Program Synthesis for Low-Power Accelerators

Ras Bodik
Mangpo Phitchaya Phothilimthana
Tikhon Jelvis
Rohin Shah
Nishant Totla
What we do and talk overview

Our main expertise is in **program synthesis**

a modern alternative/complement to compilation

Our applications of synthesis: hard-to-write code

parallel, concurrent, dynamic programming, end-user code

In this project, we explore spatial accelerators

an accelerator programming model aided by synthesis
Future Programmable Accelerators
no cache coherence
limited interconnect
no clock!
spatial & temporal partitioning
crazy ISA
small memory
back to 16-bit nums
crazy ISA
small memory
back to 16-bit nums

no cache coherence
limited interconnect
no clock!

spatial & temporal partitioning

good programming model
What we desire from programmable accelerators

We want the obvious conflicting properties:
- high performance at low energy
- easy to program, port, and autotune

Can’t usually get both
- transistors that aid programmability burn energy
  - ex: cache coherence, smart interconnects, ...

In principle, most decisions can be done in compilers
- which would simplify the hardware
  - but the compiler power has proven limited
We ask

How to use fewer “programmability transistors”?

How should a programming framework support this?

Our approach: synthesis-aided programming model
GA 144: our stress-test case study
Low-Power Architectures

thanks: Per Ljung (Nokia)
Why is GA low power

The GA architecture uses several mechanisms to achieve ultra high energy efficiency, including:

- asynchronous operation (no clock tree),
- ultra compact operator encoding (4 instructions per word) to minimize fetching,
- minimal operand communication using stack architecture,
- optimized bitwidths and small local resources,
- automatic fine-grained power gating when waiting for inter-node communication, and
- node implementation minimizes communication energy.
GreenArray GA144

- The GA144 is an 8 x 18 array of F18A asynchronous computers (cores)
- Cores are independent – no shared resources (i.e. clock, cache, memory bus)
  - Only globally routed signals are power/ground and reset! Very scalable architecture.
- Each core can (only!) communicate with its neighbors to the NSEW
- Cores located along perimeter include I/O functionality

Implemented in 180nm process
VDD = 1.8V

Instantaneous power usage ranges from
14 uW - 650 mW

Adjacent cores are mirror images of each other

F18A layout
Each node has its own:
- RAM and ROM
- Return Stack
- Data Stack

Fewer than 20k transistors per core!

Total on-chip RAM = 9216 18-bit words (equivalent amount of ROM)

slide from Rimas Avizienis
MSP430  vs  GreenArray
## Finite Impulse Response Benchmark

<table>
<thead>
<tr>
<th>Performance</th>
<th>MSP430 (65nm)</th>
<th>GA144 (180nm)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F2274 swmult</td>
<td>F2617 hwmult</td>
</tr>
<tr>
<td>usec / FIR output</td>
<td>688.75</td>
<td>37.125</td>
</tr>
<tr>
<td>nJ / FIR output</td>
<td>2824.54</td>
<td>233.92</td>
</tr>
</tbody>
</table>

GreenArrays 144 is **11x faster** and simultaneously **9x more energy efficient** than MSP 430.

Data from Rimas Avizienis
How to compile to spatial architectures
Programming/compiling challenges

Partition the code and map it to cores
  also, design the algorithm for optimal partitioning
Implement inter-core communication
  don’t introduce races and deadlocks
Manage distributed data structures
  for user-facing accelerators (GUIs), it’s not just arrays
Generate efficient code
  often on a limited ISA
Programming/Compiling Challenges

Algorithm/Pseudocode

Partition & Distribute Data Structures

Place & Route Communication

Schedule & Implement code

Comp1
Comp2
Comp3
Send X
Comp4
Recv Y
Comp5
Existing technologies

Optimizing compilers
    hard to write optimizing code transformations

FPGAs: synthesis from C
    partition C code, map it and route the communication

Bluespec: synthesis from C or SystemVerilog
    synthesizes HW, SW, and HW/SW interfaces
What does our synthesis differ?

FPGA, BlueSpec:
  partition and map to minimize cost

We push synthesis further:
  – super-optimization: synthesize code that meets a spec
  – a different target: accelerator, not FPGA, not RTL

Benefits:
  superoptimization: easy to port as there is no need to port code generator and the optimizer to a new ISA
Overview of program synthesis
Synthesis with “sketches”

Extend your language with two constructs

**spec:**
```c
int foo (int x) {
    return x + x;
}
```

**sketch:**
```c
int bar (int x) implements foo {
    return x << ??;
}
```

**result:**
```c
int bar (int x) implements foo {
    return x << 1;
}
```

instead of *implements*, assertions over safety properties can be used
Example: 4x4-matrix transpose with SIMD

a functional (executable) specification:

```c
int[16] transpose(int[16] M) {
    int[16] T = 0;
    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 4; j++)
            T[4 * i + j] = M[4 * j + i];
    return T;
}
```

This example comes from a Sketch grad-student contest
Implementation idea: parallelize with SIMD

Intel SHUFPP (shuffle parallel scalars) SIMD instruction:

\[
\text{return} = \text{shufps}(x1, x2, \text{imm8} :: \text{bitvector8})
\]
High-level insight of the algorithm designer

Matrix $M$ transposed in two shuffle phases

**Phase 1**: shuffle $M$ into an intermediate matrix $S$ with some number of shufps instructions

**Phase 2**: shuffle $S$ into an result matrix $T$ with some number of shufps instructions

Synthesis with partial programs helps one to complete their insight. Or prove it wrong.
The SIMD matrix transpose, sketched

```c
int[16] trans_sse(int[16] M) implements trans {
    int[16] S = 0, T = 0;

    S[??::4] = shufps(M[??::4], M[??::4], ??);
    S[??::4] = shufps(M[??::4], M[??::4], ??);
    ...
    S[??::4] = shufps(M[??::4], M[??::4], ??);

    T[??::4] = shufps(S[??::4], S[??::4], ??);
    T[??::4] = shufps(S[??::4], S[??::4], ??);
    ...
    T[??::4] = shufps(S[??::4], S[??::4], ??);

    return T;
}
```
The SIMD matrix transpose, sketched

```c
int[16] trans_sse(int[16] M) implements trans {
  int[16] S = 0, T = 0;
  repeat (??) S[??::4] = shufps(M[??::4], M[??::4], ??);
  repeat (??) T[??::4] = shufps(S[??::4], S[??::4], ??);
  return T;
}
```

```c
int[16] trans_sse(int[16] M) implements trans { // synthesized code
  S[4::4] = shufps(M[6::4], M[2::4], 11001000b);
  S[0::4] = shufps(M[11::4], M[6::4], 10010110b);
  S[12::4] = shufps(M[0::4], M[2::4], 10001101b);
  S[8::4] = shufps(M[8::4], M[12::4], 11010111b);
  T[4::4] = shufps(S[11::4], S[1::4], 10111001b);
  T[12::4] = shufps(S[3::4], 10111001b);
  T[8::4] = shufps(S[4::4], S[1::4], 10111001b);
  T[0::4] = shufps(S[1::4], S[1::4], 10111001b);
}
```

From the contestant email:
Over the summer, I spent about 1/2 a day manually figuring it out. Synthesis time: <5 minutes.
Demo: transpose on Sketch

Try Sketch online at http://bit.ly/sketch-language
The mechanics of program synthesis (constraint solving)
What to do with a program as a formula?

Assume a formula $S_P(x,y)$ which holds iff program $P(x)$ outputs value $y$

```
program: f(x) { return x + x }
```

```
formula: $S_f(x,y): y = x + x$
```

This formula is created as in program verification with concrete semantics [CMBC, Java Pathfinder, ... ]
With program as a formula, solver is versatile

Solver as an **interpreter**: given $x$, evaluate $f(x)$

\[
S(x,y) \land x=3 \quad \text{solve for } y \quad y \mapsto 6
\]

Solver as a program **inverter**: given $f(x)$, find $x$

\[
S(x,y) \land y=6 \quad \text{solve for } x \quad x \mapsto 3
\]

This solver “bidirectionality” enables synthesis
Search of candidates as constraint solving

\[ SP(x, h, y) \text{ holds iff sketch } P[h](x) \text{ outputs } y. \]

\[
\begin{align*}
\text{spec}(x) & \{ \text{return } x + x \} \\
\text{sketch}(x) & \{ \text{return } x \ll ?? \}
\end{align*}
\]

\[ Sketch(x, y, h): y = x \times 2h \]

The solver computes h, thus synthesizing a program correct for the given x (here, x=2)

\[ Sketch(x, y, h) \land x = 2 \land y = 4 \quad \text{solve for } h \quad h \mapsto 1 \]

Sometimes h must be constrained on several inputs

\[ S(x_1, y_1, h) \land x_1 = 0 \land y_1 = 0 \land \\
S(x_2, y_2, h) \land x_2 = 3 \land y_2 = 6 \quad \text{solve for } h \quad h \mapsto 1 \]
Inductive synthesis

Our constraints encode **inductive synthesis**: 

We ask for a program $P$ correct on a few inputs.
We hope (or test, verify) that $P$ is correct on rest of inputs.
We propose a programming model for low-power devices by exploiting program synthesis.
Programming/Compiling Challenges

Algorithm/Pseudocode

Partition & Distribute Data Structures

Place & Route Communication

Schedule & Implement code

Comp1

Comp2

Comp3

Send X

Comp4

Recv Y

Comp5
Project Pipeline

Language Design
- expressive
- flexible (easy to partition)

Partitioning
- minimize # of msgs
- fit each block in a core

Placement & Routing
- minimize comm cost
- reason about I/O pins

Intracore scheduling & Optimization
- overlap comp and comm
- avoid deadlock
- find most energy-efficient code
Programming model abstractions

L0: high-level language

Partitioning

L1: L0 + virtual communication channels

Placement & Routing

L2: L1 + layout + routing

Inserting communication

L3: L2 + real communication code

Code generation

L4: machine instructions
MD5 case study
MD5 Hash

Figure taken from Wikipedia
MD5 from Wikipedia

```c
// Process the message in successive 512-bit chunks:
for each 512-bit chunk of message
  break chunk into sixteen 32-bit words w[i], 0 ≤ i ≤ 15
// Initialize hash value for this chunk:
var int a := h0
var int b := h1
var int c := h2
var int d := h3
// Main loop:
for i from 0 to 63
  if 0 ≤ i ≤ 15 then
    f := (b and c) or ((not b) and d)
    g := i
  else if 16 ≤ i ≤ 31
    f := (d and b) or ((not d) and c)
    g := (5*i + 1) mod 16
  else if 32 ≤ i ≤ 47
    f := b xor c xor d
    g := (5*i + 3) mod 16
  else if 48 ≤ i ≤ 63
    f := c xor (b or (not d))
    g := (7*i) mod 16
  temp := a
  d := c
  c := b
  b := b + leftrotate([a + f + k[i] + w[g]], r[i])
  a := temp
end for
// Add this chunk’s hash to result so far:
  h0 := h0 + a
  h1 := h1 + b
  h2 := h2 + c
  h3 := h3 + d
end for
var char digest[16] := h0 append h1 append h2 append h3 // (Output is in little-endian)
```

// leftrotate function definition
leftrotate (x, c)
  return (x << c) binary or (x >> (32-c));
Actual MD5 Implementation on GA144
MD5 on GA144

High order

Low order

Each of 64 steps

Once per message block
This is how we express MD5
Project Pipeline

L0: high-level language

Partitioning

L1: L0 + virtual communication channels

Placement & Routing

L2: L1 + layout + routing

Inserting communication

L3: L2 + real communication code

Code generation

L4: machine instructions
Annotation at Variable Declaration

typedef pair<int,int> myInt;

vector<myInt> message[16];
vector<myInt> k[64];
vector<myInt> output[4];
vector<myInt> hash[4];
vector<int> r[64];

@core indicates where data lives.

106

6

k[i]
void@((104,4)) md5() {
    for(myInt@any t = 0; t < 16; t++) {
        myInt@here a = hash[0], b = hash[1], c = hash[2], d = hash[3];
        for(myInt@any i = 0; i < 64; i++) {
            myInt@here temp = d;
            d = c;
            c = b;
            b = round(a, b, c, d, i);
            a = temp;
        }
        hash[0] += a;
        hash[1] += b;
        hash[2] += c;
        hash[3] += d;
    }
    output[0] = hash[0];
    output[1] = hash[1];
    output[2] = hash[2];
    output[3] = hash[3];
}
MD5 in CPart

typedef pair<int,int> myInt;

vector<myInt>@{[0:64]=(106,6)} k[64];

myInt@(105,5) sumrotate(myInt@(104,4) buffer, ...) {
    myInt@h sum = buffer +@here k[i] + message[g];
    ...
}

MD5 in CPart

typedef pair<int,int> myInt;

vector<myInt>@{{0:64}=(106,6)} k[64];

myInt@(105,5) sumrotate(myInt@(104,4) buffer, ...) {
  myInt@h sum = buffer +@here k[i] + message[g];
  ...
}

buffer is at (104,4)
MD5 in CPart

typedef pair<int,int> myInt;

vector<myInt>@{[0:64]=(106,6)} k[64];

myInt@(105,5) sumrotate(myInt@(104,4) buffer, ...) {
    myInt@h sum = buffer +@here k[i] + message[g];
    ...
}

k[i] is at (106, 6)

106 high order

6 low order
MD5 in CPart

typedef pair<int,int> myInt;

vector<myInt>@{[0:64]=(106,6)} k[64];

myInt@(105,5) sumrotate(myInt@(104,4) buffer, ...) {
    myInt@h sum = buffer +@here k[i] + message[g];
    ...
}

+ is at (105,5)

+ high order

105

+ low order

5
MD5 in CPart

typedef pair<int,int> myInt;

vector<myInt>@{[0:64]=(106,6)} k[64];

myInt@(105,5) sumrotate(myInt@(104,4) buffer, ...) {
    myInt@h sum = buffer +@here k[i] + message[g];
    ...
}

Implicit communication in source program.
Communication inserted by synthesizer.
**MD5 in CPart**

typedef pair<int, int> myInt;

vector<myInt>@{[0:64]=(205,105)} k[64];

myInt@(204,5) sumrotate(myInt@(104,4) buffer, ...) {
    myInt@h sum = buffer +@here k[i] + message[g];
    ...
}

 buffer is at (104,4)
+ is at (204,5)
k[i] is at (205,105)
MD5 in CPart

typedef vector<int> myInt;

vector<myInt>@{[0:64]=\{306,206,106,6\}} k[64];

myInt@{305,205,105,5} sumrotate(myInt@{304,204,104,4} buffer, ...) {
    myInt@h sum = buffer +@here k[i] + message[g];
    ...
}

buffer is at \{304,204,104,4\}
+ k[i] is at \{306,206,106,6\}

k[i] is at \{305,205,105,5\}
MD5 in CPart

typedef pair<int,int> myInt;

vector<myInt>@{[0:64]=(106,6)} k[64];

myInt@(105,5) sumrotate(myInt@(104,4) buffer, ...) {
    myInt@h sum = buffer +@here k[i] + message[g];
    ...
}

MD5 in CPart

typedef pair<int,int> myInt;

vector<myInt>@{[0:64]=(106,6)} k[64];

myInt@(105,5) sumrotate(myInt@(104,4) buffer, ...) {
    myInt@h sum = buffer +@here k[i] +@?? message[g];
    ...
}

+ happens at here which is (105,5)

+ happens at where the synthesizer decides
Summary: Language & Compiler

Language features:
• Specify code and data placement by annotation
• No explicit communication required
• Option to not specifying place

Synthesizer fills in holes such that:
• number of messages is minimized
• code can fit in each core
Project Pipeline

L0: high-level language

Partitioning

L1: L0 + virtual communication channels

Placement & Routing

L2: L1 + layout + routing

Inserting communication

L3: L2 + real communication code

Code generation

L4: machine instructions
Program Synthesis for Code Generation

Synthesizing optimal code

- **Input**: unoptimized code (the spec)
- Search space of all programs

Synthesizing optimal library code

- **Input**: sketch + spec
- Search completions of the sketch

Synthesizing communication code

- **Input**: program with virtual channels
- Insert actual communication code
1) Synthesizing optimal code

unoptimized code (spec)

synthesizer

optimal code
Our Experiment

Register-based processor

Stack-based processor

slower

faster

naive

bit trick

hand

optimized

spec

most optimal

synthesizer

hand

naive

bit trick

optimized

spec

most optimal
Our Experiment

Register-based processor

Stack-based processor

naive -> hand -> spec

bit trick

optimized -> faster

most optimal -> slower

synthesizer
Comparison

Register-based processor

Stack-based processor

slower

faster

naive

hand

optimized

translation

spec

most optimal

bit trick

hand

synthesizer
Preliminary Synthesis Times

Synthesizing a program with 8 unknown instructions takes 5 second to 5 minutes.

Synthesizing a program up to ~25 unknown instructions within 50 minutes.
# Preliminary Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>Approx. Speedup</th>
<th>Code length reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>x – (x &amp; y)</td>
<td>Exclude common bits</td>
<td>5.2x</td>
<td>4x</td>
</tr>
<tr>
<td>~(x – y)</td>
<td>Negate difference</td>
<td>2.3x</td>
<td>2x</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
<td>Inclusive or</td>
<td>1.8x</td>
</tr>
<tr>
<td>(x + 7) &amp; -8</td>
<td>Round up to multiple of 8</td>
<td>1.7x</td>
<td>1.8x</td>
</tr>
<tr>
<td>(x &amp; m)</td>
<td>(y &amp; ~m)</td>
<td>Replace x with y where bits of m are 1’s</td>
<td>2x</td>
</tr>
<tr>
<td>(y &amp; m)</td>
<td>(x &amp; ~m)</td>
<td>Replace y with x where bits of m are 1’s</td>
<td>2.6x</td>
</tr>
<tr>
<td>x' = (x &amp; m)</td>
<td>(y &amp; ~m)</td>
<td>Swap x and y where bits of m are 1’s</td>
<td>2x</td>
</tr>
<tr>
<td>y' = (y &amp; m)</td>
<td>(x &amp; ~m)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Code Length

<table>
<thead>
<tr>
<th>Program</th>
<th>Original Length</th>
<th>Output Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>x – (x &amp; y)</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>~ (x – y)</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>x \mid y</td>
<td>27</td>
<td>15</td>
</tr>
<tr>
<td>(x + 7) &amp; -8</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>(x &amp; m) \mid (y &amp; ~m)</td>
<td>22</td>
<td>11</td>
</tr>
<tr>
<td>(y &amp; m) \mid (x &amp; ~m)</td>
<td>21</td>
<td>8</td>
</tr>
<tr>
<td>x’ = (x &amp; m) \mid (y &amp; ~m)</td>
<td>43</td>
<td>21</td>
</tr>
<tr>
<td>y’ = (y &amp; m) \mid (x &amp; ~m)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2) Synthesizing optimal library code

Input:

Sketch: program with holes to be filled
Spec: program in any programing language

Output:

Complete program with filled holes
Example: Integer Division by Constant

Naïve Implementation:
Subtract divisor until reminder < divisor.
# of iterations = output value     Inefficient!

Better Implementation:

\[
\text{quotient} = (M \times n) >> s
\]

- \( n \) - input
- \( M \) - “magic” number
- \( s \) - shifting value

\( M \) and \( s \) depend on the number of bits and constant divisor.
Example: Integer Division by 3

Spec:

\[ \frac{n}{3} \]

Sketch:

\[ \text{quotient} = (?? \times n) \gg ?? \]
## Preliminary Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Solution</th>
<th>Synthesis Time (s)</th>
<th>Verification Time (s)</th>
<th># of Pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>x/3</td>
<td>((43691 \times x) \gg 17)</td>
<td>2.3</td>
<td>7.6</td>
<td>4</td>
</tr>
<tr>
<td>x/5</td>
<td>((209716 \times x) \gg 20)</td>
<td>3</td>
<td>8.6</td>
<td>6</td>
</tr>
<tr>
<td>x/6</td>
<td>((43691 \times x) \gg 18)</td>
<td>3.3</td>
<td>6.6</td>
<td>6</td>
</tr>
<tr>
<td>x/7</td>
<td>((149797 \times x) \gg 20)</td>
<td>2</td>
<td>5.5</td>
<td>3</td>
</tr>
<tr>
<td>deBruijn: (\log_2 x) ((x \text{ is power of 2}))</td>
<td>deBruijn = 46, Table = {7, 0, 1, 3, 6, 2, 5, 4}</td>
<td>3.8</td>
<td>N/A</td>
<td>8</td>
</tr>
</tbody>
</table>

*Note: these programs work for 18-bit number except \(\log_2 x\) is for 8-bit number.*
3) Communication Code for GreenArray

Synthesize communication code between nodes

Interleave communication code with computational code such that

There is no deadlock.
The runtime or energy consumption of the synthesized program is minimized.
The **key** to the success of **low-power computing** lies in inventing better and more effective **programming frameworks**.