System on Chip C (SoC-C)
Efficient programming abstractions for heterogeneous multicore Systems on Chip

Alastair Reid
Yuan Lin
Krisztian Flautner
Edmund Grimley-Evans

ARM Ltd
University of Michigan
ARM Ltd
ARM Ltd
Mobile Consumer Electronics Trends

Mobile Application Requirements Still Growing Rapidly
- Still cameras: 2Mpixel → 10 Mpixel
- Video cameras: VGA → HD 1080p → …
- Video players: MPEG-2 → H.264
- 2D Graphics: QVGA → HVGA → VGA → FWVGA → …
- 3D Gaming: > 30Mtriangle/s, antialiasing, …
- Bandwidth: HSDPA (14.4Mbps) → WiMax (70Mbps) → LTE (326Mbps)

Feature Convergence
- Phone
  - + graphics + UI + games
  - + still camera + video camera
  - + music
  - + WiFi + Bluetooth + 3.5G + 3.9G + WiMax + GPS
  - + …
Pocket Supercomputers

The challenge is not processing power
The challenge is energy efficiency
Different Requirements

**Desktop/Laptop/Server**
- 1-10Gop/s
- 10-100W

**Consumer Electronics**
- 10-100Gop/s
- 100mW-1W

10x performance
1/100 power consumption
= 1000x energy efficiency
... leading to Different Hardware

Drop Frequency 10x
- Desktop: 2-4GHz
- Pocket: 200-400MHz

Increase Parallelism 100x
- Desktop: 1-2 cores
- Pocket: 32-way SIMD Instruction Set, 4-8 cores

Match Processor Type to Task
- Desktop: homogeneous, general purpose
- Pocket: heterogeneous, specialised

Keep Memory Local
- Desktop: coherent, shared memory
- Pocket: processor-memory clusters linked by DMA
Example Architecture

- Control Processor
- Distributed Memories
- SIMD Instruction Set
- Data Engines
- Accelerators
- Artist’s impression
What’s wrong with plain C?

C doesn’t provide language features to support

- Multiple processors (or multi-ISA systems)
- Distributed memory
- Multiple threads
Use Indirection (Strawman #1)

Add a layer of indirection

- Operating System
- Layer of middleware
- Device drivers
- Hardware support

All impose a cost in Power/Performance/Area
Raise Pain Threshold (Strawman #2)

Write efficient code at very low level of abstraction

Problems

- Hard, slow and expensive to write, test, debug and maintain
- Design intent drowns in sea of low level detail
- Not portable across different architectures
- Expensive to try different points in design space
Our Response

Extend C

- Support Asymmetric Multiprocessors
- SoC-C language raises level of abstraction
- ... but take care not to hide expensive operations

Use (simple) compiler technology

- Explicit design intent allows error checking
- High-level compiler optimizations
- Compiler takes care of low-level details
Overview

Pocket-Sized Supercomputers

- Energy efficient hardware is “lumpy”
- … and unsupported by C
- … but supported by SoC-C

How SoC-C tackles the underlying hardware issues

- Using SoC-C
- Compiling SoC-C

Conclusion
3 steps in mapping an application

1. Decide how to parallelize
2. Choose processors for each pipeline stage
3. Resolve distributed memory issues
A Simple Program

int x[100];
int y[100];
int z[100];
while (1) {
    get(x);
    foo(y,x);
    bar(z,y);
    baz(z);
    put(z);
}
Step 1: Decide how to parallelize

```c
int x[100];
int y[100];
int z[100];
while (1) {
    get(x);
    foo(y,x);
    bar(z,y);
    baz(z);
    put(z);
}
```

50% of work
**Step 1: Decide how to parallelize**

```c
int x[100];
int y[100];
int z[100];
PIPEDLINE {  
    while (1) {
        get(x);
        foo(y,x);
        FIFO(y);
        bar(z,y);
        baz(z);
        put(z);
    }
}
```
SoC-C Feature #1: Pipeline Parallelism

Annotations express coarse-grained pipeline parallelism

- PIPELINE indicates scope of parallelism
- FIFO indicates boundaries between pipeline stages

Compiler splits into threads communicating through FIFOs

- Uses IN/OUT annotations on functions for dataflow analysis

FIFO

- passes ownership of data
- does not copy data
Step 2: Choose Processors

```c
int x[100];
int y[100];
int z[100];
PIPELINE {
    while (1) {
        get(x);
        foo(y,x);
        FIFO(y);
        bar(z,y);
        baz(z);
        put(z);
    }
}
```
Step 2: Choose Processors

```c
int x[100];
int y[100];
int z[100];
PIPELINE {
    while (1) {
        get(x);
        foo(y,x) @ P0;
        FIFO(y);
        bar(z,y) @ P1;
        baz(z) @ P1;
        put(z);
    }
}
```

@ P indicates processor to execute function
SoC-C Feature #2: RPC Annotations

Annotations express where code is to execute

- Behaves like Synchronous Remote Procedure Call
  - Migrating thread model
  - Does not change meaning of program
- Bulk data is not implicitly copied to processor’s local memory
int x[100];
int y[100];
int z[100];

PIPELINE {
  while (1) {
    get(x);
    foo(y,x) @ P0;
    FIFO(y);
    bar(z,y) @ P1;
    baz(z) @ P1;
    put(z);
  }
}

P0 uses x → x must be in M0
P1 uses z → z must be in M1
P0 uses y → y must be in M0
P1 uses y → y must be in M1

Conflict?!
Hardware Cache Coherency

write x

read x

write x
Step 3: Resolve Memory Issues

int x[100];
int y[100];
int z[100];

PIPELINE {
    while (1) {
        get(x);
        foo(y,x) @ P0;
        \textcolor{blue}{SYNC(x) @ DMA;}
        FIFO(y);
        bar(z,y) @ P1;
        baz(z) @ P1;
        put(z);
    }
}

y has two coherent versions. One in M0, one in M1

\textcolor{blue}{SYNC(x) @ P} copies data from one version of x to another using processor P
SoC-C Feature #3: Compile Time Coherency

Variables can have multiple coherent versions
- Compiler uses memory topology to determine which version is being accessed

Compiler applies cache coherency protocol
- Writing to a version makes it valid and other versions invalid
- Dataflow analysis propagates validity
- Reading from an invalid version is an error
- SYNC(x) copies from valid version to invalid version
What SoC-C Provides

SoC-C language features
- Pipeline to support parallelism
- Coherence to support distributed memory
- RPC to support multiple processors/ISAs

Non-features
- Does not choose boundary between pipeline stages
- Does not resolve coherence problems
- Does not allocate processors

SoC-C is concise notation to express mapping decisions (not a tool for making them on your behalf)
Compiling SoC-C

1. Data Placement
   a) Infer data placement
   b) Propagate coherence
   c) Split variables with multiple placement

2. Pipeline Parallelism
   a) Identify maximal threads
   b) Split into multiple threads
   c) Apply zero copy optimization

3. RPC (see paper for details)
Step 1a: Infer Data Placement

```c
int x[100];
int y[100];
int z[100];
PIPELINE {
  while (1) {
    get(x);
    foo(y,x) @ P0;
    SYNC(x) @ DMA;
    FIFO(y);
    bar(z,y) @ P1;
    baz(z) @ P1;
    put(z);
  }
}
```

- Memory Topology constrains where variables could live
Step 1a: Infer Data Placement

int x[100] @ {M0};
int y[100] @ {M0,M1};
int z[100] @ {M1};
PIPELINE {
    while (1) {
        get(x@?);
        foo(y@M0, x@M0) @ P0;
        SYNC(y,?,?) @ DMA;
        FIFO(y@?);
        bar(z@M1, y@M1) @ P1;
        baz(z@M1) @ P1;
        put(z@?);
    }
}
Step 1b: Propagate Coherence

int x[100] @ {M0};
int y[100] @ {M0,M1};
int z[100] @ {M1};
PIPELINE {
  while (1) {
    get(x@?);
    foo(y@M0, x@M0) @ P0;
    SYNC(y,?,?) @ DMA;
    FIFO(y@?);
    bar(z@M1, y@M1) @ P1;
    baz(z@M1) @ P1;
    put(z@?);
  }
}

- Memory Topology constrains where variables could live
- Forwards Dataflow propagates availability of valid versions
Step 1b: Propagate Coherence

int x[100] @ {M0};
int y[100] @ {M0,M1};
int z[100] @ {M1};
PIPELINE {
  while (1) {
    get(x@?);
    foo(y@M0, x@M0) @ P0;
    SYNC(y,?,M0) @ DMA;
    FIFO(y@?);
    bar(z@M1, y@M1) @ P1;
    baz(z@M1) @ P1;
    put(z@M1);
  }
}

- Memory Topology constrains where variables could live
- Forwards Dataflow propagates availability of valid versions
Step 1b: Propagate Coherence

```
int x[100] @ {M0};
int y[100] @ {M0,M1};
int z[100] @ {M1};
PIPELINE {
    while (1) {
        get(x@?);
        foo(y@M0, x@M0) @ P0;
        SYNC(y,?,M0) @ DMA;
        FIFO(y@?);
        bar(z@M1, y@M1) @ P1;
        baz(z@M1) @ P1;
        put(z@M1);
    }
}
```

- Memory Topology constrains where variables could live
- Forwards Dataflow propagates availability of valid versions
- Backwards Dataflow propagates need for valid versions
Step 1b: Propagate Coherence

int x[100] @ {M0};
int y[100] @ {M0,M1};
int z[100] @ {M1};
PIPELINE {
  while (1) {
    get(x@M0);
    foo(y@M0, x@M0) @ P0;
    SYNC(y,M1,M0) @ DMA;
    FIFO(y@M1);
    bar(z@M1, y@M1) @ P1;
    baz(z@M1) @ P1;
    put(z@M1);
  }
}

- Memory Topology constrains where variables could live
- Forwards Dataflow propagates availability of valid versions
- Backwards Dataflow propagates need for valid versions

(Can use unification+constraints instead)
Step 1c: Split Variables

int x[100] @ {M0};
int y0[100] @ {M0};
int y1[100] @ {M1};
int z[100] @ {M1};
PIPELINE {
    while (1) {
        get(x);
        foo(y0, x) @ P0;
        memcpy(y1,y0,...) @ DMA;
        FIFO(y1);
        bar(z, y1) @ P1;
        baz(z) @ P1;
        put(z);
    }
}
Step 2: Implement Pipeline Annotation

int x[100] @ {M0};
int y0[100] @ {M0};
int y1[100] @ {M1};
int z[100] @ {M1};
PIPELINE {
  while (1) {
    get(x);
    foo(y0, x) @ P0;
    memcpy(y1, y0, ...) @ DMA;
    FIFO(y1);
    bar(z, y1) @ P1;
    baz(z) @ P1;
    put(z);
  }
}
Step 2a: Identify Dependent Operations

```
int x[100] @ {M0};
int y0[100] @ {M0};
int y1[100] @ {M1};
int z[100] @ {M1};
PIPELINE {
    while (1) {
        get(x);
        foo(y0, x) @ P0;
        memcpy(y1,y0,...) @ DMA;
        FIFO(y1);
        bar(z, y1) @ P1;
        baz(z) @ P1;
        put(z);
    }
}
```

Dependency Analysis
Split use-def chains at FIFOs
Step 2b: Identify Maximal Threads

int x[100] @ \{M0\};
int y0[100] @ \{M0\};
int y1[100] @ \{M1\};
int z[100] @ \{M1\};

PIPELINE {
    while (1) {
        get(x);
        foo(y0, x) @ P0;
        memcpy(y1, y0, …) @ DMA;
        FIFO(y1);
        bar(z, y1) @ P1;
        baz(z) @ P1;
        put(z);
    }
}

Dependency Analysis
Split use-def chains at FIFOs
Identify Thread Operations
Step 2b: Split Into Multiple Threads

```
int x[100] @ {M0};
int y0[100] @ {M0};
int y1a[100] @ {M1};
int y1b[100] @ {M1};
int z[100] @ {M1};
PARALLEL {
  SECTION {
    while (1) {
      get(x);
      foo(y0, x) @ P0;
      memcpy(y1a, y0, ...) @ DMA;
      fifo_put(&f, y1a);
    }
  }
  SECTION {
    while (1) {
      fifo_get(&f, y1b);
      bar(z, y1b) @ P1;
      baz(z) @ P1;
      put(z);
    }
  }
}
```

Perform Dataflow Analysis
Split use-def chains at FIFOs
Identify Thread Operations
Split into threads
Step 2c: Zero Copy Optimization

```c
int x[100] @ {M0};
int y0[100] @ {M0};
int y1a[100] @ {M1};
int y1b[100] @ {M1};
int z[100] @ {M1};
PARALLEL {
    SECTION {
        while (1) {
            get(x);
            foo(y0, x) @ P0;
            memcpy(y1a, y0, ...) @ DMA;
            fifo_put(&f, y1a);
        }
    }
    SECTION {
        while (1) {
            fifo_get(&f, y1b);
            bar(z, y1b) @ P1;
            baz(z) @ P1;
            put(z);
        }
    }
}
```

Generate Data
Copy into FIFO

Copy out of FIFO
Consume Data
Step 2c: Zero Copy Optimization

Calculate Live Range of variables passed through FIFOs

int x[100] @ {M0};
int y0[100] @ {M0};
int y1a[100] @ {M1};
int y1b[100] @ {M1};
int z[100] @ {M1};
PARALLEL {
  SECTION {
    while (1) {
      get(x);
      foo(y0, x) @ P0;
      memcpy(y1a,y0,...) @ DMA;
      fifo_put(&f, y1a);
    }
  }
  SECTION {
    while (1) {
      fifo_get(&f, y1b);
      bar(z, y1b) @ P1;
      baz(z) @ P1;
      put(z);
    }
  }
}
Step 2c: Zero Copy Optimization

int x[100] @ {M0};
int y0[100] @ {M0};
int *py1a;
int *py1b;
int z[100] @ {M1};

PARALLEL {
  SECTION {
    while (1) {
      get(x);
      foo(y0, x) @ P0;
      fifo_acquireRoom(&f, &py1a);
      memcpy(py1a, y0, …) @ DMA;
      fifo_releaseData(&f, py1a);
    }
  }
  SECTION {
    while (1) {
      fifo_acquireData(&f, &py1b);
      bar(z, py1b) @ P1;
      fifo_releaseRoom(&f, py1b);
      baz(z) @ P1;
      put(z);
    }
  }
}

Calculate Live Range of variables passed through FIFOs
Transform FIFO operations to pass pointers instead of copying data

Acquire empty buffer
Generate data directly into buffer
Pass full buffer to thread 2

Acquire full buffer from thread 1
Consume data directly from buffer
Release empty buffer
Order of transformations

Dataflow-sensitive transformations go first
- Inferring data placement
- Coherence checking within threads
- Dependency analysis for parallelism

Parallelism transformations
- Obscures data and control flow

Thread-local optimizations go last
- Zero-copy optimization of FIFO operations
- Continuation passing thread implementation
Related Work

Language
- OpenMP: SMP data parallelism using ‘C plus annotations’
- StreamIt: Pipeline parallelism using dataflow language

Pipeline parallelism
- Multiple independent reinventions

Hardware
  (not cited by paper)
The SoC-C Model

Program as though using SMP system
- Single multithreaded processor: RPCs provide a “Migrating thread Model”
- Single memory: Compiler Managed Coherence handles “bookkeeping”

Use Implicit Parallelism to avoid restructuring code
- Pipeline parallelism
- Data parallelism

Compiler Does Low-Level “Bookkeeping”
- Inter-thread communication → Zero-copy optimization
- Thread programming model → Efficient event-driven execution

Efficiency
- Avoid abstracting expensive operations
- 90-10 rule: lower level interfaces can be mixed with high level abstractions
Language Design Meta Issues

Compiler only uses simple analyses
- Easier to maintain consistency between different compiler versions/implementations

Programmer makes the high-level decisions
- Code and Data Placement
- Inserting SYNC
- Load balancing

Implementation by many source-source transforms
- Programmer can mix high- and low-level features
- 90-10 rule: use high-level features when you can, low-level features when you need to
SoC-C’s Overall Goal

Let Hardware teams design efficient hardware by enabling Software teams to handle resulting complexity.
Step 3a: Resolve Overloaded RPC

```c
int x[100] @ {M0};
int y0[100] @ {M0};
int *py1a;
int *py1b;
int z[100] @ {M1};
PARALLEL {
  SECTION {
    while (1) {
      get(x);
      DE32_foo(0, y0, x);
      fifo_acquireRoom(&f, &py1a);
      DMA_memcpy(py1a, y0, ...);
      fifo_releaseData(&f, py1a);
    }
  }
  SECTION {
    while (1) {
      fifo_acquireData(&f, &py1b);
      DE32_bar(1, z, py1b);
      fifo_releaseRoom(&f, py1b);
      DE32_baz(1, z);
      put(z);
    }
  }
}
```

Replace RPC by architecture specific call:

```
bar(…)@P1 → DE32_bar(1, …)
```
Step 3b: Split RPCs

int x[100] @ {M0};
int y0[100] @ {M0};
int *py1a;
int *py1b;
int z[100] @ {M1};
PARALLEL {
SECTION {
while (1) {
  get(x);
  start_DE32_foo(0, y0, x);
  wait(semaphore_DE32[0]);
  fifo_acquireRoom(&f, &py1a);
  start_DMA_memcpy(py1a, y0, …);
  wait(semaphore_DMA);
  fifo_releaseData(&f, py1a);
}
}
SECTION {
while (1) {
  fifo_acquireData(&f, &py1b);
  start_DE32_bar(1, z, py1b);
  wait(semaphore_DE32[1]);
  fifo_releaseRoom(&f, py1b);
  start_DE32_baz(1, z);
  wait(semaphore_DE32[1]);
  put(z);
}
}
}

RPCs have two phases

- start RPC
- wait for RPC to complete

DE32_foo(0,…);

start_DE32_foo(0,…);

wait(semaphore_DE32[0]);
Two Ways to Exploit Parallelism

Perform twice as much work
- 2 cores can perform 2x more work

Perform same work for less energy
- DVFS (reduce current frequency)
  - halving frequency and doubling #cores saves ~50% energy/op
- Shorter pipeline (reduce peak frequency)
  - halving frequency and doubling #cores saves ~30% energy/op
- Techniques can be combined to give wider range of scaling
- Energy savings requires performance almost linear w/ #cores
Parallel Speedup

**Efficient**
- Same performance as hand-written code

**Near Linear Speedup**
- Very efficient use of parallel hardware

**DVB-T Inner Receiver**
- More realistic OFDM receiver
- 20 tasks, 500-7000 cycles per function, 29000 total
Summary of SoC-C Extensions

Small extensions to C to tackle

1. Multiple processors / Heterogeneity
   - Mapping tasks to engines
   - Event-based programming

2. Distributed memory
   - Coherence
   - Inference

3. Parallelism
   - Pipelining
   - Interthread FIFOs

Raises level of abstraction

Allows compiler to optimize code

No need to restructure code/data as hardware changes
Benefits of SoC-C

Raises level of abstraction
→ Programmer is free to focus on high-level goals
→ Compiler detects coherency errors in programmer annotations
→ Reduce development time and cost

Allows compiler to optimize code
→ Higher level programming with no performance penalty
→ Compiler reduces amount of data copying
→ Compiler generates same code programmer wrote by hand

No need to restructure code/data as hardware changes
→ memory topology
→ number and relative speed of engines
What SoC-C Gives You

**Efficient**
- Compiler generates the same code that a programmer writes
- Neither more nor less efficient than hand-written code
- Doesn’t require programmer with brain the size of a planet

**Allows rapid design space exploration**
- Programmer controls the mapping
- Changing mapping requires small number of changes
- Compiler checks changes for consistency

**Allows rapid porting of code**
- Add annotations, don’t restructure
- Structure of code reflects application, not hardware