Synthesis-Based Compiler for Low-Power Multicore Architectures

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

University of California, Berkeley
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
Future Architectures

What is the future architecture? Convergence point unclear. But it will be some combination of

1. Many small cores (less control overhead, smaller bitwidth)
2. Simple interconnect (reduce communication energy)
3. New ISAs (specialized, more compact encoding)

Future architectures

- Many small cores
- Simple interconnect
- New ISAs

Challenges

- Fine-grained partitioning
- SW-controlled messages
- New compiler optimizations

What we are working on

- Programming model for future multicore architectures
- Synthesis-aided “compiler”
Compilers: State of the Art

Synthesis, an alternative to compilation

- **Compiler**: transforms the source code
- **Synthesis**: searches for a correct, fast program
Program Synthesis (Example)

**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;
}
```

**Synthesized program:**

```c
int[16] trans_sse(int[16] M) implements transpose {
    int[16] S = 0, T = 0;
    repeat (??) S[4::4] = shufps(M[6::4], M[2::4], 11001000b);
    repeat (??) T[12::4] = shufps(S[12::4], S[8::4], 11010111b);
    repeat (??) T[16::4] = shufps(S[16::4], S[12::4], 10110100b);
    return T;
}
```

**Synthesis time < 10 seconds.**

**Search space > 10^{70}**
Our Plan

1. New Programming Model
2. New Approach Using Synthesis

```
Partitioner

Location Mapper

Communication Code Generator

Per-core HLPs with communication code

Per-core optimized machine code
```

HL Program

HLP with partition annotation

HLP with location annotation & routing info

Code Generator
Case study: GreenArrays Spatial Processors

# of Instructions/second vs Power

~100x

Finite Impulse Response Benchmark

GA144 is 11x faster and simultaneously 9x more energy efficient than MSP 430.

Data from Rimas Avizienis

Specs
- Stack-based 18-bit architecture
- 32 instructions
- 8 x 18 array of asynchronous computers (cores)
- No shared resources (i.e. clock, cache, memory). Very scalable architecture.
- Limited communication, neighbors only
- < 300 byte memory per core

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.

Figure from Per Ljung

Data from Rimas Avizienis
MD5 on GA144

High order

102
shift value
R

103
message
M

Low order

002
current hash

003
message
M

004

A
B
C
D

005
rotate &
add with
carry

006
constant
K

010
constant
K

Each of 64 steps

Once per message block
Actual MD5 Implementation on GA144

900 ndef 
offset 
5080 0 ; relocation helpers 32 flh load target data 1254 = high 4 fh load 40 6 = low 6 fh load 40 const and adder and rotator 1100 = high 8 fh load 12 fh load 27 5 = low 10 fh load 12 fh load 16 add 1104 = high 14 fh load 40 4 = low 14 fh load 40 msg 1100 = high 0 org 16 fh load 39 3 = low 0 org 16 fh load 39 rot 1124 = high 18 fh load 38 2 = low 20 fh load 38 enoty point 1140 = high 22 fh load 2A 1 = low 24 fh load 40 outlaw feeder 0 = high 4 fh load 40 100 = high 28 fh load 36 200 = high 36 fh load 36

1000 load descriptor offset 5080 0 ; inward offset 100 1 / and drop if up: then down; outward offset 100 1 / and drop if down: then up; west com 0 1 and drop if right: then left; node n offset + drop +node /ram/ 106 /node west /p 6 /node west /p 105 /node west /p 5 /node west /p 104 /node west /p 4 /node west /p 103 /node west /p 3 /node west /p 102 /node west /p 2 /node west /p 101 /node inward /p 1 /node inward /p 0 /node inward /p 100 /node outward /p 200 /node 2A /p

Break slot addr node offset + break ; / 34 100 break 3 3F 0 break

912 ndef 
offset 
100 1 / and drop if 'n' offset + drop +node /ram/ 106 /node west /p 6 /node west /p 105 /node west /p 5 /node west /p 104 /node west /p 4 /node west /p 103 /node west /p 3 /node west /p 102 /node west /p 2 /node west /p 101 /node inward /p 1 /node inward /p 0 /node inward /p 100 /node outward /p 200 /node 2A /p

1000 load descriptor offset 5080 0 ; inward offset 100 1 / and drop if 'n' offset + drop +node /ram/ 106 /node west /p 6 /node west /p 105 /node west /p 5 /node west /p 104 /node west /p 4 /node west /p 103 /node west /p 3 /node west /p 102 /node west /p 2 /node west /p 101 /node inward /p 1 /node inward /p 0 /node inward /p 100 /node outward /p 200 /node 2A /p

\[...\]
MD5 from Wikipedia

//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 := (3×i + 5) mod 16
    else if 48 ≤ i ≤ 63
      f := c xor (b or (not d))
      g := (7×i) mod 16
    temp := d
    d := c
    c := b
    b := b + leftrotate((a + f + k[i] + w[i]), 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)):
Programming Model

**HL Program**

- **Partitioner**
  - HLP with partition annotation

- **Location Mapper**
  - HLP with location annotation & routing info

- **Communication Code Generator**
  - Per-core HLPs with communication code

- **Code Generator**
  - Per-core optimized machine code
typedef pair<int, int> myInt;

vector<myInt> k[64];

myInt sumrotate(myInt buffer, ...) {
    myInt sum = buffer + k[i] + message[g];
    ...
}

buffer is at (104,4)
+
... is at (105,5)
k[i] is at (106,6)
Partitioning Synthesizer

1. HL Program
2. Partitioner
3. HLP with partition annotation
4. Location Mapper
5. HLP with location annotation & routing info
6. Communication Code Generator
7. Per-core HLPs with communication code
8. Code Generator
9. Per-core optimized machine code
Optimal Partitions from Our Synthesizer

- Benchmark: simplified MD5 (one iteration)
- Partitions are automatically generated.

**256-byte mem per core**
*Initial data placement specified*

<table>
<thead>
<tr>
<th>High</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>⬇️🔗🔗</td>
<td>2 M</td>
</tr>
<tr>
<td>102 M</td>
<td>3 M</td>
</tr>
<tr>
<td>103 M</td>
<td>5 K</td>
</tr>
<tr>
<td>105 K</td>
<td>6 K</td>
</tr>
</tbody>
</table>

**512-byte mem per core**
*Different initial data placement*

<table>
<thead>
<tr>
<th>High</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>⬇️🔗🔗</td>
<td>2 R</td>
</tr>
<tr>
<td>103 M</td>
<td>3 M</td>
</tr>
<tr>
<td>106 K</td>
<td>5 K</td>
</tr>
<tr>
<td>105 K</td>
<td>6 K</td>
</tr>
</tbody>
</table>

**512-byte mem per core**
*Same initial data placement*

<table>
<thead>
<tr>
<th>High</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>⬇️🔗🔗</td>
<td>2 R</td>
</tr>
<tr>
<td>103 M</td>
<td>3 M</td>
</tr>
<tr>
<td>105 K</td>
<td>6 K</td>
</tr>
<tr>
<td>106 K</td>
<td>5 K</td>
</tr>
</tbody>
</table>
How Does Partitioning Synthesizer Work?

Program with **concrete** partition assignment

Interpreter & capacity checker

# of messages passing in **concrete value**

```c
int@1 a;
int@2 b;
b = a;
```

#msgs = 1
How Does Partitioning Synthesizer Work?

program with symbolic partition assignment

interpreter & capacity checker

symbolic execution

equation of symbolic variables
concrete value

# of messages passing in

#msgs = (ite (= sym1 sym2) 0 1))

Solve for values of symbolic variables such that

• # of messages < upperbound
• constraints on capacity hold

Use Rosette (by Emina Torlak) to implement the interpreter and the checker

• The formula is generated automatically.
• Rosette provides Kodkod as a constraint solver.
Partition Location Problem

HL Program

Partitioner

HLP with partition annotation

Location Mapper

HLP with location annotation & routing info

Communication Code Generator

Per-core HLPs with communication code

Code Generator

Per-core optimized machine code

Simulated Annealing
Communication Code

HL Program

Partitioner

HLP with partition annotation

Location Mapper

HLP with location annotation & routing info

Communication Code Generator

Per-core HLPs with communication code

Transformation

Code Generator

Per-core optimized machine code
Matrix Multiplication: HLP

// C = A x B

int[] A[36], B[36], C[36];

for(i from 0 to 6) {
    for(j from 0 to 6) {
        int sum = 0;
        for(k from 0 to 6) {
            sum = sum + A[6*i+k] * B[6*k+j];
        }
        C[6*i+j] = sum;
    }
}

Matrix Mul:plica:on:

HLP
// C = A x B

int[]@1 A[36];
int[]@2 B[36];
int[]@3 C[36];

for(i from 0 to 6) {
    for(j from 0 to 6) {
        int@2 sum = 0;
        for(k from 0 to 6) {
            sum = sum +@2 A[6 *@1 i+ @1 k] * B[6 *@2 k +@2 j];
        }
        C[6 *@3 i +@3 j] = sum;
    }
}
int A[36];
for(int i = 0; i < 6; ++i) {
    for(int j = 0; j < 6; ++j) {
        for(int k = 0; k < 6; ++k) {
            write("right",A[6*i+k]);
        }
    }
}

int B[36];
for(int i = 0; i < 6; ++i) {
    for(int j = 0; j < 6; ++j) {
        int sum = 0;
        for(int k = 0; k < 6; ++k) {
            sum = sum + read("left") * B[6*k+j];
        }
        write("right",sum);
    }
}

int C[36];
for(int i = 0; i < 6; ++i) {
    for(int j = 0; j < 6; ++j) {
        C[6*i+j] = read("left");
    }
}
Matrix Multiplication: Per-core HLPs

```c
int A[36];
for(int i = 0; i < 6; ++i) {
    for(int j = 0; j < 6; ++j) {
        for(int k = 0; k < 6; ++k) {
            write("right",A[6*i+k]);
        }
    }
}

int B[36];
for(int i = 0; i < 6; ++i) {
    for(int j = 0; j < 6; ++j) {
        int sum = 0;
        for(int k = 0; k < 6; ++k) {
            sum = sum + read("left") * B[6*k+j];
        }
        write("right",sum);
    }
}

int C[36];
for(int i = 0; i < 6; ++i) {
    for(int j = 0; j < 6; ++j) {
        C[6*i+j] = read("left");
    }
}
```

```
1
A

2
\[+\]
\[\]
B

3
C
```
Our Plan

1. HL Program
2. Partitioner
3. HLP with partition annotation
4. Location Mapper
5. HLP with location annotation & routing info
6. Communication Code Generator
7. Per-core HLPs with communication code
8. Code Generator
9. Per-core optimized machine code
Retargetable Code Generation

Synthesis-based code translation needs only these:
- define space of programs to search, via code templates
- define semantics of machine instructions, as if writing an interpreter

Example: define exclusive-or for a stack architecture
xor = lambda: push(pop() ^ pop())

Synthesizer can generate code from
- a template with holes as in transpose example --> sketching
- an unconstrained template --> superoptimization
Current prototype synthesizes a program with
- 8 unknown instructions in 2 to 30 seconds
- ~25 unknown instructions within 5 hours

Synthesized functions are
- 1.7x – 5.2x faster and 1.8x – 4x shorter than naïve implementation of simple GreenArrays functions
- 1.1x-1.4x faster and 1.5x shorter than optimized hand-written GreenArrays functions by experts (MD5 App Note)

Synthesize efficient division by constant

<table>
<thead>
<tr>
<th>Program</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>x/3</td>
<td>(43691 * x) &gt;&gt; 17</td>
</tr>
<tr>
<td>x/5</td>
<td>(52429* x) &gt;&gt; 18</td>
</tr>
<tr>
<td>x/6</td>
<td>(43691 * x) &gt;&gt; 18</td>
</tr>
<tr>
<td>x/7</td>
<td>(149797 * x) &gt;&gt; 20</td>
</tr>
</tbody>
</table>
Future Work

- Make synthesizer retargetable
- Release it!
- Design spatial data structures
- Build low-power gadgets for audio, vision, health, ...

We will answer

“how to build tools for these extreme architectures?”