Protecting Software Investment

- Device sizes are growing
  - Moore’s law:
    2x transistors every 18 months
- Product / design cycles are shrinking
- Riding Moore’s Law: *Reuse*
  - Module reuse is here
  - What about software reuse?

- Need to let *software* survive, automatically scale to next generation device
- Need a stronger model for HW-SW interface, with better parallelism
Outline

♦ Motivation, Streams
♦ SCORE
♦ SCORE for Reconfigurable Systems
♦ Scheduling
♦ Compilation
♦ Summary
Software is Everywhere

Programmable System on a Chip
Why Doesn’t Software Scale?

♦ Is software reusable if have…
  • One more processor?
  • One more accelerator core?
  • Larger FPGA?
  • Larger memory?

♦ Will performance improve?

♦ Usually no – why?
  • Device size exposed to programmer
    ♦ Number of processing elements, memory size, bandwidths
  • Algorithmic decisions locked in early

♦ Need a better model / abstraction
A Lesson from ISA Processors

♦ ISA (*Instruction Set Architecture*)
  decouples SW from HW
  • Hide details from SW: # function units, timing, memory size
  • SW survives to compatible, next generation devices
  • Performance scales with device speed + size
  • Survival for decades—e.g. IBM 360, x86

♦ But an ISA cannot scale forever
  • Latency scales with device size (cycles to cross chip, access mem)
  • Need parallelism to hide latency
    ♦ *ILP:* expensive to extract + exploit (caches, branch pred., etc.)
    ♦ *Data:* (Vector, MMX) limited applicability; MMX not scalable
    ♦ *Thread:* (MP, multi-threaded) IPC expensive; hard to program

*Gluing together conventional processors is insufficient*
What is the Next Abstraction?

♦ Goal: more hardware $\Rightarrow$ better performance, without rewriting / recompiling software

♦ Need to:
  • Abstract device sizes
    ♦ Number of fn units / cores; memory size; memory ports
  • Abstract latencies + bandwidths
  • Support rescheduling / reparallelizing a computation to available resources
  • Handle large, heterogeneous systems
  • Have predictable performance
Streams, Process Networks

♦ Stream = FIFO communication channel
  with blocking read, non-blocking write,
  conceptually unbounded capacity
  • Basic primitive for communication, synchronization
  • Exposed at all levels – application (programming model), architecture

♦ Application = graph of stream connected processes (threads), memories
  • Kahn Process Network, 1974
  • Stream semantics ensure determinism regardless of communication timing,
    thread scheduling, etc. (Kahn continuity)

♦ Architecture = graph of stream connected processors (cores), memories
  • Processor (core) runs one or more processes (threads)
  • Some processes always present, on fixed cores (e.g. off-chip interfaces)
  • Some processes sequenced on programmable cores (e.g. DCT on DSP)
Stream-Aware Scheduling

- Streams expose inter-process dependencies (data flow)
- Streams enable efficient, flexible schedules
  - Efficient: fewer blocked cycles, shorter run time
  - Automatically schedule to available resources
    - Number of processors, memory size, network bandwidth, etc.
  - E.g. Fully spatial, pipelined
  - E.g. Time multiplexed with data batching
    - Amortize cost of context swap over larger data set

![Diagram of stream-aware scheduling]

11/21/02 Eylon Caspi — SOC 2002
Stream Reuse

♦ Persistent streams enable reuse
  • Establish connection once (network route / buffer)
  • Reuse connection while processes loaded
  • Cheap (single cycle) stream access
    ♦ Amortize per-message cost of communication
Outline

♦ Motivation, Streams
♦ SCORE
♦ SCORE for Reconfigurable Systems
♦ Scheduling
♦ Compilation
♦ Summary
Components of a Streaming SoC

- **Graph based compute model**
  - Streams

- **Scheduler**
  - To decide where and when processes run, memory is stored

- **Hardware support**
  - Common network interface for all cores, supports stalling, queueing (e.g. OCP)
  - Stream buffer memory
  - Task swap capability for sequenced cores
  - Controller for core sequencing, DMA, and online parts of scheduler (e.g. microprocessor)
SCORE Compute Model

- **Program** = data flow graph of stream-connected threads
  - Kahn process network (blocking read, non-blocking write)

- **Compute: Thread**
  - Task with local control

- **Communication: Stream**
  - FIFO channel, unbounded buffer capacity, blocking read, non-blocking write

- **Memory: Segment**
  - Memory block with stream interface (e.g. streaming read)

- **Dynamics**:
  - Dynamic local thread behavior → dynamic flow rates
  - Unbounded resource usage: may need stream buffer expansion
  - Dynamic graph allocation

- **Model admits parallelism at multiple levels**: ILP, pipeline, data
Outline

♦ Motivation, Streams
♦ SCORE
♦ SCORE for Reconfigurable Systems
♦ Scheduling
♦ Compilation
♦ Summary
SCORE for Reconfigurable Hardware

♦ SCORE: **Stream Computations Organized for Reconfigurable Execution**

♦ Programmable logic + Programmable Interconnect
  • E.g. Field Programmable Gate Arrays (FPGAs)

♦ Hardware scales by tiling / duplicating
  • High parallelism; spatial data paths
  • Stylized, high bandwidth interconnect

♦ But no abstraction for software survival, to date
  • No binary compatibility
  • No performance scaling
    ♦ Designer targets a specific device, specific resource constraints
Virtual Hardware

- **Compute model has unbounded resources**
  - Programmer does not target a particular device size

- **Paging**
  - “Compute pages” swapped in/out (like virtual memory)
  - Page context = thread (FSM to block on stream access)

- **Efficient virtualization**
  - Amortize reconfiguration cost over an entire input buffer
  - Requires “working sets” of tightly-communicating pages to fit on device
SCORE Reconfigurable Hardware Model

♦ **Paged FPGA**
  - Compute Page (CP)
    - Fixed-size slice of reconfig. hardware (e.g. 512 4-LUTs)
    - Fixed number of I/O ports
    - Stream interface with input queue
  - Configurable Memory Block (CMB)
    - Distributed, on-chip memory (e.g. 2 Mbit)
    - Stream interface with input queue
  - High-level interconnect
    - Circuit switched with valid + back-pressure bits

♦ **Microprocessor**
  - Run-time support + user code
Heterogeneous SCORE

- SCORE extends to other processor types
- Network interface
  - Route traffic to network or buffer
  - Block on empty/full stream access
Efficient Streams on Microprocessor

✶ Stream instructions:  
  * stream_read (reg, idx)
  * stream_write (reg, idx)
Application: JPEG Encode
JPEG Encode Performance Scaling

JPEG Encode Run-Time
Quasi-Static Scheduling

Scheduling Heuristic:
- Exhaustive
- Topological
- Min-cut

Hardware Size (CP-CMB Pairs)

Makespan (M Cycles)
Performance Scaling Observations

♦ Performance scales predictably with added hardware

♦ Time sequencing is efficient
  • Application can run on substantially fewer pages than page threads, with negligible performance loss

♦ Scheduling heuristics work well
  • Scheduling analysis can be cheap
CAD with Streams is Hierarchical

♦ Two level CAD hierarchy
  • (1) inside page—compiler
  • (2) between pages—scheduler

♦ Architecture is locally synchronous, globally asynchronous
  • Traditional timing closure only inside page
  • Gracefully accommodate high page-to-page latency
    ♦ Not a free lunch – still impacts application performance
Outline

♦ Motivation, Streams
♦ SCORE
♦ SCORE for Reconfigurable Systems
♦ Scheduling
♦ Compilation
♦ Summary
Page Scheduling

- Where, when, how long to run each page thread?

- **Time slice model**
  - Reconfigure all pages together

- **Decisions for each time slice:**
  - *Temporal Partitioning*
    - Choose group of pages to run
  - *Resource allocation*
    - Allocate stream buffers to/from non-resident page threads ("stitch buffers")
    - Place and Route resident pages, buffers, user segments
  - *Reconfiguration*
  - "Buffer lock" recovery
    - If deadlock due to insufficient buffer size, then expand buffers
JPEG Encode (improved)

- 11 page threads
- 5 user segments
JPEG Encode: Temporal Partitions

- Assume device has 4 CPs, 16 CMBs
JPEG Encode: Stitch Buffers

♦ Stitch buffer ♦

Diagram showing the process of JPEG encoding with emphasis on stitch buffers.
# JPEG Encode: Resource Assignment

## Time Slice

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>É</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CP 0</strong></td>
<td>DCT</td>
<td>Fan</td>
<td>Bits</td>
<td>É</td>
</tr>
<tr>
<td><strong>CP 1</strong></td>
<td>Tran</td>
<td>Fan</td>
<td>Huff</td>
<td>É</td>
</tr>
<tr>
<td><strong>CP 2</strong></td>
<td>DCT'</td>
<td>ZLE</td>
<td>Mix</td>
<td>É</td>
</tr>
<tr>
<td><strong>CP 3</strong></td>
<td>Zig</td>
<td>Quant</td>
<td>-</td>
<td>É</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>É</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CMB 0</strong></td>
<td>Stitch_1_0</td>
<td>Stitch_1_0</td>
<td>Stitch_3_0</td>
<td>É</td>
</tr>
<tr>
<td><strong>CMB 1</strong></td>
<td>Stitch_0_0</td>
<td>Stitch_2_0</td>
<td>Stitch_2_0</td>
<td>É</td>
</tr>
<tr>
<td><strong>CMB 2</strong></td>
<td>Stitch_0_1</td>
<td>Table 1</td>
<td>-</td>
<td>É</td>
</tr>
<tr>
<td><strong>CMB 3</strong></td>
<td>Stitch_0_2</td>
<td>Table 2</td>
<td>-</td>
<td>É</td>
</tr>
<tr>
<td><strong>É</strong></td>
<td>É</td>
<td>É</td>
<td>É</td>
<td>É</td>
</tr>
</tbody>
</table>
Reconfiguration Control

♦ **Between time slices:**
  - Halt CPs, CMBs
  - Wait for in-flight communication to drain into input queues
  - Save CP, CMB context to CMB memory
  - Compute / look-up next temporal partition
  - Swap CMB contents to/from off-chip memory, if necessary
  - Load CP, CMB context from CMB memory; reconfigure
  - Reconfigure interconnect
  - Start CPs, CMBs
Schedule Binding Time

♦ When to make each scheduling decision?
♦ Dynamic (at run time)
  • High run-time overhead
  • Responsive to application dynamics
♦ Static (at load time / install time)
  • Low run-time overhead
  • Possible wasted cycles due to mis-predicted application dynamics
Dynamic Scheduler

- **Premise:** dynamic rate applications benefit from dynamic page groups, time slice duration
  - *E.g.* compressor / decompressor stages
- **Temporal partitioning:**
  - List schedule, order pages by size of available, buffered input
- **Memory management:**
  - Allocate 1Mb stitch buffers and swap off-chip, as necessary, every time slice
- **Place + Route:**
  - Every time slice
    (several 10’s of pages; possible with HW assist)
- **Result:**
  - Very high run-time overhead for scheduling decisions
Dynamic Scheduler Results

- Scheduler overhead is 36% of total execution time

Dynamic Scheduler Total Execution Time
(Wavelet Encoder)

Ideal System (no sched. ovhd.)  Realistic System
Dynamic Scheduler Overhead

- **Average overhead per time slice** = 127K cycles
  - Scheduler overhead = 124K cycles (avg.)
  - Reconfiguration = 3.5K cycles (avg.)
- **Mandates a large time slice – 250K cycles**
  - A thread may idle for much of it if it blocks or exhausts input

**Dynamic Scheduler Overhead per Timeslice**
(Wavelet Encoder)

![Graph showing overhead versus array size](image)
Quasi-static Scheduler

- **Premise:** reduce run-time overhead by iterating a schedule generated off-line

- **Temporal partitioning:**
  - Multi-way graph partitioning constrained by
    - Precedence (of SCCs)
    - Number of CPs • Number of CMBs
  - 3 partitioning heuristics
    - Exhaustive search for max utilization
    - Min cut • Topological sort

- **Memory management:**
  - Off-line allocate 1Mb stitch buffers and schedule off-chip swaps

- **Place + Route:** • Off-line

- **Quasi-static** if can end a time-slice early

- **Result:** • Low run-time overhead AND better quality
Quasi-Static Scheduler Overhead

- Reduced average overhead per time slice by 7x
  - Scheduler overhead = 14K cycles (avg.)
  - Reconfiguration = 4K cycles (avg.)

Dynamic Scheduler Overhead per Timeslice

Static Scheduler Overhead per Timeslice
Quasi-Static Scheduler Results

- **Reduced total execution time by 4.5x** (not 36%)
  - Because:  
    1. Better partitioning (global view, not greedy)  
    2. Can end time slice early if everything stalls

Wavelet Encoder Total Execution Time

![Graph showing execution time vs array size for Dynamic, Expected Run-time, and Quasi-Static methods.](image-url)
Temporal Partitioning Heuristics

♦ “Exhaustive”
  • Goal: maximize utilization, i.e. non-idle page-cycles
  • How: cluster to avoid rate mis-matches
    ♦ Profile avg. consumption, production rates per firing, for each thread
    ♦ Given a temporal partition (group of pages) + I/O rates, deduce average firing rate (Synchronous Data Flow balance eqns)
    ♦ Firing rate ~ page utilization (% non-idle cycles)
    ♦ Exhaustive search of feasible partitions for max total utilization
  • Tried up to 30 pages (6 hours for small array, minutes for large array)

♦ “Min Cut”
  • FBB Flow-Based, Balanced, multi-way partitioning [Yang+Wong, ACM 1994]
  • CP limit → area constraint
  • CMB limit → I/O cut constraint;
    every user segment has edge to sink to cost CMB in cut

♦ “Topological”
  • Pre-cluster strongly connected components (SCCs)
  • Pre-cluster pairs of clusters if reduce I/O of pair
  • Topological sort, partition at CP limit
Partitioning Heuristics Results

JPEG Encode Run-Time
Quasi-Static Scheduling

Makespan (M Cycles)

Hardware Size (CP-CMB Pairs)

- Exhaustive
- Topological
- Min-cut
Partitioning Heuristics Results (2)

JPEG Decode Run-Time
Quasi-Static Scheduling

Makespan (M Cycles)

Hardware Size (CP-CMB Pairs)

Exhaustive
Topological
Min-cut
Scheduling: Future Work

- Buffer sizing based on stream rates
- Software pipelining of time slices

<table>
<thead>
<tr>
<th>Time slice:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original:</td>
<td>P1</td>
<td>P2</td>
<td>P3</td>
<td>P4</td>
<td>P1</td>
<td>P2</td>
</tr>
<tr>
<td>Software Pipelined Time Slices:</td>
<td>P1</td>
<td>P2</td>
<td>P1</td>
<td>P2</td>
<td>P1</td>
<td>P2</td>
</tr>
</tbody>
</table>

Diagram showing the comparison between the original and software pipelined time slices.
Outline

- Motivation, Streams
- SCORE
- SCORE for Reconfigurable Systems
- Scheduling
- Compilation
- Summary
Programming Model: TDF

- **TDF = intermediate, behavioral language for:**
  - SFSM threads (*Streaming Extended Finite State Machine*)
  - Static graph composition, hierarchical

- **State machine for:**
  - Firing signatures (input guards)
  - Control flow (branching)

- **Firing semantics:**
  - When in state X, wait for X’s inputs, then fire (consume, act)

```c
select (input boolean s,
     input unsigned[8] t,
     input unsigned[8] f,
     output unsigned[8] o )
{
    state S (s) : if (s) goto T;
        else goto F;
    state T (t) : o=t; goto S;
    state F (f) : o=f; goto S;
}
```
The Compilation Problem

Programming Model
- Communicating SFSMs
  - unrestricted size, # IOs, timing

Execution Model
- Communicating page configs
  - fixed size, # IOs, timing
- Paged virtual hardware

Compilation is a resource-binding xform on state machines + data-paths
Impact of Communication Delay

- With virtualization, Inter-page delay is unknown, sensitive to:
  - Placement
  - Interconnect implementation
  - Page schedule
  - Technology – wire delay is growing

- Inter-page feedback is slow
  - Partition to contain FB loops in page
  - Schedule to contain FB loops on device
Latency Sensitive Compilation with Streams

- **Pipeline Extraction**
  - Shrink SFSM by extracting control-independent functions
  - Help timing and page partitioning

- **Page Partitioning – SFSM Decomposition**
  - State clustering for minimum inter-page transitions

- **Page Packing**
  - Reduce area fragmentation
  - Contain streams
Pipeline Extraction

- Hoist uncontrolled, feed-forward data-flow out of FSM
- **Benefits:**
  - Shrink FSM cyclic core
  - Extracted pipeline has more freedom for scheduling + partitioning

**Pipeline Extraction Example:**

```
state foo(x):
  if (x==0) ...
```

```
state foo(xz):
  if (xz) ...
```
Pipeline Extraction – SFSM Area

Area for 47 Operators
(Before Pipeline Extraction)

- JPEG Encode
- JPEG Decode
- MPEG (I)
- MPEG (P)
- Wavelet Encode
- IIR

Operator (sorted by area)
Pipeline Extraction – Extractable Area

Extractable Data-Path Area
for 47 Operators

- JPEG Encode
- JPEG Decode
- MPEG (I)
- MPEG (P)
- Wavelet Encode
- IIR

Area (4-LUTs)

Operator (sorted by data-path area)
Delay-Oriented SFSM Decomposition

- **Indivisible unit:** state (CF+DF)
  - Spatial locality in state logic

- **Cluster states into page-size sub-machines**
  - Inter-page communication for data flow, state flow

- **Sequential delay is in inter-page state transfer**
  - Cluster to maintain local control
  - Cluster to contain state loops

- **Similar to:**
  - VLIW trace scheduling \[\text{[Fisher '81]}\]
  - FSM decomp. for low power \[\text{[Benini/DeMicheli ISCAS '98]}\]
  - VM/cache code placement
  - GarpCC HW/SW partitioning \[\text{[Callahan '00]}\]
Page Packing

♦ Cluster SFSMs + pipelines
  • (1) avoid area fragmentation, (2) contain streams

♦ Contain stream buffers
  • Stream buffer implemented as registers inside page
    is fixed size, may cause deadlock (buffer-lock)
  • Choice 1: if prove stream has bounded buffer, then ok to contain
    (halting problem – dis/provable only for some cases)
  • Choice 2: if cannot prove, use buffer-expandable page-to-page I/O
  • Choice 3: if cannot prove, do not pack
Outline

♦ Motivation, Streams
♦ SCORE
♦ SCORE for Reconfigurable Systems
♦ Scheduling
♦ Compilation
♦ Summary
Summary

♦ SCORE enables software to survive, automatically scale to next-gen device

♦ Stream + Process network abstraction at all levels (application, architecture)

♦ Demonstrated scalable, hybrid reconfigurable architecture for SCORE
  • Applications
  • Compiler
  • Device simulator
  • Programming model
  • Scheduler
  • Architecture

♦ More info on the web
  • http://brass.cs.berkeley.edu(SCORE/)
Device Simulation

♦ Simulator engine
  • Cycle level
  • Behavioral model (single-step) for each page thread, emitted by compiler
  • Simplified timing model:
    • Page-to-page latency = 1 cycle
    • CMB access latency = 1 cycle

♦ Device characteristics
  • FPGA based on HSRA [U.C. Berkeley, FPGA ’99]
    ♦ CP = 512 4-LUTs
    ♦ CMB = 2Mbit DRAM, 64-bit data interface, fully pipelined
    ♦ Area for CP-CMB pair:
      | .25µ: | 12.9 mm2 | 1/9 of PII-450 |
      | .18µ: | 6.7 mm2  | 1/16 of PIII-600 |
      | .13µ: | 3.5 mm2  | 1/42 of P4-2GHz |
  • Page reconfiguration time, from CMB = 5000 cycles
  • Synchronous, 250MHz (but some apps not properly timed for 250)
  • Microprocessor: x86 (PIII)
More Dynamic vs. Static Results

JPEG Encode Run-Time

- Dynamic
- Static
- Quasi-Static

Hardware Size (CP-CMB Pairs) vs. Makespan (M Cycles)
More Dynamic vs. Static Results (2)

JPEG Decode Run-Time

![Graph showing makespan vs. hardware size (CP-CMB Pairs)]

- Dynamic
- Static
- Quasi-Static