Challenges in Correct-by-Construction
Multiprocessor Programming

Albert Cohen
and many more contributors

PARKAS project-team, INRIA and ENS Paris  IRT SystemX

EPFL-INRIA, January 9, 2015
1. Challenge: Great Architectures Need Great Tools

Challenge: Great Architectures Need Great Tools

Challenge: Convergence of Computation and Control

Research Program and Invitation
Challenge: **Great Architectures Need Great Tools**


### MPPA-256

3 Billions transistors, 256 user cores onto a single 28 nm silicon chip

- VLIW Core
  - VLIW processor
  - High power efficiency
  - FPU: 32bits / 64 bits IEEE 754
- Compute Cluster
  - Optimized shared memory / core ratio
  - 17 Low Power cores
  - 2MB of shared memory
- Manycore Processor
  - Two NoCs to connect clusters
  - 34MB of on-chip memory
  - Quad core SMP for I/O management
The Software Stack to the Rescue

[Faraboschi, HiPEAC 2014]
Popular Approach: Domain-Specific Optimization

Hybrid hexagonal/parallelogram tiling for higher dimensional stencils
[Grosser et al., CGO 2014]
Polyhedral Hybrid Tiling for Higher Dimensional Stencils

Experimental Results 2D - $3072^2$

- **Lapacian**
  - ppcg-classical: 32
  - Par4All: 42
  - Overtile: 64
  - ppcg-hybrid: 90

- **Heat**
  - ppcg-classical: 45
  - Par4All: 48
  - Overtile: 62
  - ppcg-hybrid: 135

- **Gradient**
  - ppcg-classical: 58
  - Par4All: 82
  - Overtile: 100
  - ppcg-hybrid: 109

GFLOPS on GTX 470

6 / 39
Selected Polyhedral Tools and Building Blocks

Integer Set Library: http://isl.gforge.inria.fr
LLVM/Polly: http://polly.llvm.org
GCC/Graphite: http://gcc.gnu.org
PPCG: http://ppcg.gforge.inria.fr

Other production compilers: IBM XL, Reservoir Labs R-Stream
Other research tools: Pluto, PPCG, PolyOpt
Other research building blocks: Cloog, Omega+, PolyLib, PIP
Several other research and experimental projects (DSLs, C++ packages)
Beyond Kernels... What About “Real Applications”?  

CARP project  
ARM, CodePlay, RealEyes, Rightware,  
Imperial College, ENS/INRIA,  
RWTH Aachen, U. Twente

- Compiler construction for DSLs: support for multi-level parallelization, loop transformation, mapping to accelerator APIs  
  → Application to VOBLA and SPEAR-DE DSLs

- Language for performance tuning experts  
  → Application to Rodinia, SHOC, commercial application

PENCIL approach

- Domain Specific Languages
- DSL → PENCIL compilers
- PENCIL – Platform Neutral Compute Intermediate Language
- Polyhedral compilation
- Direct OpenCL programming
- OpenCL
- NVIDIA GPUs
- AMD GPUs
- ARM GPUs
- Other accelerators
2. Challenge: Convergence of Computation and Control

Challenge: Great Architectures Need Great Tools

Challenge: Convergence of Computation and Control

Research Program and Invitation
Challenge: Convergence of Computation and Control

Embedded control systems running on complex parallel hardware

Parallelism required, but how to rule out:
- Data races?
- Heisenbugs?
- Relaxed consistency?
- Time unpredictability?
Challenge: *Convergence of Computation and Control*

1. Consolidate multiple safety-critical applications on the same platform
2. + Multiple *time* criticality levels
3. + Non-functional properties: real time, resources, human behavior
4. + Parallel implementation on a shared-memory multicore platform with Alstom Transport
Model-Based Design With Synchronous Languages

Matlab Simulink/StateFlow: mixed continuous/discrete signals, data-flow and automata
Foundations: Kahn Process Networks

Kahn networks, 1974
Gilles Kahn (1946–2006)

Denotational: least fixpoint of a system of equations over continuous functions, for the Scott topology lifted to unbounded streams

\[ s \text{ stream prefix of } s' \implies f(s) \text{ stream prefix of } f(s') \]

Operational: communicating processes over FIFO channels with blocking reads
Foundations: Kahn Process Networks

Kahn networks, 1974
Gilles Kahn (1946–2006)

Denotational: least fixpoint of a system of equations over continuous functions, for the Scott topology lifted to unbounded streams

\[ \text{s stream prefix of } s' \implies f(s) \text{ stream prefix of } f(s') \]

Operational: communicating processes over FIFO channels with blocking reads

→ Deterministic by design
→ Dynamic process creation, parallel composition, reactive systems
→ Effective means to distribute computations
KPN at Work: A Parallel Synchronous Language

Semantics
Asynchronous *futures* to control desynchronization
Same semantics as the sequential program without *async* and ! operators

Expressivness
- Synchronous language: *logical time programming*
- Futures: decouple/explicit the *beginning* and *end* of computations
- Together they allow for *programing parallelism*:
  - producer-consumer decoupling
  - data-parallelism
  - fork-join (instantaneous, and over time steps)
  - task pipelines

Safety
- Statically serializable: futures in a pure language
- No dynamic memory allocation or thread creation
- Proven runtime system (scheduler, FIFO ring buffer)
- Proven compilation flow (part of it)
KPN at Work: The OpenStream Experiment

*Mitigate the von Neumann bottlenecks*

- decoupled, producer/consumer task-parallel pipelines
- fight Amdahl’s law, scavenge parallelism from all sources
- but also preserve local memory, communication bandwidth, consecutive memory accesses
Mitigate the von Neumann bottlenecks

- decoupled, producer/consumer task-parallel pipelines
- fight Amdahl’s law, scavenge parallelism from all sources
- but also preserve local memory, communication bandwidth, consecutive memory accesses
Stream Semantics

```c
#pragma omp task output (x) // Task T1
x = ...;
for (i = 0; i < N; ++i) {
  int window_a[2], window_b[2];

  #pragma omp task output (x << window_a[2]) // Task T2
  window_a[0] = ...; window_a[1] = ...;
  if (i % 2) {
    #pragma omp task input (x >> window_b[2]) // Task T3
    use (window_b[0], window_b[1]);
  }
  #pragma omp task input (x) // Task T4
  use (x);
}
```
Stream Semantics

```c
#pragma omp task output (x) // Task T1
    x = ...;
for (i = 0; i < N; ++i) {
    int window_a[2], window_b[2];

    #pragma omp task output (x << window_a[2]) // Task T2
    window_a[0] = ...; window_a[1] = ...
    if (i % 2) {
        #pragma omp task input (x >> window_b[2]) // Task T3
        use (window_b[0], window_b[1]);
    }
    #pragma omp task input (x) // Task T4
    use (x);
}
```

Interleaving of stream accesses

[Pop and Cohen, TACO 2012], [Drebes et al., TACO 2014]
**Stream Semantics**

```c
#pragma omp task output (x) // Task T1
    x = ...;
for (i = 0; i < N; ++i) {
    int window_a[2], window_b[2];

    #pragma omp task output (x << window_a[2]) // Task T2
        window_a[0] = ...; window_a[1] = ...;
    if (i % 2) {
        #pragma omp task input (x >> window_b[2]) // Task T3
            use (window_b[0], window_b[1]);
    }
    #pragma omp task input (x) // Task T4
        use (x);
}
```

**Interleaving of stream accesses**

[Pop and Cohen, TACO 2012], [Drebes et al., TACO 2014]
Transfer objective: transparent embedding in OpenMP 4

Foster collaborations and technology transfers
U. Manchester, U. of Sienna, Ohio State U., UPMC, BSC, Thales, CAPS Enterprise, Kalray

<table>
<thead>
<tr>
<th>Open source platform</th>
<th>New stream synchronization algorithms</th>
<th>Feed-Forward Data Flow Runtime</th>
<th>Control-Driven Data Flow</th>
<th>Proof techniques for concurrent algorithms for relaxed memory</th>
<th>Benchmarks Manual and Automatic</th>
<th>Profiling visualization and analytics</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC4.9 ~30 kLoC</td>
<td>WeakRB libKPN</td>
<td>Compilation and NUMA extensions (heuristics, Owner Writable Memory)</td>
<td>CDDF</td>
<td></td>
<td>~40 kLoC</td>
<td></td>
</tr>
<tr>
<td>GCC summit '08, '09</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

OpenStream

http://openstream.info
Control-Driven Data Flow

Define the formal semantics of imperative programming languages with dynamic, dependent task creation

- control flow: dynamic construction of task graphs
- data flow: decoupling dependent computations (Kahn)
Control-Driven Data Flow – Results

- **Deadlock classification**
  - insufficiency deadlock: missing producer before a barrier or control program termination
  - functional deadlock: dependence cycle
  - spurious deadlock: deadlock induced by CDDF semantics on dependence enforcement (Kahn prefixes)

- **Conditions on program state allowing to prove**
  - deadlock freedom
  - compile-time serializability
  - functional and deadlock determinism

<table>
<thead>
<tr>
<th>Condition on state $\sigma = (k_e, A_e, A_o)$</th>
<th>Deadlock Freedom properties $\neg D(\sigma)$ $\neg ID(\sigma)$ $\neg FD(\sigma)$ $\neg SD(\sigma)$</th>
<th>Serializability $\neg ID(\sigma)$</th>
<th>Determinism Func$^\text{al}$ &amp; Deadlock</th>
</tr>
</thead>
<tbody>
<tr>
<td>$TC(\sigma) \wedge \forall s, \neg MPMC(s)$</td>
<td>Weak than Kahn continuity</td>
<td>no no yes yes</td>
<td>if $\neg ID(\sigma)$ no yes</td>
</tr>
<tr>
<td>$SCC(H(\sigma)) = \emptyset$</td>
<td>Common case, static over-approx.</td>
<td>no no yes yes</td>
<td>if $\neg ID(\sigma)$ no yes</td>
</tr>
<tr>
<td>$SC(\sigma) \wedge \Omega(k_e) \in \Pi$</td>
<td>Less restrictive than strictness</td>
<td>yes yes yes yes</td>
<td>yes no yes</td>
</tr>
<tr>
<td>$\forall \sigma, SC(\sigma)$</td>
<td>Relaxed strictness</td>
<td>yes yes yes yes</td>
<td>yes yes yes</td>
</tr>
</tbody>
</table>

Weaker than Kahn continuity

Common case, static over-approx.

Less restrictive than strictness

Relaxed strictness
A work-stealing algorithm for relaxed memory models

Two implementations: C11 and ARM inline assembly

Based on state-of-the-art sequentially consistent algorithm (Chase and Lev, 2005)

Proven for the POWER/ARM relaxed memory model (axiomatic POWER model by Mador-Haim et al., 2012). POWER and ARM have the same memory model

[Lè et al., PPoPP 2013]
void __lockfunc __op__lock(locktype__t *lock)
{
    for (;;) {
        preempt_disable();
        if (likely(__raw__op__trylock(lock)
            break;
        preempt_enable();

        if (!((lock)->break_lock)
            (lock)->break_lock = 1;
        while (!__op__can_lock(lock) && (!
            _raw__op__relax(&lock->

        )

    (lock) ->break_lock = 0;
}

int steal(Deque *q) {
    size_t t = load_explicit(&q->top, acquire);
    thread_fence(seq_cst);
    size_t b = load_explicit(&q->bottom, acquire);
    int x = EMPTY;
    if (t < b) {
        /* Non-empty queue. */
        Array *a = load_explicit(&q->array, relaxed);
        x = load_explicit(&a->buffer[t % a->size], relaxed);
        if (!compare_exchange_strong_explicit(&q->top, &t, t + 1, seq_cst, relaxed))
            /* Failed race. */
            return ABORT;
    }
    return x;

Lemma 3. The following properties involving barriers

(i) \(Wx,\rightarrow Wy,\rightarrow Rz,\land Wx,\rightarrow Ry,\rightarrow Rz,\rightarrow Wx,\rightarrow Rz,\rightarrow\)

(ii) \(A.Wx,\rightarrow B.Rx,\rightarrow B.Wy,\rightarrow C.Rx,\rightarrow A.Wx,\rightarrow C.Rx,\rightarrow\)

(iii) Let \(X\) stand for \(A.Wx,\rightarrow B.Rx,\lor (A \sim B).Wx,\) and \(Y\) stand for \(C.Wy,\rightarrow D.Ry,\lor (C \sim D).Wy,\) then the following holds:

\(\neg(X,\rightarrow B.Ry,\rightarrow C.Wy,\land Y,\rightarrow D.Rx,\rightarrow A)\)
void __lockfunc __op__lock(locktype__t *lock)
{
    for (;;) {
        preempt_disable();
        if (likely(_raw__op__trylock(lock)
            break;
        preempt_enable();
        if (!(!lock)->break_lock)

    }
}

int steal
    size_t
thread
    size_t
int x;
    if (t++)
        /* No */
    Array *a = load_explicit(&q->array, relaxed);
    x = load_explicit(&a->buffer[t % a->size], relaxed);
    if (!compare_exchange_strong_explicit(&q->top, &t, t + 1, seq_cst, relaxed))
        /* Failed race. */
        return ABORT;
    return x;

Lemma 3. \textbf{The following properties involving barriers}\n\[\begin{align*}
    (i) \quad & (W_x,_{\text{sync}} \rightarrow W_y,_{\text{pp-sat}} R_z,_{\text{sync}} \land \ W_x,_{\text{sync}} \rightarrow R_y,_{\text{sync}} \\
    \quad & \quad \quad \quad \implies W_x,_{\text{pp-sat}} \rightarrow R_z,_{\text{sync}} \\
    (ii) \quad & (A.W_x,_{\text{sync}} \rightarrow B.R_x,_{\text{pp-sat}} \rightarrow B.W_y,_{\text{pp-sat}} C.R_x,_{\text{sync}} \\
    \quad & \quad \quad \quad \quad \quad \quad \quad \quad \implies C.W_y,_{\text{sync}} \rightarrow A.R_x,_{\text{sync}}
\end{align*}\]
Work Stealing (1)

Each core has an associated double-ended queue (deque)

- New tasks are *pushed* to the core’s deque
- When a task finishes, another is *taken* from the core’s deque

![Diagram](image)
Work Stealing (2)

If a task finishes, and the queue is empty, the core steals from another deque.

Cores alternate between the worker and thief roles.

We focus on the study of a single deque (hereafter, the deque).
Expected Properties

**Legal reads**  Only tasks pushed are taken or stolen

**Uniqueness**  A task pushed into the deque cannot be taken or stolen more than once

**Existence**  Tasks are not lost because of concurrency

**Progress**  Given a finite number of tasks, the deque is eventually emptied
Concurrency Problems

Two thieves attempt to steal the same task
Worker and thief contend for the same task
Steal/Steal Resolution in SC

Proven by Chase and Lev for sequential consistency (SC)
Steal/steal resolution with compare-and-swap (CAS)
Proven by Chase and Lev for sequential consistency (SC)
Steal/steal resolution with compare-and-swap (CAS)

race won: steal x
race lost: return failure

other deques
Take/Steal Resolution in SC

Proven by Chase and Lev for sequential consistency (SC)
Potential take/steal races only if one task left; detected through comparison of indices (t and b)
If one task: worker “self steals” from its own deque with a CAS
Take/Steal Resolution in SC

Proven by Chase and Lev for sequential consistency (SC)
Potential take/steal races only if one task left; detected through comparison of indices (t and b)
If one task: worker “self steals” from its own deque with a CAS

Diagram:
- A task (X) checks if its own deque is CASable
- If CASable, it proceeds
- If not CASable, it checks other deques
- If other deques are CASable, it steals from there
Take/Steal Resolution in SC

Proven by Chase and Lev for sequential consistency (SC)
Potential take/steal races only if one task left; detected through comparison of indices (t and b)
If one task: worker “self steals” from its own deque with a CAS
### Summary of Operations in SC

Table below shows how operations affect indices

<table>
<thead>
<tr>
<th>Operation</th>
<th>Effect on Indices</th>
</tr>
</thead>
<tbody>
<tr>
<td>push</td>
<td>++b</td>
</tr>
<tr>
<td>take (deque size &gt; 1)</td>
<td>--b</td>
</tr>
<tr>
<td>take (deque size = 1)</td>
<td>++t if CAS successful</td>
</tr>
<tr>
<td>steal</td>
<td>++t if CAS successful</td>
</tr>
</tbody>
</table>
A Relaxed Concurrency Problem (1)

1. Let’s assume the deque starts with three tasks

W_x, v denotes a write of the value v to the variable x
Blue bins represent the views of Worker’s deque from each core
A Relaxed Concurrency Problem (1)

1. Let’s assume the deque starts with three tasks
2. Thief 1 steals two tasks; the others don’t know yet

Thief 1
Thief 2
Worker

push
0 1 2 3
Worker’s deque

Wt,1
Wt,2

steal
steal

Wb,3

Thief 1 sees Wt,1
Thief 2 sees Wt,2
Worker sees Wb,3

$W_{x,v}$ denotes a write of the value $v$ to the variable $x$

Blue bins represent the views of Worker’s deque from each core
A Relaxed Concurrency Problem (1)

1. Let’s assume the deque starts with three tasks
2. Thief 1 steals two tasks; the others don’t know yet
3. Thief 2 sees the two steals and attempts to steal $x[2]$

$R_{x,v}$ denotes a read of the value $v$ from the variable $x$

Black arrows represent communications
A Relaxed Concurrency Problem (1)

1. Let’s assume the deque starts with three tasks
2. Thief 1 steals two tasks; the others don’t know yet
3. Thief 2 sees the two steals and attempts to steal $x[2]$
4. Worker sees \textit{only the first steal} and takes $x[2]$

\textbf{Rx, v} denotes a read of the value v from the variable x
Black arrows represent communications
A Relaxed Concurrency Problem (2)

Why does it happen?

Different views of the indices in each core
The state of the deque is *relative* to the core that observes it
(≠ SC where the state of the deque is the same for all)

The worker does not realize that it is taking the last element
(*from its viewpoint, t ≠ b-1*)
Hence no CAS to resolve the conflict
**Sequentially Consistent Ideas (1)**

**Why is it different in SC?**

In SC, all memory events are totally ordered
Transitively so

Operations on the same variable are ordered by *coherency*
Sequentially Consistent Ideas (1)

**Why is it different in SC?**

In SC, all memory events are totally ordered
Transitively so

Thief 1

Thief 2

Worker

```
push
Wt,2
steal
Wt,1
steal
Rt,2 Rb,3
steal
take
Rb,3 Wb,2 Rt,τ
```

Operations on the same variable are ordered by *coherency*
Rb,3 in Thief 2 occurs before Wb,2; otherwise, it would read 2
Why is it different in SC?

In SC, all memory events are totally ordered
Transitively so

Transitively, Worker has already seen $Wt,2$ overwrite $Wt,1$
Hence, $Rt,\tau$ cannot read 1
In SC, a test in one core gives information about other cores: the state is the same for all at any given time.

- Can test for special cases (e.g., single-task queue $t = b-1$)
- Can test for invariants (e.g., well-formed queue $t \leq b$)
- Can use induction on these invariants

SC (all cores) $t \leq b$ implies no take/steal conflict

**Does not hold in a relaxed memory model**
Recall the situation in the previous example

Rb,3 in Thief 2 reads from Wb,3 in Worker
Rt,2 in Thief 2 reads from Wt,2 in Thief 1
Recall the situation in the previous example:

What values of $t$ can we read in Worker?

With relaxed semantics, $\tau$ can be either 1 or 2.
A *sync* memory barrier instruction guarantees that *all the writes that have been observed* by the core issuing the barrier instruction are propagated to all the other cores before the core can continue.

With memory fences, however, Rt,τ cannot read 1. The two fences stall each other until one has finished.
Experiments with work-stealing throughput: *task taken per second*

<table>
<thead>
<tr>
<th></th>
<th>Comb: $b = 1; d = 10^7$</th>
<th>Tree: $b = 3; d = 15$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core i7 (2 threads)</td>
<td>$4.87862 \times 10^8$</td>
<td>$3.60838 \times 10^8$</td>
</tr>
<tr>
<td>Tegra 3 (2 threads)</td>
<td>$5.47223 \times 10^7$</td>
<td>$4.12112 \times 10^7$</td>
</tr>
</tbody>
</table>
Experiments with work-stealing throughput: *task taken per second*

<table>
<thead>
<tr>
<th></th>
<th>Comb: $b = 1; d = 10^7$</th>
<th>Tree: $b = 3; d = 15$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core i7 (2 threads)</td>
<td>$4.87862 \times 10^8$</td>
<td>$3.60838 \times 10^8$</td>
</tr>
<tr>
<td>Tegra 3 (2 threads)</td>
<td>$5.47223 \times 10^7$</td>
<td>$4.12112 \times 10^7$</td>
</tr>
</tbody>
</table>

**Beyond Task Scheduling**

Orchestrating a Kahn network involves *dependent* task scheduling

Much slower: producer-consumer synchronizations accumulate coherence/communication delays on the critical path

$\rightarrow$ *stream processing*
Ongoing Work: *libKPN*

Goal: runtime system for Kahn networks

- Baseline: lightweight scheduling with load balancing
- Enhancement: FIFOs for scalable and efficient dynamic dependence resolution and task scheduling
- Best-of-both-worlds?
  (and making it lock-free whenever it does not hurt performance)

*Batched FIFOs with transparent caching of indices, lightweight task suspension and non-obtrusive wake-up*

This is work in progress... but some early results are available
→ Formally proven algorithm in the axiomatic memory model of C11
→ Open runtime system implementation
Example: Data-Flow Parallelization of LDPC and FMRadio

LDPC encoding (Thales C&S)

FMRadio (StreamIt benchmark) (important role of the compiler)
Example: Data-Flow Parallelization of Cholesky

Cholesky factorization on 12-core Xeon
3. Research Program and Invitation

Challenge: Great Architectures Need Great Tools

Challenge: Convergence of Computation and Control

Research Program and Invitation
Research Program and Invitation

Between the hammers of \textit{correctness, programmability, performance} and the anvil of \textit{cost}

Where \textit{reliability meets cost efficiency}

1. Program, test, verify, compile a \textit{single source code}, serving as
   \begin{itemize}
   \item an abstract model for \textit{static analysis}
   \item an executable model for \textit{simulation}
   \item the source of the actual \textit{sequential/parallel code}
   \end{itemize}

2. Ensure safety and efficiency at \textit{design and compilation-time}

3. Rely on \textit{proven, efficient runtime execution environments}