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

Phitchaya Phothilimthana

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-121
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-121.html

May 15, 2015

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

by Phitchaya Mangpo Phothilimthana

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor R. Bodik
Research Advisor

(Date)

* * * * * * *

Professor K. Yelick
Second Reader

(Date)

Abstract

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

by

Phitchaya Mangpo Phothilimthana

Master of Science in Computer Science

University of California, Berkeley

Professor Rastislav Bodik, Co-chair

Professor Katherine Yelick, Co-chair

We developed *Chlorophyll*, a synthesis-aided programming model and compiler for the GreenArrays GA144, an extremely minimalist low-power spatial architecture that requires partitioning a program into fragments of no more than 256 instructions and 64 words of data. This processor is approximately 100-times more energy efficient than other commercially available processors, but currently it can only be programmed using a low-level stack-based language.

The Chlorophyll programming model allows programmers to provide their insight on program partitioning by specifying partial partitioning of data and computation. The Chlorophyll 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 the compilation into smaller synthesis subproblems: partitioning, layout, and code generation. We show that the synthesized programs are no more than 19% slower than highly optimized expert-written programs on the MD5 benchmark and are faster than programs produced by a heuristic, non-synthesizing version of our compiler.

Chapter 1

Introduction

1.1 Motivation

Energy requirements have been dictating simpler processor implementations with more energy dedicated to computation and less to processor control. Simplicity is already the norm in low-power systems, where 32-bit ARM dominates the phone computer class [40]; the 16-bit TI 430MSP is a typical example of a low-power embedded controller; the even simpler 8-bit Atmel AVR controller powers Arduino [2].

The GreenArrays GA144 is a recent example of a low-power minimalistic spatial processor[1], composed of many small, simple, identical cores [17]. Likely the most energy-efficient commercially available processor, it consumes 9-times less energy and runs 11-times faster than the TI MSP430 low-power microcontroller on a finite impulse response benchmark [3]. Naturally, energy efficiency comes at the cost of low programmability; among the many challenges of programming the GA144, programs must be meticulously partitioned and laid out onto the physical cores.

We imagine that future low-power processors will likely be similar to the GA144. First, they will likely be spatial with simple interconnects between resources or cores. Second, they will likely have radically different Instruction Set Architectures (ISAs) from what we commonly use today. Third, they will likely be minimalistic, providing little programmability support and therefore placing a greater burden on programmers and compilers.

In this thesis, we introduce a new programming model and a synthesis-based compiler for such spatial processors. Our primary hardware target is the GA144 which takes these design features—spatiality, idiosyncrasy of ISA, and minimalism—to extremes, maximizing the demands on our programming tool chain; if we can build a synthesizer for this processor, we should be able to build ones for other low-power processors as well.

[1] A *spatial architecture* is an architecture for which the user or the compiler must assign data, computations, and communication primitives explicitly to its specific hardware resources such as computing units, storage, and an interconnect network.

GreenArrays Low-Power Spatial Processor

The GA144 is a stack-based 18-bit processor consisting of 144 cores with no clock or shared memory [17, 18]. It consumes less energy per instruction than any other commercially available architectures [28]. A small number of GA144 applications have been developed directly in arrayForth, a low-level stack-based language, but using this low-level language presents many difficulties.

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 with the computation code of a core, carefully avoiding deadlocks and race conditions.

Each core contains only a tiny amount of memory and 2 small circular stacks (one for data and one for return addresses), which together offer fewer than 100 18-bit words of storage per core. This forces programs and data structures to be partitioned over multiple cores. For instance, even a heavily optimized MD5 hash implementation has to be partitioned across 10 cores on the GA144 [19].

Since the GA144 is an 18-bit architecture, wider words must be implemented in software. Additionally, the machine code is stack-based, so it is relatively foreign to most programmers who have only ever used register-based systems.

Our system tries to help the programmer overcome these difficulties, presenting a more familiar, higher-level abstraction and automatically handling some of the challenges described above.

Challenges and Solutions

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

First, classical compilers that transform code using heuristic-guided tree rewrites may not be able to bridge the abstraction gap of low-power programming. When optimizing the architecture for energy efficiency sacrifices programmability features in the hardware (such as hardware-controlled caches) the abstraction gap grows larger. This growing gap 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[33]; and (ii) low-power architectures will be actively investigated for a while, presenting a moving target and delaying compiler development.

Our solution uses syntax-guided synthesis [39, 1]—we sketch the desired program and let the synthesizer search for an implementation that meets the specification. Program synthesis is a form of automatic programming using formal verification. Rather than writing a program directly, the user provides a goal (the specification) and the synthesizer automatically generates the program.

Second, programmers prefer to control hardware at a higher level of abstraction. For example, to optimize their programs, programmers prefer to manually partition data struc-

tures and code but not to deal with the low-level details of the resulting communication code. Our programming model allows programmers to selectively partition key data structures and code, leaving the remaining partitioning and 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, but not on decompositions of larger programs [39, 20, 35]. In order to scale synthesis to large programs, we decompose a large problem into smaller ones. Our approach has three synthesis subproblems: program partitioning, layout and routing, and optimized program generation (as well as code separation, a classical compilation problem). The resulting synthesis-aided compiler uses a suitable solver for each subproblem.

In summary, we make the following contributions:

- We developed a programming model that allows the programmer to optionally partition data structures and code. Our model facilitates fine-grained partitioning over the spatial architecture.
- We designed and evaluated a compiler that solves three consecutive synthesis subproblems. Our design shows how to decompose synthesis to scale to large, practical problems.
- We introduced a low-effort approach to building compilers for unusual architectures without sacrificing much performance.
- We wrote the first high-level compiler for the minimalistic GA144 architecture. Its generated code performs within a factor of 1.65 of hand-written code. The only alternative for running high-level programs is an interpreter that runs orders of magnitudes slower, negating the architecture's energy benefits.

1.2 Overview

Chlorophyll decomposes the problem of compiling a high-level program to spatial machine code into four main subproblems: partitioning, layout and routing, code separation, and code generation. These subproblems are difficult for traditional compilers. In this paper, we show how these problems can be solved naturally using synthesis techniques.

Step 1 (partition) The input to this step is a source program with partition annotations which specify the logical core (partition) where code and data reside. The annotations allow the programmer to provide insight about the partitioning or experiment with different partitioning just by changing the annotations. An input program does not have to be fully annotated. For example, in this program

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

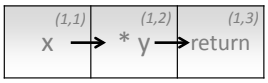
we specify that the result will be delivered at partition 0 but do not specify the partitions of variable x, y, and operation $+$.

The compiler then infers (i.e. synthesizes) the rest of the partition annotations such that each program fragment (per-core program) fits into a core, minimizing a static over-approximation of the amount of messages between partitions. Here is one possible mapping (for a very tiny core):

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

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. ! is the send operation. 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.

Step 2 (layout) The layout synthesizer maps program fragments onto physical cores, minimizing a refined approximation of communication costs. It also determines a communication path (routing) between each pair of cores. We map this synthesis problem to an instance of the well-known Quadratic Assignment Problem (QAP) which can be solved exactly or approximately [25, 11, 14, 37]. We chose to use the Simulated Annealing algorithm as it is one of the fastest techniques and produces a nearly optimal solution [11]. Given the partitioned *mult* function from the previous step, the figure below shows the result of this step.



Step 3 (code separation) The separator splits the fully partitioned program into per-core program fragments, inserting sends and receives for communication. This step uses a classical program transformation. We guarantee that the resulting separated programs are deadlock-free by disallowing instruction reordering within each core. Our running example results in these program fragments:

```
// 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 first naïvely compiles each program fragment into machine code. The code is then optimized with a superoptimizing synthesizer, which searches the space of possible instruction sequences to find ones that are correct and fastest [29]. Although the superoptimizer is allowed to reorder evaluations, it preserves the

order of sends and receives which is sufficient to prevent deadlock. We apply a sliding window technique to the synthesizer to adaptively merge small code sequences into bigger ones and input it back into the synthesizer. The synthesizer persistently caches synthesized code to avoided unnecessary recomputation.

The rest of this thesis is organized as follows. The next chapter describes the programming model and the compiler—which decomposes the compilation problem into these four steps—that we have developed. Chapter 3 presents evaluation of the compiler. Chapter 4 describes related work. Chapter 5 states our conclusion.

Chapter 2

Programming Model and Compiler

In this chaper, we introduce our programming model for partitioning and describe how we built our compiler in fours steps—partition, layout, code separation, and code generation—using program synthesis.

2.1 Programming Model for Partitioning

The Chlorophyll language is designed to simplify reasoning about partitioning and to obviate the need for explicit communication code. We achieve these goals by extending a simple type system with a *partition type* and optimally inferring unspecified partitions with our *partitioning synthesizer*. In this section, we introduce the Chlorophyll language, its type system, and the partitioning process.

Language Overview

Chlorophyll syntax is a subset of C with *partition annotation* specifying the partitions of data and operations. In order to make fine-grained partitioning possible, we track the partition of every piece of data and operation. Figure 2.1(a) shows the `LeftRotate` program implemented in Chlorophyll. On line 18, we set the partition of variable r to be 6 by annotating its declaration. On line 12, we assign the partitions of distributed array x such that for $0 \leq i < 32$, $x[i]$ lives in partition 0, and the rest in partition 1. On line 21, operation + is assigned to partition 6. On line 20, operation − is assigned to $place(z[i])$; when $0 \leq i < 32$, operation − at partition 4 is executed, and when $32 \leq i < 64$, operation − at partition 5 is executed. Note that most of the data and operations in the program are left unannotated—their partitions will be automatically inferred by the partitioning synthesizer.

Programming Constructs and Space

Constants, variables, arrays, operators, and statements all occupy space in memory. Most programming constructs, such as variable declarations, variable accesses, variable assign-

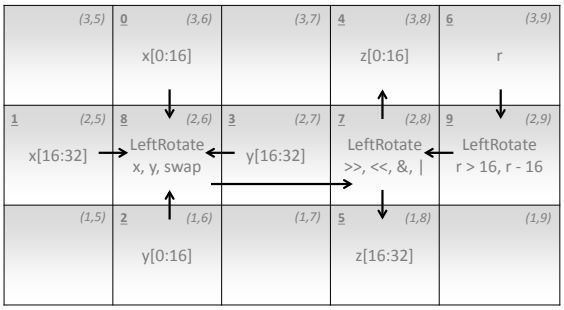
```
1  int leftrotate(int x, int y, int r) {
2    if(r > 16) {
3      int swap = x;
4      x = y;
5      y = swap;
6      r = r - 16;
7    }
8    return ((y >> (16 - r)) | (x << r)) & 65535;
9  }
10
11 void main() {
12   int@{[0:32]=0,[32:64]=1} x[64];
13   int@{[0:32]=2,[32:64]=3} y[64];
14   int@{[0:32]=4,[32:64]=5} 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@9 r) {
  if(r >@9 16) {
    int@8 swap = x;
    x = y;
    y = swap;
    r = r -@9 16;
  }
  return ((y!7 >>@7 (16 -@7 r!7)) |@7 (x!7 <<@7 r!7)) &@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!9)!4 -@4 1; //!8 is 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!9)!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. The numbers at the top-left corner of the boxes represent partition IDs corresponding to the partition annotations in the source code.

```
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 2.1: Example program written in Chlorophyll, and intermediate results from partitioning, layout, and code separation steps

ments and binary operations, occupy a constant amount of memory, so we can estimate the space occupied by the program with a simple lookup table. However, we have to handle control flow constructs and arrays with more care as they may require more complex communication between the involved partitions.

Control Flow Constructs (*for, while,* and *if-else*) When the body of a control flow construct is spread across many partitions, called *body partitions*, the actual control flow logic needs to be placed in each of these partitions as well. For example, the result of the condition expression x *+@2* y, is at partition 2. This evaluated value is sent to all the body partitions, each of which in turn uses the received value as its condition.

Chlorophyll only supports *for* loops of the form *for (i from e1 to e2) {...}*, where $e1$ and $e2$ are constants. The iterator i starts from $e1$ and is incremented by 1. The condition of the loop is $i < e2$. These restrictions allow Chlorophyll to produce more efficient code. Specifically, each of the body partitions uses its own copy of i. This reduces the amount of communication between partitions. Other sorts of iteration are supported by *while* loops which have no restrictions on conditions.

Arrays There are two kinds of arrays in Chlorophyll :

- ***Non-distributed arrays*** only live in one partition. An index into this type of array has to live at the same partition as the array itself.
- ***Distributed arrays*** live in multiple partitions. Arrays x, y and z from `LeftRotate` are examples. This type of array can only be indexed by affine expressions of surrounding loop variables and constants. Accessing this type of array requires no communication because the indexes are comprised of loop variables, which live in every body partition. Chlorophyll currently does not support other kinds of indexing into distributed arrays.

Partition Annotations Programmers specify partitions of data and operators using partition annotations. Partition annotations (A) can be expressed as follows:

$A :=$ N | `place(`var`)` | `place(`$array$`)`
$N :=$ `natural number` $var :=$ `variable` $array :=$ `array access`

$place(x)$ refers to the partition where variable x lives, and $place(y[i])$ refers to the partition where the ith entry of array y lives. $place(y[i])$ can only be used inside the body of a *for* loop with iterator i.

Current Limitations

Chlorophyll does not handle recursive calls, multidimensional arrays, or array accesses with index expressions that use non-loop variables. Unbounded loops can be implemented using *while*; however, the *for* loop is currently restricted to the form described earlier.

Partition Type and Typing Rules

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

$$\begin{array}{rlrl} \tau := & \tau@\rho & \tau := & \texttt{val} \mid \texttt{int} \mid \texttt{void} \\ \rho := & N \mid \texttt{any} \mid \rho_{dist} & \rho_{dist} := & \{(N,)^+\} \\ N := & \texttt{natural number} & & \end{array}$$

Our types consist of data types τ and partition types ρ. For simplicity, the data types only include `int`, `val`, and `void`. ρ_{dist} is a type of distributed array.

The typing rules shown in Figure 2.2 (omitting some trivial rules) enforce that operands and operators are in the same partition. Constants and loop variables have partition type `any` indicating that they can be at any partition. The *partition subtype* rule allows an expression with partition type `any` to be used everywhere. In the *access dist-array* rule, the type checker needs to evaluate e at compile time. This is possible because our type system ensures that the index to a distributed array is only comprised of loop variables and constants, and the language enforces finite loop bounds. Thus, the compiler can break a loop that iterates over a distributed array into multiple loops, each of which accesses a chunk of the array that lives on a particular partition. For example, the loop in `LeftRotate` is broken into two loops: one iterating from 0 to 31, and another from 32 to 63.

! is an operation for sending data from one partition to another. It will be translated to both a write operation at the sending partition and a read operation at the receiving partition. It is the only operation that accepts an operand whose partition type may not be a subtype of the output's 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.

Partitioning Process

Partitioning a program can be thought of as a type inference on partition types. The partitioning synthesizer is constructed from 1) the *communication interpreter*, which counts the number of communications needed and 2) the *partition space check*, which ensures code and data fit in the memory of the appropriate core.

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

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

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

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

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

$$\frac{\Gamma \vdash f : \tau_1@\rho_1 \to \tau_2@\rho_2 \quad \Gamma \vdash e : \tau_3@\rho_3 \quad \tau_3@\rho_3 < \tau_1@\rho_1}{\Gamma \vdash f\ e : \tau_2@\rho_2}\ \text{[function call]}$$

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

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

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

Figure 2.2: Typing rules

Communications Interpreter

Let $Comm(P, \sigma, x)$ be a function that counts the number of communications in a given program P with complete annotated partitions σ and a concrete input x. The communication count is calculated with $MaxComm(P, \sigma) = \max_{x \in Input} Comm(P, \sigma, x)$, where $Input$ is a set of all valid inputs to the program, assuming *while* loops are executed a certain number of times (currently 100). $MaxComm$ computes the maximum number of communications by considering all possible program paths. For most constructs, the communication count is equal to sum of its components' counts. ! increments the communication count by 1. Loops multiply the count. Conditional statments add the communication count by the number of the body partitions (subtracted by 1 if one of the body partitions is the same as the partition of the result of the condition expression).

Partition Space Check

Operations, statements, and communication operations (i.e. read and write) occupy the memory of the partitions they belong to. Constants occupy memory of the partitions in which they are inferred to be (usually the partitions of their operators or the left-hand-side variables they are assigned to). If-elses, loops, and loop variables occupy memory in all of their body partitions. Given a program with complete partition annotations, the partition space checker computes 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.

Partitioning Synthesizer

We implemented the communication count interpreter and the partition space checker using Rosette, a language for building light-weight synthesizers [41, 42]. We represented 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 communication count interpreter compute a concrete number of communications, and the partition space checker simply verifies that the memory constraint holds. Given a partially annotated or unannotated program (a program with some or all symbolic partitions), the result from the communication count interpretation is a formula in terms of the symbolic variables, and the partition space check becomes a constraint on the symbolic variables.

Once we obtain a formula from the communication count and the 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 a solution, we attempt to reduce the communication count further by asking the solver the same query with an additional constraint setting an upper bound on the count. We lower the upper bound until no solution can be found.

Pre-Partitioning Process

Before the partition process takes place, *loop splitting* is performed. Since the traditional approach to loop splitting is difficult to implement, we used Rosette to implement a loop splitting synthesizer similar to the way we implemented 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. The first loop iterates over i from a_0 to b_0, the second loop from a_1 to b_1, and so on. We then check that $a_0 = 1$ and $b_2 = 10$. For every iteration l such that $0 \leq l < k - 1$, we check that $a_{l+1} = b_l$. For every iteration l such that $0 \leq l < k$,

where $a_l \leq i < b_l$, we check that $x[i]$ belongs to only one partition, as well as $x[i-1]$. We implemented the checker as if the bounds are concrete. When the bounds are unknown, they become symbolic values, and the checking conditions are used as constraints. 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.

Example and Rationale

Figure 2.1(b) shows the result after partitioning the program in Figure 2.1(a) with 64 words of memory per core. Notice that ! operations are automatically inserted into the program. If the programmer writes partition annotations such that it is impossible to partition the program into program fragments that fit on cores, this will result in a compile-time error.

We support manual annotations based on the philosophy that the programmer and compiler generally have different strengths and that we should let the programmer provide high-level insights to help the compiler. This makes our synthesizer more scalable.

2.2 Layout

In this step, we assign program fragments to physical cores by solving an instance of QAP, stated as follows:

Given a set of facilities F, a set of locations L, a flow 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:

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

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

This QAP instance can be solved with techniques ranging from Branch and Bound search with pruning [25], to Simulated Annealing (SA) [11], to Ant System [14], to Tabu Search [37].

According to our preliminary experiments, SA takes the least amount of time and generates the best (often optimal) solutions.[1]

We used an existing SA implementation for the layout synthesizer in our compiler. The compiler generates a flow graph f by adding flow units for every `!` operator and conditional statement, and the graph is given to the SA program. The result maps program fragments to physical cores. We use this result to generate a communication path, which is the shortest path between every two program fragments. The layout and routing of the program in Figure 2.1(b) is shown in Figure 2.1(c).

2.3 Code Separation

The program is separated into multiple program fragments communicating through read and write operations. We chose this particular scheme because GA does not support shared memory; cores can only communicate with neighbors using synchronous channels. We preserve the order of operations within each program fragment with respect to their order in the original program to prevent deadlock. The rest of this section describes this process for each language construct.

Basic Statements A program without control flow, functions, or arrays is simple to separate. We traverse the program AST in post-order fashion, assign sub-expressions to the appropriate program fragments according to their partition types, and add communication code preserving the original order. For example, consider

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

Assume partitions 1, 2, and 3 map to cores (0,1), (0,2), and (0,3) arranged from west to east. The result after 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` are the east and west ports. Note the implicit parallelism in this program: $1 + 2$ and $3 + 4$ are executed in parallel.

Functions A function call in the original program corresponds to one or more function calls, each of which is at one of the cores where the function resides. For instance, in this program

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

[1] On 8×18 grid locations and a random flow graph of 144 facilities, SA took 52 seconds, Ant took 157 seconds, Tabu took 1163 seconds, and Branch and Bound timeout. SA returned the best solution compared to Ant and Tabu.

`f` is split across partitions 1, 2, and 3 with the same layout as the previous example. Invoking `f` requires the invocations at all three partitions:

```
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();
```

Arrays Distributed arrays are stored in multiple cores and are the main sources of parallelism in our 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 updates the distinct parts of the array in parallel.

Figure 2.1(d) shows the `LeftRotate` program at core (2,6) given the layout and routing shown in Figure 2.1(c).

2.4 Code Generation Using Modular Superoptimization

This section explains our machine code generation process given single-core programs as inputs, and describes our optimization via a *modular superoptimization algorithm*.

Typically, generation of optimized machine code is carried out using an algorithm that selects instruction sequences and performs local optimization along the way [15]. This type of algorithm is well-suited for applications in which the optimizations are known, and we can determine all of the valid ways to generate code. However, this rewrite-based approach is not easily adapted to our target machine. For example, it is unclear how to design rules sufficient to take advantage of common non-local optimizations using hardware features like the bounded, circular stacks.

We sidestep the problem of rule creation by searching for an optimized program in the space of candidate programs. One such approach is called superoptimization [29, 23, 20, 35]. A superoptimizer searches the space of all instruction sequences and verifies these candidate

programs behaviorally against a reference implementation. If an optimized program exists in the candidate space, this approach will find it.

Thus, superoptimization leads to an attractive procedure for generating optimal code for unusual hardware: (1) generate naïve code to use as a specification and then (2) synthesize optimal code that matches the specification. Unfortunately, superoptimizers scale to sequences of only about 25 instructions [23, 20, 35], which can be smaller than basic blocks in programs, which may contain up to 100 instructions.

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

- An obvious way to scale superoptimization is to break down large code sequences (specifications) into smaller ones, superoptimize the small segments, and then compose the optimal segments. However, choosing segment boundaries arbitrarily can cause this approach to miss possible optimizations.
- A straightforward method for specifying the input-output behavior of the program segments prevents some hardware-specific optimizations. For example, the method may reject a segment that leaves garbage values on the stack even when it is acceptable to do so.

Therefore, we propose our code generation strategy, as summarized in Figure 2.3. The compiler first produces code without optimizations using a naïve code generator, and employs a superoptimizer to generate optimized code. In the next subsection, we explain the naïve code generator and the terminology used in the rest of this section. We then explain

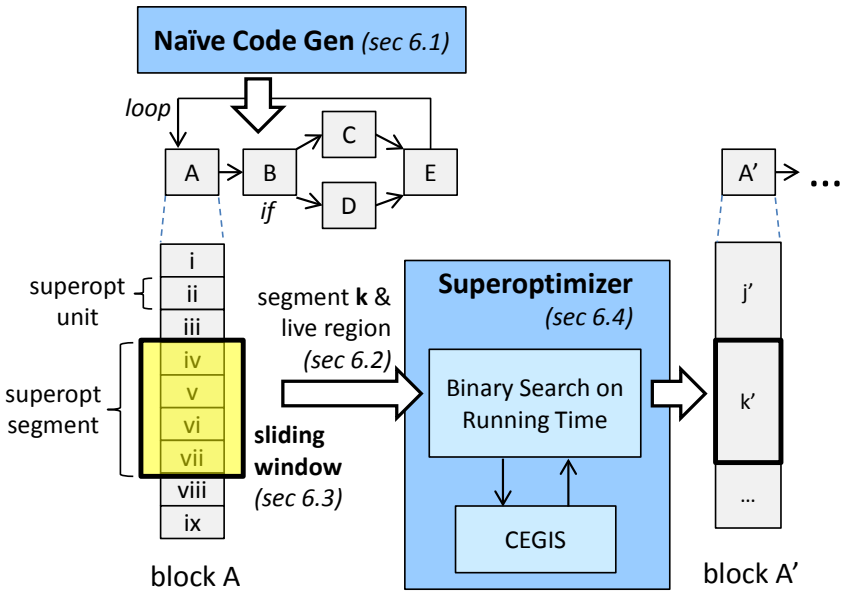


Figure 2.3: Overview of the modular superoptimizer

solutions to the above two problems in the following two subsections. Finally, we describe our superoptimizer for program segments and our approach to encoding the space of candidates as a set of constraints.

Naïve Code Generation and Terminology

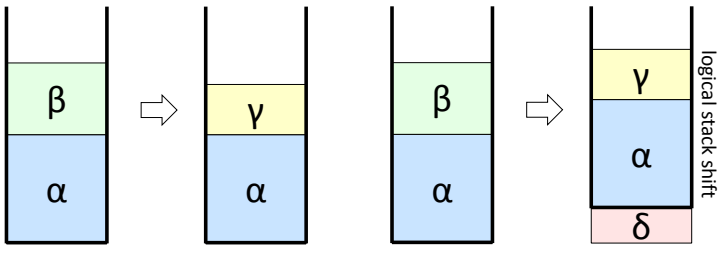
The naïve code generator translates each per-core high-level program into machine code that preserves the program's control flow. The straight-line portions of machine code are stored in many small units called *superoptimizable unit*s. A superoptimizable unit corresponds to one operation in the high-level program and thus contains a few instructions. Contiguous superoptimizable units can be merged into a longer sequence called a *superoptimizable segment*.

We define a state of the machine as a collection of data stack, return stack, memory, and special registers. Each superoptimizable unit contains not only a sequence of instructions but also a *live region* that indicates which parts of the machine's state store live variables at the end of executing the sequence of instructions. The live region of a superoptimizable segment is simply the live region of the last superoptimizable unit. Currently, a live region always contains the entire memory and usually contains some parts of the return stack and data stack, and some of the registers.

Sequences of instructions P and P' change the state of the machine from S to T and T' respectively. Given a live region L, we define $P \overset{L}{\equiv} P'$ if $Extract(T, L) \equiv Extract(T', L)$, where $Extract$ extracts values that reside in the given live region. 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 programs that overflow the data stack or return stack at any point.

Specifications for Modular Superoptimization

We specify the behavior of a segment using a sequence of instructions P and its live region L. In this section, we will focus on the constraints on the data stack since it is used for



(a) Basic (b) Improved

Figure 2.4: Specification on data stack

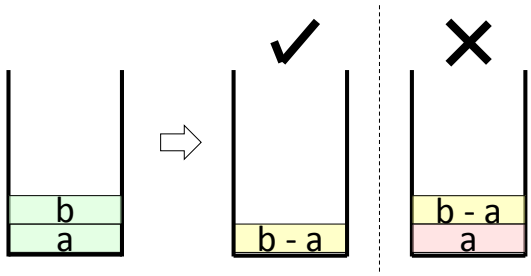


Figure 2.5: Basic specification rejects an instruction sequence that leaves a at the bottom of the stack.

performing every kind of computation and may be used for storing data.

Assume an instruction sequence P changes the data stack from $\alpha|\beta$ to $\alpha|\gamma$ as shown in Figure 2.4(a), and $\alpha|\gamma$ is in the live region. α is a part of the stack that contains intermediate values that will be used later. β is the part of the stack that needs to be removed, and γ is the part of the stack that needs to be added. P' is equivalent to P if P' produces $\alpha|\gamma$, and the stack pointers after executing P and P' are pointing to the same location.

However, this specification is too strict, preventing some optimizations. For instance, consider the example in Figure 2.5 when α is empty, and we want $b - a$ on top of the stack. The shortest sequence of instruction that has this behavior is eight instructions long, with the three final instructions dedicated to removing a remaining garbage value (a in this case) from the stack. It is, in fact, legal to leave a at the bottom of the stack, saving space by eliminating the three 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, as shown in Figure 2.4(b), such that P' is equivalent to P if it produces $\delta|\alpha|\gamma$ without any constraint on the stack pointer, where δ can be empty. Since GA stacks are circular, leaving garbage items at the bottom of the stack is essentially shifting the logical stack upward. Note that this specification allows not only upward but also downward logical stack shifts. Thus, this modified specification allows the superoptimizer to discover hardware-specific optimizations that otherwise cannot be discovered when using the straightforward specification.

Sliding Window

Our current tool can superoptimize approximately 16 instructions in a reasonable amount of time. Often, straight-line portions of programs contain more than 16 instructions. Therefore, we have to decompose a long sequence of instructions into smaller ones and run the superoptimizer on the smaller sequences in order to make superoptimization technique scalable. Instead of breaking the long sequence into multiple fixed-length sequences, our *sliding-window* technique adaptively merges superoptimizable units, which usually contain a few instructions, into a *superoptimizable segment*, which will be given as an input to the su-

peroptimizer. The sliding-window technique makes our modular superoptimization more effective when superoptimization instances timeout or do not find better (more optimal) solutions. More specifically, in these scenarios, instead of entirely skipping the current fixed sequence and working on the next one, the superoptimizer will adjust its window size and attempt to optimize a part of the same sequence (with or without additional instructions) again.

Given a sequence of superoptimizable units called a *unit sequence*, the sliding window technique proceeds as follows.

1. Start with an empty superoptimizable segment.
2. Append the superoptimizable unit at the head of the unit sequence to the superoptimizable segment, until the number of instructions is greater than the upper bound.
3. Superoptimize the segment.
4. If a valid superoptimized segment is found, append the segment to the global output, and repeat from 1. If no valid superoptimized segment is found, append only the first unit to the global output, remove the first unit from the superoptimizable segment, and repeat from 2. If superoptimization times out, add the last unit from the segment back to the head of the sequence, and repeat from 3.
5. The process is done when the unit sequence is empty.

Alternatively, dynamic programming, as used in peephole superoptimization [5], can be applied to produce an even more optimal result, but it requires more time than does the sliding windows technique. Dynamic programming is appropriate for peephole superoptimization because the window size is only up to three instructions, while our window size is up to 16 instructions.

Superoptimization and Program Encoding

Given a program segment and its specification as described in the previous section, our superoptimizer uses counterexample-guided inductive synthesis (CEGIS) to search for an equivalent program segment [39]. Within the CEGIS loop, we use the Z3 [12] SMT solver to perform the search.

We model the program segment's approximate execution time based on the cost of each instruction as provided by GreenArrays. We use this cost model to perform a binary search over generated programs looking for optimal performance. Each step involves looking for a program that finishes under a certain time limit by adding that time as a constraint to our SMT formula and synthesizing a program that meets both our performance and our correctness criteria. We can similarly optimize for the length of the program segment instead of its execution time.

Encoding to SMT Formulas

At a particular point of a program, the state of the machine consists of two registers, the data stack, the return stack, memory, and stack pointers. Since each core can communicate with its four neighbors, we represent the data that the core receives and sends using a communication channel, which is an ordered list of (data, neighbor port, read/write) tuples. Hence, the machine's state also includes a communication channel representing the data the core expects to receive or send along with the relevant ports. We use this communication channel to preserve the order of receives and sends to prevent deadlock.

The stacks, the memory, and the communication channel are represented by large bitvectors because Z3 can handle large bitvectors much faster than arrays of integers or arrays of bitvectors. Each instruction in a program converts a machine's state into a new machine's state. We encode each instruction in our SMT formula as a switch statement that alters a machine's state according to which instruction value is chosen.

Address Space Compression

Address space compression is necessary to scale superoptimization to large problems. Each core in GA144 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 occupies 32 words of memory. When the formula generator translates programs to formulas, it discards the free memory space and includes just enough memory to contain all variables and arrays; the smaller the memory, the smaller the search space.

Arrays occupy substantial memory space but are usually accessed with a symbolic index during superoptimization. The index is symbolic if it is an expression of one or more variables as it depends on the values of those variables. In light of this observation, we compress the memory of the input program by truncating each array to contains only two entries, 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. Verification is much faster than synthesis, so we can verify programs with a full address space in a reasonable amount of time.

2.5 Interactions Between Steps

Since our compilation problem is decomposed into four subproblems, we lose some optimization opportunities, and in some circumstances the compiler produces program fragments that do not fit on cores. We will discuss these issues in this section.

Program Size and Iterative Refinement Method

One goal of our compiler is to partition a high-level program into program fragments such that each fragment can fit in a core. Although the partitioning synthesizer overapproximates the size of each fragment, it still does not consider all communication code. For example, assume that partition A sends some data to partition B. The partitioner increases the sizes of both partitions A and B to reflect the effects of the necessary communication code. However, after the layout step, it is possible that partition A and B are not next to each other. In this case, partition A communicates to partition B via one or more intermediate partitions. Since the partitioner does not have any knowledge about the intermediate nodes, the partitioner does not take into account the space occupied by the communication code associated with the intermediate nodes. As a result, it is possible that the generated program partitions will be too large.

For most programs, our compiler generates final programs that fit in cores. Occasionally, the estimation fails, and an iterative refinement reruns the compilation with larger estimation for the too-large fragments, until all final fragments fit in cores.

Optimization Opportunity Loss

There are some lost optimization opportunities that result from decomposing the compilation problem into smaller subproblems. We discuss a few examples of optimization losses in this section.

First, partitioning before optimizing may lead to missed opportunities. For example, let A, B, and C be program fragments that do not fit in one core. Assume the partitioner groups A and B together because that yields the lowest communication count. However, if B and C are grouped together, the superoptimizer may find a very large execution time reduction such that grouping B and C together yields faster code than grouping A and B does.

Second, our schedule-oblivious routing strategy introduces another potential loss. Assume core A can communicate with core B via either core X or Y, and X is very busy before A sends data to B, while Y is not. The current routing strategy will route data from A to B via either X or Y arbitrarily. However, in this particular case, we should route through Y so that B will receive the data from A more quickly, without having to wait for X to finish its work.

Finally, the scope of superoptimization may prevent some optimizations. We do not optimize across superoptimizable segments, because we want the compiler to finish in a reasonable amount of time. However, knowing the semantics of the segments that come before the current segment could definitely allow the superoptimizer to discover additional optimizations. Increasing the scope to include loops and branches will help even more.

Chapter 3

Evaluation

In this section, we present the results of running programs on the GA144 chip to test our hypothesis that using synthesis provides advantages over traditional compilation.

Hypothesis 1 *The 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. This heuristic partitioning strategy is similar to the merging algorithm used in the instruction partitioner in the space-time scheduler for Raw [26]. 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 between every pair of partitions that communicate is equal to one. Third, we compare the performance of programs generated with and without superoptimization. Last, we compare sliding-window algorithm against fixed-window algorithms, in which the superoptimization windows are fixed.

For each benchmark, five different versions of the program are generated.

1. ***sliding s+p+l***: sliding-window superoptimization, partitioning synthesizer, and layout synthesizer
2. ***fixed s+p+l***: with fixed-window superoptimization, partitioning synthesizer, and layout synthesizer
3. ***ns+p+l***: with no superoptimization, partitioning synthesizer, and layout synthesizer
4. ***ns+hp+l***: with no superoptimization, heuristic partitioner, and layout synthesizer
5. ***ns+hp+il***: with no superoptimization, heuristic partitioner, and imprecise layout synthesizer

We run five benchmarks in this experiment.

- *Prefixsum* sequentially computes the prefixsum of a distributed array spanning 10 cores.
- *SSD* computes the 36-bit sum of squared distance between two distributed 18-bit arrays of size 160, each of which spans four cores. SSDs of different chunks of an array can be computed in parallel since there is no dependency between them.
- *Convolution* performs 1D convolution on a 4-core distributed array with kernel's width equal to five in parallel. The program first fills in the ghost regions to eliminate loop dependency before the main convolution computation starts.
- *Sqrt* computes the 16-bit square roots of 32-bit inputs.
- *Sin-Cos* computes $cos(x)$ and $sin(x)$.

The execution time result shown in Figure 3.1 confirms our hypothesis. First, comparing *ns+p+l* (third bars) vs. *ns+hp+l* (fourth bars) shows that the partitioning synthesizer offers 5% on average and up to 11% speedup over the heuristic partitioner. Second, comparing *ns+hp+l* (fourth 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 count into account, it fails to group the heavily communicating cores next to each other; as a result, the communication paths of different parallel groups share some common cores, preventing those groups from running in parallel. In Prefixsum, the imprecise layout generates program fragments that are too large. Third, comparing *sliding s+p+l*

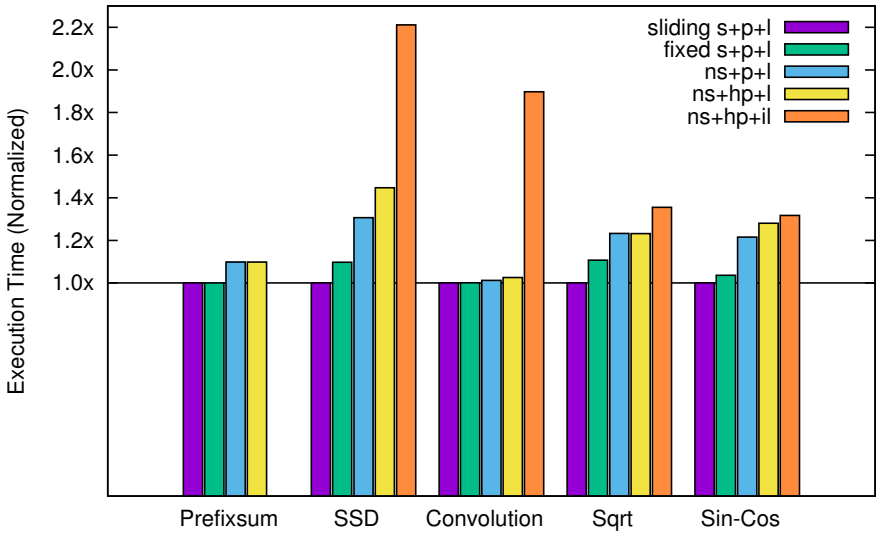


Figure 3.1: Execution time of multicore benchmarks normalized to program generated by the complete synthesizing compiler

(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 programs generated with sliding-window superoptimization are 4% on average and up to 11% faster than programs generated with fixed-window strategy.

Hypothesis 2 *The partitioning synthesizer produces smaller programs and is more robust than the heuristic one.*

The previous experiment shows that the partitioning synthesizer does not generate a slower program for any of the five benchmarks. In this experiment, we look at the number of cores the programs occupy, on the same set of benchmarks. In three out of five benchmarks, the synthesizer generates programs that require significantly fewer cores (using 50-72% of the number of cores used by the heuristic).

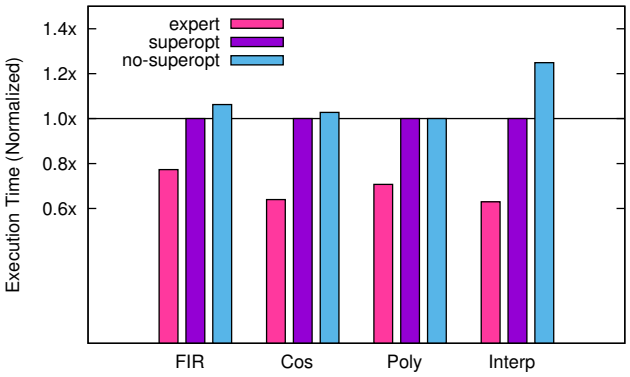
Another experiment also shows that the heuristic algorithm requires parameter tuning specific to each program, while synthesis does not. The heuristic partitioner does not account for the space occupied by communication code because calculating the size of communication code precisely is complicated in the heuristic one. Therefore, we set the space limit per core by scaling the available space by a factor k, ranging between 0 and 1, in the heuristic partitioner. The higher the scaling factor, the smaller the number of 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). Hence, the synthesizer is more robust than the heuristic.

Hypothesis 3 *Programs generated with synthesis are comparable to highly-optimized expert-written programs.*

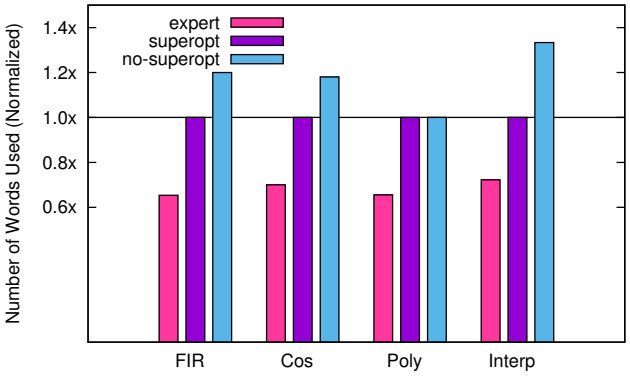
We compare the execution time and program size of highly-optimized programs written by GA144 developers, programs generated with superoptimization, and programs generated without superoptimization. We have access to the following single-core expert-written programs.

- *FIR* applies 16th-order discrete-time finite impulse response filter on a sequence of samples.
- *Cos* computes cosine.
- *Polynomial* evaluates a polynomial using Horner's method given the coefficients and an input.
- *Interp* performs linear interpolation on input data given a sequence of reference points.

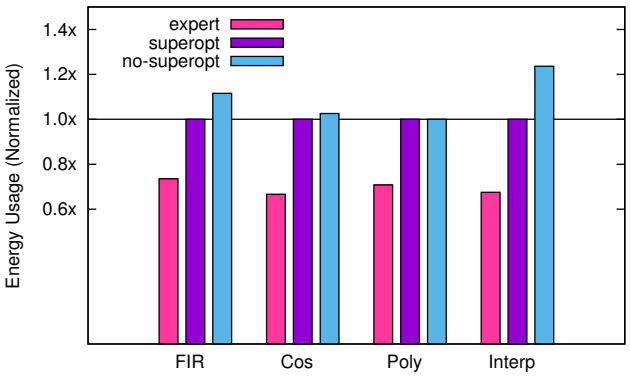
Figure 3.2 shows that our generated programs are 46% slower, 44% less energy-efficient, and 47% longer than the experts' on average, and the superoptimizer improves the running



(a) Execution time



(b) Space



(c) Energy Usage

Figure 3.2: Single-core benchmarks

time by 7%, reduces the energy used by 8%, and shortens the program length by 14% compared to no superoptimization on average.

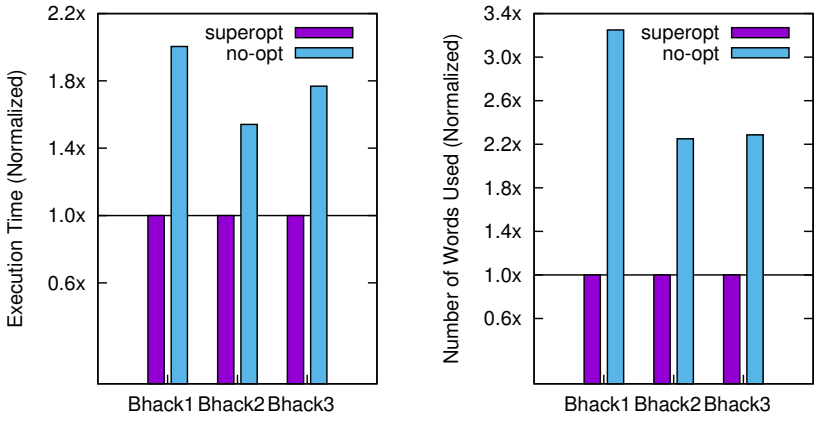
The only multicore application written by experts against which we can compare is MD5 hash function. The other applications published on the GreenArrays website, including SRAM control cluster, programmable DMA channel, and dynamic message routing, require interaction with a GA virtual machine and specific I/O instructions for accessing external memory that Chlorophyll does not support. The MD5 benchmark computes 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 no partition annotations to the operators, 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 partitioning synthesizer. We first ignore all functions except main. After we solve main, we reintroduce other functions 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. Thus, we can generate code for different partitioning (without superoptimization) in a very short amount of time.

We generate two versions of MD5 program. First, we partition the program such that the generated non-superoptimized code is slightly bigger than memory, but the excess is small enough that the final superoptimized code still fits. We also generate a second version that fits on cores without superoptimization. The generated program with superoptimization is 7% faster and 19% more energy-efficient than the one without superoptimization, and uses 10 fewer cores. Compared to the experts' implementation, it is only 19% slower and 31% less energy-efficient, and it uses two-times more cores. This result confirms that our generated programs are comparable with experts' not only on small programs but also on a real application.

Hypothesis 4 *The 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*, $(x \oplus y) \oplus (x\&y)$. Figure 3.3 shows that superoptimization provides 1.8x speedup and 2.6x code length reduction on average. The superoptimizer successfully discovers bit tricks $x\& \sim y$, $\sim x+y$ 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 CSE within program segments, and exploits special instructions that do not exist in common ISAs. Hence, the superoptimizer can discover an unlimited number of optimizations specific to the machine, while the optimizing compiler can only perform a limited number of optimizations implemented by the compiler developers.



(a) Execution time (b) Space

Figure 3.3: Bithack benchmarks

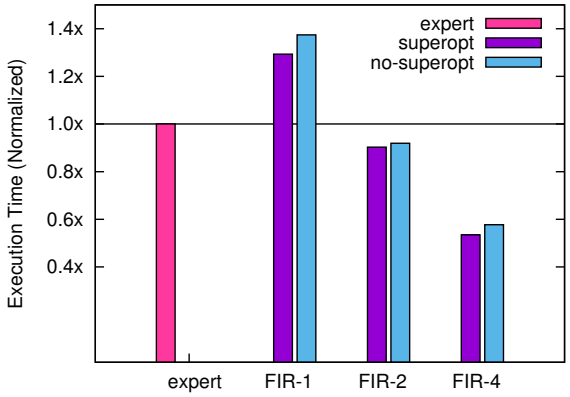


Figure 3.4: FIR benchmark

Hypothesis 5 *Chlorophyll increases programmers' 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 the GA144 and TI MSP430 micro-controller. He managed to learn arrayForth to program the GA144. However, he was able to implement only two benchmarks: FIR and a simple pedometer application [3]. In contrast, with our compiler, we can implement five different FIR implementations within an afternoon. Figure 3.4 shows the running time of three different implementations of FIR—sequential FIR-1, parallel FIR-2 on two cores, and parallel FIR-4 on four cores—as well as the experts' implementation. 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 to exploit parallelism to get the fastest implementation. Although superoptimization makes compilation slower, we can still test implementations quickly by running the non-superoptimized program for a rough estimate of the performance.

Hypothesis 6 *The compiler can be improved by providing more human insights 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. Hoewver, an even faster division can be implemented when a divisor is known; $x/k = (k_1 * x) >> k_2$ 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 specification. Then, we provide this template to the superoptimizer to fill in the numbers for a specific divisor, similar to Sketch [39]. Given the template, the compiler can produce a program that is 6-time faster and 3-time shorter than the experts' general integer division program within three seconds. In theory, the superoptimizer can discover the entire program without the sketch, but it could take much longer time to synthesize since this program is 33-instruction long.

Thus, adding more templates improves performance of generated programs and scalability of the synthesizer. Regarding the performance improvement, this is similar to implementing optimizations for traditional compilers. However, synthesis is in general more powerful because it does not rely on a lookup table and simply discovers faster code by searching.

Figure 3.5 and 3.6 show the compile times for the single-core benchmarks and multicore benchmarks used in our experiments respectively. Partitioning is also slow, but such algorithms are generally slow; consider, for example, partitioning for FPGA [43]. We address the issue by allowing programmers to accelerate the partitioning process by pinning data or code to specific cores when they have relevant insights.

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

Figure 3.5: Superoptimization time (in hours) and program length (in words) for single-core benchmarks. A word in program sequences contains either four instructions or a constant literal. *Bithack-3 takes 25.08 hours when the program segment length is capped at 30 instructions. With the default length (16 instructions), it takes 2.5 hours.

Benchmarks	# of Given Core	Loop Split (s)	Part (s)	Layout (s)	Superopt (hr)
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	16.07

Figure 3.6: Compile time of multicore benchmarks. Time is in seconds except for superoptimization time, which is in hours. The compiler runs on an 8-core machine, so it superoptimizes up to eight independent instances in parallel. Layout time only depends on the number of given GA cores. Heuristic partitioning takes less than one second to generate a solution.

Chapter 4

Related Work

4.1 Programming Models

A number of programming models have been developed for spatial architectures for different application domains. StreamIt, a programming model for streaming applications, decomposes the compilation problem much as we do [16]. Partitions are defined by programmers using *filters* and can be merged by the compiler. GA144 also shares many characteristics with systolic arrays. Systolic arrays are designed for massively parallel applications such as applications with rhythmic communications [22]. Thus, the programming model for systolic arrays is domain-specific, tailored to such applications [24, 21]. Unlike StreamIt or Systolic, Chlorophyll targets more general-purpose programming.

The high performance computing (HPC) community has developed 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. We generate code without these guards by splitting loops and statically determining the partitions for every variable and operation at compile time. The partitioning problem also appears in the HPC domain. Many Distributed Fortran compilers simply apply an "owner computes" rule, distributing data and computation to align with the output data's positions [7, 30]. 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 large.

Our memory model is Partitioned Global Address Space (PGAS), a model used many languages [9, 44, 31, 34]. Although these languages offer programmers control over mapping operators to computing resources, they do not provide the programmers an easy way of exploring different mappings.

4.2 Type Systems

Many distributed programming languages have exploited type systems to ensure properties of interest. Delaval et al. presented a type system for the automatic distribution of high-order synchronous dataflow programs, allowing programmers to localize some expressions onto processors [13]. The type system can infer the localization of non-annotated values to ensure the consistency of the distribution. Like our compiler, the framework generates local programs to be executed by each computing resource from a centralized typed program. X10 introduces *place type* and exploits type inference to eliminate dynamic references of global pointers [10]. Titanium, similarly, uses type inference to minimize the number of global pointers in the program [27].

4.3 Heuristic-based Compilers

There is substantial work on heuristic-based compilers for spatial architectures. The partitioning and placement algorithms used in TRIPS compiler, Raw space-time scheduler, and Occam to transputer system, may be applied with some modifications to our problem. However, these architectures are substantially different from GA144.

TRIPS compiler distributes a computation DAG of up to 128 instructions in each hyperblock onto 16 cores [8, 38]. Chlorophyll partitions much larger programs—MD5, for example has, 4,600 instructions—with loops and branches onto 144 cores. TRIPS also has hardware-supported routing, while GA144 does not. In Raw compiler, the space-time scheduler decomposes the partitioning problem into three subproblems: clustering, merging, and global data partitioning [26], while Chlorophyll solves the partitioning problem as one problem. The merging algorithm is essentially the same as the heuristic partitioner with which we compare Chlorophyll in our evaluation. The transputer compiler and StreamIt's Raw compiler also use SA for solving the layout problem [36, 16].

4.4 Constraint-based Compilers

Though not as common as heuristic-based compilers, constraint-based compilers have been studied and used in practice.

Vivado Design Suite performs High-Level Synthesis that transforms a C, C++ or SystemC design specification into a RTL implementation, which in turn can be synthesized onto a FPGA [43]. The programmer can specify additional constraints using directives, such as controlling the binding process of operations to cores, albeit in ways that are much more limited than our programming model facilitates. For example, multiplication is implemented by a specific hardware multiplier in the RTL design using a specific core.

Yuan et al. solve hardware/software partitioning and pipelined scheduling on runtime reconfigurable FPGAs using an SMT solver [45]. Although the problem domains of our compiler and of their partitioner and scheduler are different, Yuan et al. also shows that

solutions obtained from the SMT solver are superior to the solutions obtained from a heuristic algorithm, but that constraint solving techniques face scalability challenges.

Another constraint-based approach to solve the placement and routing problems uses ILP to map the computation DAG to the graph representing the hardware's structure [32]. The constraints represent placement of computation, data routing, event timing, resource utilization, and optimization for the hardware-specific objective function. However, we cannot apply this technique directly to our partition and layout problems because our computation graphs contain cycles, and the case-study architectures in the ILP scheduling paper include hardware support for data routing, unlike GA144.

4.5 Superoptimization

The original superoptimizer by Massalin finds the shortest optimized version of a program by enumerating every possible program [29]. Each candidate program is checked on manually supplied test cases. A more recent take on superoptimization is Denali [23], which uses goal-directed search, allowing it to scale better. As in our system, the search is performed by an automated theorem prover. *Stochastic* superoptimization [35] 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. We may speed up our superoptimization step by using this technique, but different types of mutations may be required to make this technique work well with stack-based architectures. Superoptimization has also been used to generate peephole optimization rules [4]. Unlike Chlorophyll, the peephole superoptimizer optimizes all possible sequences of up to 4 instructions offline (before compile time). We cannot afford this offline technique since we wish to superoptimize much longer sequences of instructions.

Another variation on superoptimization is *component-based synthesis* [20]. 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 suite our use case 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.

Program synthesis is also used to optimize domain-specific programs. An automatic SIMD vectorization restructures a loop to expose data parallelism, extracts the equivalence relation from the loop body, and then synthesizes a new loop body [6]. Unlike our superoptimizer, this SIMD synthesizer searches by enumeration with heuristics to reduce synthesis time.

Chapter 5

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, the traditional compilation approach becomes even more difficult and less practical to implement.

We have built the first synthesis-aided compiler for extremely minimalist architectures and introduced a new spatial programming model to provide programmability for programmer-unfriendly hardware. Our compiler decomposes the compilation problem into smaller subproblems that can be solved by various synthesizers or easy-to-implement transformations. Although program synthesis may not scale to large problems on its own, our work shows that we can overcome these issues by decomposing problems into smaller ones and relying on more human insight.

The contribution of this thesis is not that our algorithms for partitioning, layout, routing, and code generation are individually superior to the existing ones, but we show that our compiler is simpler than a classical compiler while producing comparable code. Program synthesis techniques enable compiler developers to quickly develop a new high-performance compiler for a radical architecture without knowing how to implement optimizations specific to the architecture.

Bibliography

[1] Alur, R. Syntax-guided synthesis. Tutorial at FMCAD.

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

[3] Avizienis, R., and Ljung, P. Comparing the Energy Efficiency and Performance of the Texas Instrument MSP430 and the GreenArrays GA144 processors. Tech. rep., 2012.

[4] Bansal, S., and Aiken, A. Automatic generation of peephole superoptimizers. In *ASPLOS* (2006).

[5] Bansal, S., and Aiken, A. Binary translation using peephole superoptimizers. In *OSDI* (2008).

[6] Barthe, G., Crespo, J. M., Gulwani, S., Kunz, C., and Marron, M. From relational verification to simd loop synthesis. In *PPoPP* (2013).

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

[8] Burger, D., Keckler, S. W., McKinley, K. S., Dahlin, M., John, L. K., Lin, C., Moore, C. R., Burrill, J., McDonald, R. G., Yoder, W., and Team, t. T. Scaling to the end of silicon with edge architectures. *Computer* (July 2004).

[9] Carlson, W. W., Draper, J. M., and Culler, D. E. S-246, 187 introduction to upc and language specification.

[10] Chandra, S., Saraswat, V., Sarkar, V., and Bodik, R. Type inference for locality analysis of distributed data structures. In *PPoPP* (2008).

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

[12] De Moura, L., and Bjørner, N. Z3: An efficient smt solver. In *TACAS* (2008).

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

[14] Dorigo, M., and Stützle, T. *Ant Colony Optimization*. Bradford Company, Scituate, MA, USA, 2004.

[15] Fraser, C. W., Henry, R. R., and Proebsting, T. A. Burg: fast optimal instruction selection and tree parsing. *SIGPLAN Not. 27*, 4 (Apr. 1992), 68–76.

[16] Gordon, M. I., Thies, W., Karczmarek, M., Lin, J., Meli, A. S., Lamb, A. A., Leger, C., Wong, J., Hoffmann, H., Maze, D., and Amarasinghe, S. A stream compiler for communication-exposed architectures. In *ASPLOS* (2002).

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

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

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

[20] Gulwani, S., Jha, S., Tiwari, A., and Venkatesan, R. Synthesis of loop-free programs. In *PLDI* (2011).

[21] Hughey, R. Programming systolic arrays. Tech. rep., Brown University, 1992.

[22] Johnson, K. T., Hurson, A. R., and Shirazi, B. General-purpose systolic arrays. *Computer 26*, 11 (Nov. 1993), 20–31.

[23] Joshi, R., Nelson, G., and Randall, K. Denali: a goal-directed superoptimizer. In *PLDI* (2002).

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

[25] Lawler, E. L. The quadratic assignment problem. *Manage. Sci. 9* (1963), 586–599.

[26] Lee, W., Barua, R., Frank, M., Srikrishna, D., Babb, J., Sarkar, V., and Amarasinghe, S. Space-time scheduling of instruction-level parallelism on a raw machine. In *ASPLOS* (1998).

[27] Liblit, B., Aiken, A., and Yelick, K. Type systems for distributed data sharing. In *Static Analysis*, R. Cousot, Ed., vol. 2694 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003, pp. 273–294.

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

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

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

[31] Nieplocha, J., Palmer, B., Tipparaju, V., Krishnan, M., Trease, H., and Aprà, E. Advances, applications and performance of the global arrays shared memory programming toolkit. *Int. J. High Perform. Comput. Appl. 20*, 2 (May 2006), 203–231.

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

[33] Robison, A. D. Impact of economics on compiler optimization. In *Joint ACM-ISCOPE Conference on Java Grande* (2001).

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

[35] Schkufza, E., Sharma, R., and Aiken, A. Stochastic superoptimization. In *ASPLOS* (2013).

[36] Shen, H. Occam implementation of process-to-processor mapping on the hathi-2 transputer system. *Microprocessing and Microprogramming 33*, 3 (1992).

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

[38] Smith, A., Gibson, J., Maher, B., Nethercote, N., Yoder, B., Burger, D., McKinle, K. S., and Burrill, J. Compiling for edge architectures. In *CGO* (2006).

[39] Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., and Saraswat, V. Combinatorial sketching for finite programs. In *ASPLOS* (2006).

[40] Team, T. Can Intel Challenge ARM's Mobile Dominance?, 2012.

[41] Torlak, E., and Bodik, R. Growing solver-aided languages with Rosette. In *Onward!* (2013).

[42] Torlak, E., and Bodik, R. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI* (2014), pp. 530–541.

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

[44] Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., and Aiken, A. Titanium: A high-performance java dialect. In *Concurrency: Pract. Exper.* (1998).

[45] Yuan, M., Gu, Z., He, X., Liu, X., and Jiang, L. Hardware/software partitioning and pipelined scheduling on runtime reconfigurable fpgas. *ACM Trans. Des. Autom. Electron. Syst. 15*, 2 (Mar. 2010).