic numbers depending on k. We modify the superoptimizer so that it understands division and accepts the division instruction in an input spec. Then, we provide this template to the superoptimizer to fill in the numbers for a specific divisor similar to Sketch [31]. Given the template, the compiler can produce a program that is 6x faster and 3x shorter than the experts' general integer division program within 3 seconds. In theory, the superoptimizer can discover the entire program without the sketch, but it could take much longer since this program is 33 instructions long.

Thus, adding more templates improves performance of generated programs and scalability of the synthesizers. This is similar to implementing optimizations for traditional compilers. Synthesis is in general more powerful because it does not rely on any lookup table and discovers faster code by searching.

Figure 9 and 10 show the compile time for single-core benchmarks and multicore benchmarks used in our experiments respectively. Note that our superoptimizer is slower than stochastic superoptimization [29], since the stochastic one runs on a cluster of machines, while ours runs on a single machine. Partitioning is also slow, but such algorithms are generally slow, such as partitioning for FPGA [34]. We address the issue by allowing the programmer to accelerate the partitioning process by pinning data or code to cores that they have insight about.

8. Related Work

A number of compilers have been developed for partitioning data and computations and mapping them to physical spatial cores in various ways for different purposes. StreamIt decomposes the compilation problem and applies SA to solve the layout problem similar to ours [12]. However, partitions are mostly defined by programmers using *filters*. However, Raw processors have much larger local memory than GA144, and StreamIt is a DSL for streaming applications, so it is hard to implement applications such as MD5 on very tiny core processors. The Vivado Design Suite performs High-Level Synthesis (HLS) that transforms a C, C++ or SystemC design spec into a RTL implementation that in turn can be synthesized onto a FPGA [34]. The programmer can specify additional constraints using directives, such as controlling the binding process of operations to cores, etc. This is much more limited compared to our programming model. For example, operations such as multiplications are implemented by a specific hardware multiplier in the RTL design using a specific core.

Another approach to solve the placement and routing problem uses ILP [27] by mapping the DAG (computation) to the graph (structure of hardware). The constraints represent placement of computation, data routing, managing event timing and resource utilization, and optimization for the hardware-specific objective function. Even though this approach is retargetable to multiple hardwares, Chlorophyll is more flexible because it allows the programmer to easily specify fine-grained placement and routing, while also synthesizing for other extreme hardware restrictions.

GA144 shares many similarities with systolic arrays. However, Systolic arrays are mostly used for applications with rhythmic communications because the arrays are designed to run massively parallel programs [18]. The higher level languages for systolic arrays are domain-specific for such applications [17, 20]. In contrast, GA144 is developed for energy efficiency, and its target applications are not exclusively to rhythmic communication applications.

The high performance computing community has been developing programming models to support programming on distributed memory. Our code separation technique is similar to compiling High Performance Fortran (HPF) for distributed memory computers. HPF generates a guard for every array access, checking if a processor owns that entry of the array with some optimizations. Instead, we can generate code without these guards by performing loop fission and statically determining the partitions for every variable and operation during compile time. The partitioning problem also comes up. Many Distributed Fortran compilers simply apply "owner computes" rule according to the output data decomposition [5, 25]. This partitioning technique does not suit our case since the fixed placement of operations according to the data distribution might result in partitions that are too big.

Our memory model is PGAS, similar to that of many languages. They offers programmers more control over mapping operators to processors [6, 26, 28, 35]. Data repartitioning is fairly easy, but operation repartitioning is not. Applying type inference to determine properties of interest has been done in distributed memory programming. X10 introduces *place type* and exploits type inference to eliminate dynamic references of global pointers [7]. Titanium, similarly, uses type inference to minimize the number of global pointers in the program [22].

The original superoptimizer by Massalin finds the shortest program by enumerating every possible program [24]. Each candidate program is checked on manually supplied test cases. A more recent take on superoptimization is Denali [19], using goal-directed search, allowing it to scale better. Like our system, the search is performed by an automated theorem prover. *Stochastic* superoptimization [29] introduces a different search technique: a Markov Chain Monte Carlo (MCMC) sampler, maximizing a function of correctness and performance. This approach scales to longer programs over much larger instruction sets like x86. However, it did not translate well from a register-based system to the stack-based GA144. Superoptimization has also been used to generate peephole optimization rules rather than actual programs [3]. This is another approach to supporting novel architectures like GA144 without having to manually write an optimizing compiler. This would not work well for translating to GA144 because going from a register-based to stack-based system requires many non-local transformations.

Benchmarks	FIR	Cos	Polynomial	Interp	Bithack 1	Bithack 2	Bithack 3
Program Length (words)	90	59	29	48	13	9	16
Superoptimize Time (hrs)	3.23	2.35	1.42	10.01	0.37	4.92	25.08*

Figure 9. Superoptimize time and program length of single-core benchmarks. A word in program sequences contains either 4 instructions or a constant literal. *Bithack-3 takes 25.08 hours when program segment length is capped at 30 instructions in order to generate the most optimal solution. With the default length (16 instructions), it takes 2.5 hours.

Another variation on superoptimization is *component-based synthesis* [16]. It synthesizes a circuit-style (loop-free) composition from a limited collection of instructions. This constrains the number of times any given instruction can be used, shrinking the search space. This approach does not map well to GA144 because coming up with a set of components from the high-level input is difficult, especially since the stack-manipulation instructions needed for an efficient program are not easily predictable.

9. Conclusion

Building efficient optimizing compilers is difficult, even for traditional architectures that are designed for programmability. With radically stripped down and evolving target architectures such as GA144, however, the traditional compilation approach becomes more difficult and less practical to implement. We have built the first synthesis-aided compiler for extremely minimalist architectures. Our compiler decomposes the compilation problem into smaller subproblems which can be solved by various synthesizers and easy-to-implement transformations. We introduce a new spatial programming model for fine-grain partitioning aiming to provide programmability on top of non programmer-friendly hardware.

The synthesis-aided compilation technique we introduce allows programmers to implement programs without reasoning about low-level details. It also enables compiler devleopers to quickly develop new high-performance compilers for radical architectures without knowing how to implement optimizations specific to an architecture. Although program synthesis may not scale on a large problem on its own, our work shows that we can overcome this problem by decomposing the problem into smaller ones and providing more human insight.

References

[1] Arduino. Arduino playground: Avr code. `http://playground.arduino.cc/Main/AVR`.

[2] Rimas Avizienis and Per Ljung. Comparing the Energy Efficiency and Performance of the Texas Instrument MSP430 and the GreenArrays GA144 processors. Technical report, 2012.

[3] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *ASPLOS*, 2006.

[4] Sorav Bansal and Alex Aiken. Binary translation using peephole superoptimizers. In *OSDI*, 2008.

[5] Zeki Bozkus, Alok Choudhary, Tomasz Haupt, Geoffrey Fox, and Sanjay Ranka. Compiling hpf for distributed memory mimd computers. In *The Interaction of Compilation Technology and Computer Architecture*. 1994.

[6] William W. Carlson, Jesse M. Draper, and David E. Culler. S-246, 187 introduction to upc and language specification.

[7] Satish Chandra, Vijay Saraswat, Vivek Sarkar, and Rastislav Bodik. Type inference for locality analysis of distributed data structures. In *PPoPP*, 2008.

[8] David T. Connolly. An improved annealing scheme for the QAP. *European Journal of Operational Research*, 46(1):93 – 100, 1990.

[9] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, 2008.

[10] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. Bradford Company, Scituate, MA, USA, 2004.

[11] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. Burg: fast optimal instruction selection and tree parsing. *SIGPLAN Not.*, 27(4):68–76, April 1992.

[12] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *ASPLOS*, 2002.

[13] GreenArrays. *Product Brief: GreenArrays Architecture*, 2010.

[14] GreenArrays. *Product Brief: GreenArrays GA144*, 2010.

[15] GreenArrays. *Appplication Note AB001: An Implementation of the MD5 Hash*, 2012.

[16] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.

[17] Richard Hughey. Programming systolic arrays. Technical report, Brown University, 1992.

[18] Kurtis T. Johnson, A. R. Hurson, and Behrooz Shirazi. General-purpose systolic arrays. *Computer*, 26(11):20–31, November 1993.

[19] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: a goal-directed superoptimizer. In *PLDI*, 2002.

[20] Monica S. Lam. *A Systolic Array Optimizing Compiler*. Kluwer Academic Publishers, Norwell, MA, USA, 1989.

[21] Eugene L. Lawler. The quadratic assignment problem. *Manage. Sci.*, 9:586–599, 1963.

[22] Ben Liblit, Alex Aiken, and Katherine Yelick. Type systems for distributed data sharing. In Radhia Cousot, editor, *Static Analysis*, volume 2694 of *Lecture Notes in Computer Science*, pages 273–294. Springer Berlin Heidelberg, 2003.

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

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

[25] John Merlin. Techniques for the automatic parallelisation of 'distributed fortran 90', 1992.

[26] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edoardo Aprà. Advances, applications and performance of the global arrays shared memory programming toolkit. *Int. J. High Perform. Comput. Appl.*, 20(2):203–231, May 2006.

[27] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. A general constraint-centric scheduling framework for spatial architectures. In *PLDI*, 2013.

[28] V. Sarawat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 language specification, October 2012.

[29] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ASPLOS*, 2013.

[30] Jadranka Skorin-Kapov. Tabu search applied to the quadratic assignment problem. *INFORMS Journal on Computing*, 2(1):33–45, 1990.

[31] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.

[32] Trefis Team. Can Intel Challenge ARM's Mobile Dominance?, 2012.

[33] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with Rosette. In *Symp. on New Ideas in Programming and Reflections on Software (Onward!)*, 2013.

[34] Xilinx. Vivado design suite. `http://www.xilinx.com/products/design-tools/vivado/`.

[35] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance java dialect. In *In ACM*, pages 10–11, 1998.