Chlorophyll

Synthesis-Aided Compiler for Low-Power Spatial Architectures

Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Ras Bodik
unusual ISA

- small memory
- narrow bitwidth

- no cache
- limited interconnect

spatial & temporal partitioning
unusual ISA

- small memory
- narrow bitwidth
- no cache
- limited interconnect
- spatial & temporal partitioning
Need a new way of building a compiler!

- small memory
- narrow bitwidth
- no cache
- limited interconnect
- spatial & temporal partitioning
Synthesis-Aided Compiler
## Classical vs. Synthesis Compiler

<table>
<thead>
<tr>
<th></th>
<th>Classical</th>
<th>Synthesis-Aided</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Approach</strong></td>
<td>Apply heuristic transformations</td>
<td>Find best program in defined search space</td>
</tr>
<tr>
<td><strong>Required Components</strong></td>
<td>• Transformations</td>
<td>• Defined search space</td>
</tr>
<tr>
<td></td>
<td>• Legality analysis</td>
<td>• Equivalence checker</td>
</tr>
<tr>
<td></td>
<td>• Heuristics</td>
<td>• Abstract cost function</td>
</tr>
<tr>
<td><strong>Output’s Performance</strong></td>
<td>Depends on heuristic quality</td>
<td>Optimal in defined search space</td>
</tr>
<tr>
<td><strong>Building Effort</strong></td>
<td>High</td>
<td>Low</td>
</tr>
</tbody>
</table>

- **Classical Synthesis - Aided Approach**
  - Apply heuristic transformations
  - Find best program in defined search space
  - Required Components: Transformations, Legality analysis, Heuristics
  - Output’s Performance: Depends on heuristic quality
  - Building Effort: High

- **Synthesis Compiler**
  - High building effort
Case study: GreenArrays Spatial Processor

On FIR benchmark, [Avizienis, Ljung]
GA144 is 11x faster and simultaneously 9x more energy efficient than TI MSP 430.

Specs
• Stack-based 18-bit architecture
• 144 tiled cores
• Limited communication (neighbors only)
• No cache, no shared memory
• < 300 bytes of memory per core
• 32 instructions

Example challenges of programming spatial architectures like GA144:
• Bitwidth slicing: Represent 32-bit numbers by two 18-bit words
• Function partitioning: Break functions into a pipeline with just a few operations per core.
Our Contributions

Spatial programming model
  – Flexible control over partitioning

Low-effort approach to compiler construction
  – Solved a compilation problem as a synthesis problem
  – To scale synthesis, decomposed it into subproblems

Empirical evaluation
  – Easy-to-build compiler architecture
  – Performance within 2x of expert-written code
Compiler Workflow

Spatial Program

Partitioner
Program + partition annotation
(logical cores)

Layout
Program + location annotation & routing info (physical cores)

Code Separator
Per-core code with communication code

Code Generator
Per-core optimized machine code

constraint solving minimizing # of msgs

Quadratic Assignment Problem

a traditional transformation

superoptimization
Spatial programming model

Spatial Program

Partitioner
Program + partition annotation
(logical cores)

Layout
Program + location annotation & routing info (physical cores)

Code Separator
Per-core code with communication code

Code Generator
Per-core optimized machine code
Spatial programming model

How does one want to program a spatial architecture?

1. Write algorithm in high-level language without dealing with low-level details
2. Have control over partitioning of data and computation if desired

```c
int a, b;
int ans = a * b;
```
Spatial programming model

```c
int @1 a, b;
ext @3 ans = a * @2 b;
```

**Partition Type**

*pins data and operators to specific partitions (logical cores)*

Similar to [Chandra et al. PPoPP’08]
int@1 a, b;
int@3 ans = a * @2 b;

Do not need to handle data routing and communication code
Distributed Partition Type

```c
int@6 k[10];

int@[0:5]=6, [5:10]=7 k[10];

int::2@[0:10]=(6,7) k[10];
```
Unspecified Partitions

How to compile a partially annotated program?

```c
int a, b;
int@3 ans = a * b;
```
Unspecified Partitions

How to compile a partially annotated program?

```c
int a, b;
int ans = a * b;
```
Partitioning Synthesizer

Program

Partitioner

Program + partition annotation (logical cores)

Layout

Program + location annotation & routing info (physical cores)

Code Separator

Per-core code with communication code

Code Generator

Per-core optimized machine code

constrain solving to minimize # of msgs
How Does Partitioning Synthesizer Work?

**Idea:** infer partition types subject to

- Communication count constraint
  
  # of messages is minimized

- Space constraint
  
  code and data fit in each core

**How:** use Rosette (by Emina Torlak, Session 9A)

- Implement a type checker
- Get type inference for free
Layout Synthesizer

Program

Partitioner

Program + partition annotation (logical cores)

Layout

Program + location annotation & routing info (physical cores)

Code Separator

Per-core code with communication code

Code Generator

Per-core optimized machine code

Quadratic Assignment Problem
• Simulated annealing
code separator

program

partitioner

program + partition annotation (logical cores)

layout

program + location annotation & routing info (physical cores)

code separator

a traditional transformation

code generator

per-core code with communication code

per-core optimized machine code
Code Generator

Program → Partitioner
Program + partition annotation (logical cores) → Layout
Program + location annotation & routing info (physical cores) → Code Separator
Per-core code with communication code → Code Generator
Per-core optimized machine code

superoptimization
Code Generator

Classical compiler backend

IR¹ → IR² → IR³ → ... → Optimized machine code

Optimizing Code Gen

Our compiler backend

Naïve Code Gen

A

B

C

D

E

Optimal program = minimum cost

Super-optimizer

Machine code

[Massalin et al. ASPLOS’87, Bansal et al. ASPLOS’06, Gulwani et al. PLDI’11, ...]

Super-optimized machine code
• Counter-Example-Guided Inductive Synthesis (CEGIS)
  • encode program as SMT formula
  • solve using Z3

• Minimizing one of:
  • Execution time
  • Energy consumption
  • Program length
Problem with Superoptimizers

• Synthesizing the entire program is not scalable.
  – Start-of-the-art synthesizers can generate up to 25 instructions [Schkufza et al. ASPLOS’13, Gulwani et al. PLDI’11].

• Must decompose the superoptimization.
Modular Superoptimizer

Naïve Code Gen

Basic block A

- i
- ii
- iii
- iv
- v
- vi
- vii
- viii
- ix

An instruction

A

B

C

D

E

loop

if
Modular Superoptimizer

Naïve Code Gen

Basic block A

segment <= 16 instructions
Naïve Way to Decompose

Fixed Windows

block A

<table>
<thead>
<tr>
<th>i</th>
<th>ii</th>
<th>iii</th>
<th>iv</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>vi</td>
<td>vii</td>
<td>viii</td>
</tr>
<tr>
<td>ix</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

segment i

segment j

segment k

block A’

i’

j’

k’

Timeout

Superoptimizer

Minimize Running Time

CEGIS

Minimize Running Time
A Better Way

Sliding Window

block A

block A'

sliding window

i

ii

iii

iv

v

vi

vii

viii

ix

Superoptimizer

Minimize Running Time

CEGIS

Timeout
A Better Way

Sliding Window

No segment with lower cost
A Better Way

Sliding Window

Find segment with lower cost
A Better Way

Sliding Window

block A

<table>
<thead>
<tr>
<th>i</th>
<th>ii</th>
<th>iii</th>
<th>iv</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>vi</td>
<td>vii</td>
<td>viii</td>
<td>ix</td>
<td></td>
</tr>
</tbody>
</table>

segment k

block A’

<table>
<thead>
<tr>
<th>i</th>
<th>j’</th>
</tr>
</thead>
<tbody>
<tr>
<td>k’</td>
<td></td>
</tr>
</tbody>
</table>
Address Space Compression

Trick to speed up synthesis time

Program $P$ with memory

compress $\downarrow$

Program $P_c$ with memory

$\downarrow$

Superoptimizer

$\downarrow$

Program $P'_c$ with memory

decompress $\downarrow$

Program $P'$ with memory

array $z$ (10 entries) $\times y$

array $z$ (2 entries) $\times y$

array $z$ (2 entries) $\times y$

array $z$ (10 entries) $\times y$

Then verify if $P \equiv P'$
Empirical Evaluation

Hypothesis 1
Synthesis generates faster code than a heuristic compiler.

Synthesizing partitioner vs. Heuristic partitioner
a greedy algorithm
Empirical Evaluation

Hypothesis 1
Synthesis generates faster code than a heuristic compiler.

Synthesizing partitioner vs. Heuristic partitioner
Precise layout vs. Less precise layout
assumes each message is sent once
Hypothesis 1
Synthesis generates faster code than a heuristic compiler.

<table>
<thead>
<tr>
<th>Synthesizing partitioner</th>
<th>vs.</th>
<th>Heuristic partitioner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precise layout</td>
<td>vs.</td>
<td>Less precise layout</td>
</tr>
<tr>
<td>Superoptimizer</td>
<td>vs.</td>
<td>No superoptimizer</td>
</tr>
<tr>
<td>Sliding window</td>
<td>vs.</td>
<td>Fixed window</td>
</tr>
</tbody>
</table>
Empirical Evaluation

Hypothesis 1
Synthesis generates faster code than a heuristic compiler.

- Precise layout $\rightarrow$ up to 1.8x speedup
- Synthesizing partitioner $\rightarrow$ 5% speedup
- Superoptimizer $\rightarrow$ 15% speedup
  (4% from sliding window)

Better
Empirical Evaluation

Hypothesis 2
Our compiler produces code comparable to the expert’s code.

On MD5 benchmark, the expert uses many advanced tricks:
• 10 cores
• Self-modifying code
• Circular array data structure
• Different modes of operations for different cores
  • Instruction fetch from local memory
  • Instruction fetch from neighbors

We define success to be within 2x of the expert’s code.
Empirical Evaluation

Hypothesis 2
Our compiler produces code comparable to the expert’s code.

On 4 smaller benchmarks, Chlorophyll was on average
• 46% slower
• 44% less energy-efficient

On a larger benchmark (MD5), Chlorophyll was
• 65% slower
• 69% less energy-efficient
Hypothesis 3
Chlorophyll increases programmer productivity and offers the ability to explore different implementations quickly.
Using program synthesis as a core compiler building block enables us to build a new compiler with low effort that still produces efficient code.

Thank you