Streaming and Dynamic Compilers for High Performance Embedded Computing

Peter Mattson, Jonathan Springer,
Charles Garrett, Richard Lethin

Reservoir Labs, Inc.

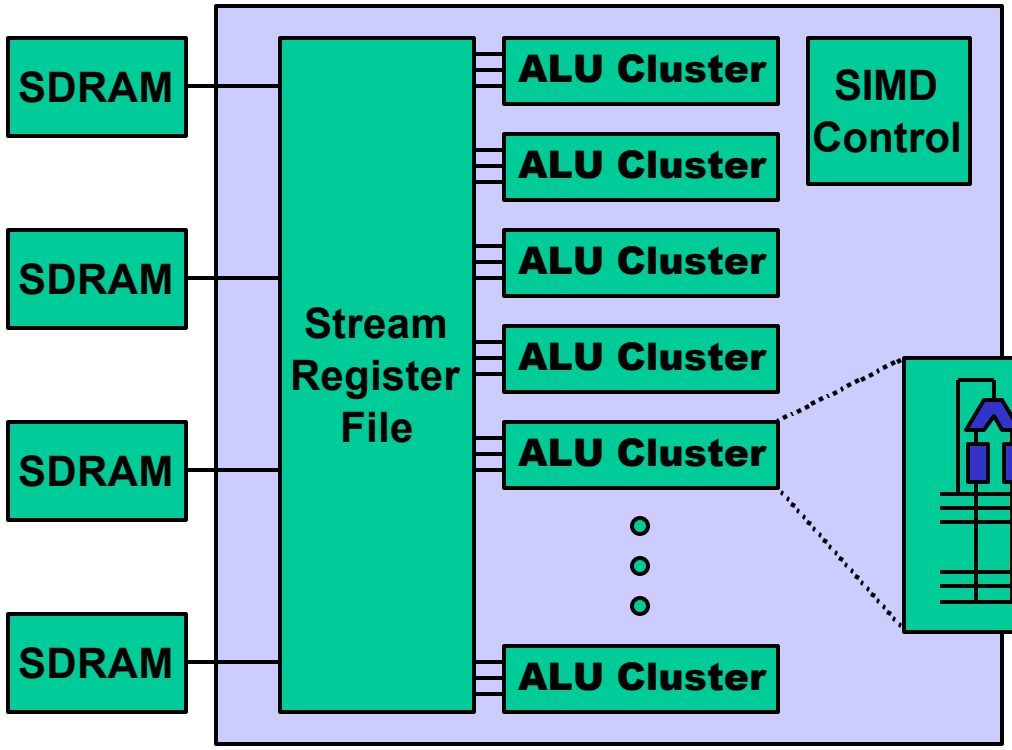
Outline

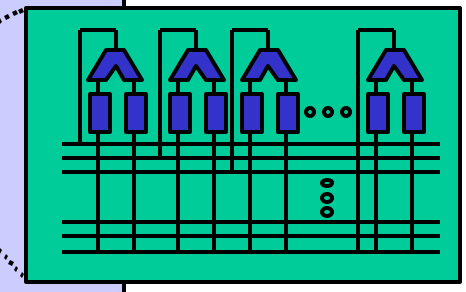
- **Streaming Languages for HPEC / Polymorphic Computer Architectures (PCA).**
  - **Mapping challenges**
  - **R-Stream™ compiler for StreamC and KernelC**
  - **Streaming Language Design Choices**
  - **Thoughts on Mapping**

- **Dynamic Compilation**
  - **PCA objectives that dynamic compilation helps meet**
  - **Runtime Compilation Issues/Technologies**
  - **Example of How Dynamic Compilation Helps with Component Selection**
  - **Insertion of New Code into Running Systems**
  - **Approaches for using Dynamic Compilation Reliably**

Polymorphic (PCA) Architectures

IMAGINE (Stanford: Rixner et. al., 1998)





- High arithmetic/memory ratio
- Parallelism
- Less control logic
- Synchrony
- Programmable
- Exposed resources

Programming and Mapping Challenge

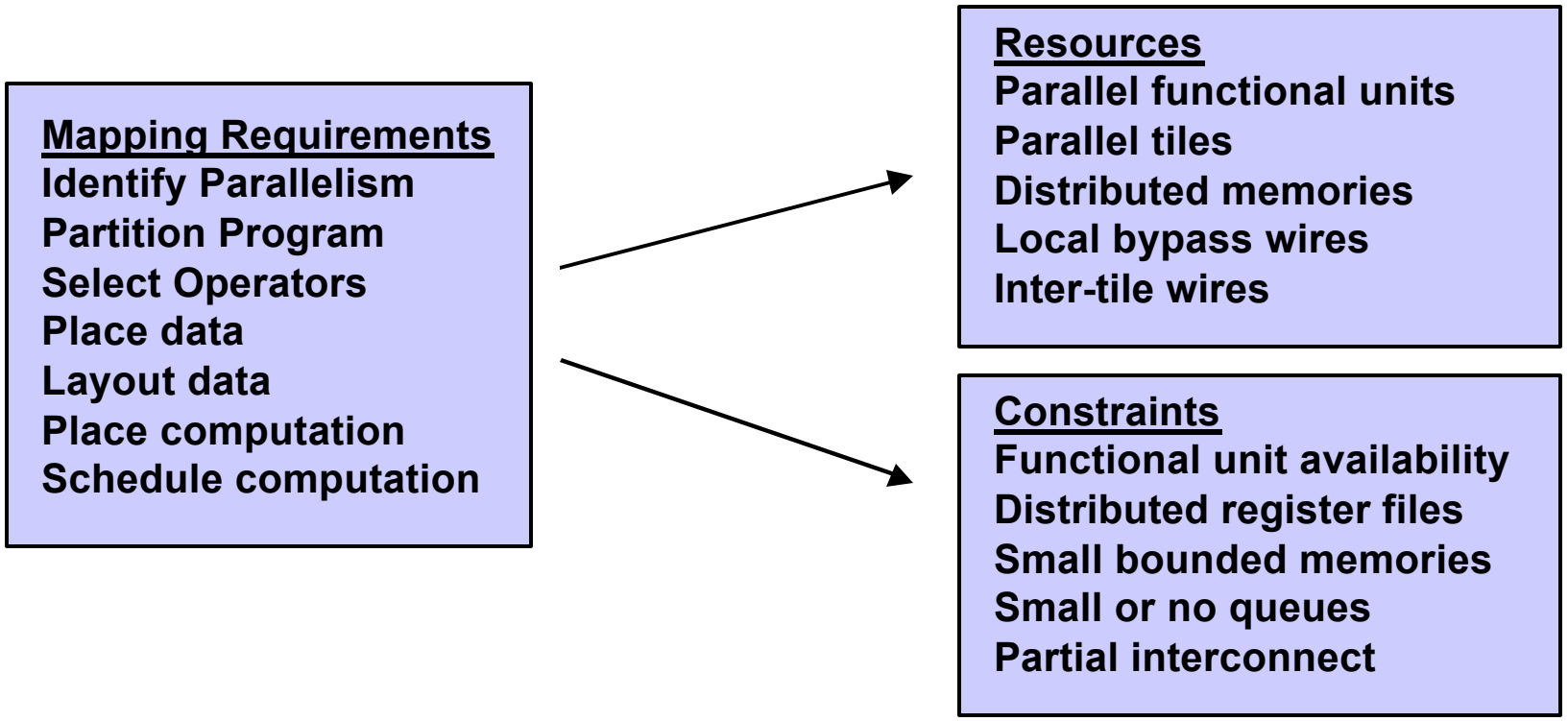
Other Examples: RAW (MIT), VIRAM (Berkeley), Smart Memories (Stanford)

Imagine Performance (Dally et. al 2002)

	Arithmetic Bandwidth	Application Performance
Applications		
Stereo Depth Extraction	11.92 GOPS (16-bit)	320x240 8-bit gray scale
MPEG-2 Encoding	15.35 GOPS (16- and 8-bit)	320x288 24-bit color at 287 fps
QR Decomposition	10.46 GFLOPS	192x92 matrix decompositions in 1.44 ms
Polygon Rendering	5.91 GOPS (fp and integer)	35.6 fps for 720x720 “ADVS” benchmark
Polygon Rendering with Real-Time Shading Language	4.64 GOPS (fp and integer)	16.3M pixels/second 11.1M vertices/second
Kernels		
Discrete Cosine Transformation	22.6 GOPS (16-bit)	34.8 ns per 8x8 block (16-bit)
7x7 Convolution	25.6 GOPS (16-bit)	1.5us per row of 320 16-bit pixels
FFT	6.9 GFLOPs	7.4 us per 1,024-point floating point complex FFT

(6 ALUs * 8 PES * 400 MHz = 19.2 GOPS (peak), 0.18um, 2.56cm^2,)

Compiler Mapping Challenges



Increasing the arithmetic/memory ratio while holding silicon area fixed simultaneously increases resources and tightens constraints.

- ➔ **Automatic compilation MUCH harder**
- ➔ **Place some burden on programmer and invest in compilers**
- ➔ **Introduce new languages to assist programmer**

KernelC / StreamC Languages (Stanford: Mattson 2001)

KernelC Example

```
kernel SAXPY(
   istream<float> x_s,
   istream<float> y_s,
   ostream<float> result_s,
   float a)
{
   loop_stream(x_s) {
      float x, y, result;
      x_s >> x;
      y_s >> y;
      result = a * x + y;
      result_s << result;
   }
}
```

StreamC Example

```
streamprog testSAXPY (String args) {
   const int testSize = 128;
   stream<float> x_s(testSize);
   stream<float> y_s(testSize);
   stream<float> result_s(testSize * 2);
   stream<float> evenResult_s =
      result_s(0, testSize, STRIDE, 2);
   stream<float> oddResult_s =
      result_s(1, testSize, STRIDE, 2);
      // initialize x_s and y_s
      ...
      // compute
      SAXPY(x_s, y_s, evenResult_s, 3.4);
      SAXPY(x_s, y_s, oddResult_s, 6.7);
      ...
}
```

Make High Level Dataflow and Low Level Parallelism Explicit

R-Stream™: Compiling KernelC and StreamC

Compiling KernelC

- Replicate kernel across clusters for SIMD control.
- Generate conditional stream IO operations for data-driven dynamic behavior.
- Modulo-schedule the kernel to get a tight loop.
- Explicitly manage communication resource.

➔ Output is Imagine SIMD control program.

Compiling StreamC

- Inline/flatten program into Kernel call sequence.
- Constant propagate so strip size is explicit.
- Allocate Stream Register File (Local Memory) with a Process like Register Allocation.

➔ Output is C++ program for Imagine controller, that generates stream initiators and synchronization calls.

R-Stream™ can address the mapping challenge posed by these new architectures.

StreamC and KernelC Language: Observations

- Emphasis is on letting programmer express partitioning and parallelism. KernelC is like a new assembly language (e.g., modulo scheduling now moves to the assembler.) The major development challenges for compilation – determining partitioning and mapping - are at the level of the StreamC compiler.

- Nevertheless, there is interaction between StreamC and KernelC compilers which leads to some hand-holding by the programmer of the compiler. Some of this is due to architectural problems (e.g., spilling scratch registers) and some is unavoidable phase interaction. Should they be fused?

- C++ syntax: with special templates and libraries can be compiled by a generic C++ (e.g., Microsoft Visual C++) compiler for code development.

- Expression of conditional stream IO operations allows data-dependent data motion, and maps directly to special hardware on Imagine or can be synthesized with software operations (Kapasi et. al. 2000).

Streaming Language Design Choices

Imperative (e.g. StreamC, Brook [Stanford])

kernel1(…, streamB);
kernel2(streamB, …);
Easy to represent state, control-flow, finite streams
Good for describing array-based algorithms (stride)
Probably map well to conventional DSP

Dataflow (e.g. StreamIt [MIT], Streams-C [LANL])

kernel1.output1.connect(streamB);
kernel2.input1.connect(streamB);
go();
Easy to represent task parallelism, infinite streams
Good for describing continuously-running systems

The history of streaming languages is long.

These programs could be transformed, easily, into FORTRAN

Challenge is how to map!

This gets much harder with dynamic behavior.

There are other things we might need to express, (e.g., flexibility in ordering semantics, periodic invocation,) which arise from nature of embedded systems (Lee 2002)

Thoughts on Mapping

- **Key technologies for compilation for PCA:**
  - **Optimizations that incorporate constraints of small bounded memories.**
  - **Optimizations that incorporate communications bandwidth constraints.**
  - **Optimizations that incorporate non-traditional objectives (e.g., latency, power).**

- **Probably:**
  - **Will see increasing fusion of compiler phases to allow fine-grained tradeoffs between parallelism and storage use.**
  - **More and more, see compilation implemented as search/solvers over constraint systems.**
  - **New program intermediate representations forms, such as Array-SSA (Knobe 1998) or PDG will invigorate development of these key technologies.**

Handling Dynamic Behavior Looks Difficult

Dynamic Behaviors

- Data-dependent iteration counts, strides, access patterns.
- Conditionals within loops.
- Task parallelism.
- External dynamic conditions: mission changes, problem arrival, etc.
- Resource changes (hardware failures).
- Changes to application (e.g., dynamic component replacement).
- Morphing.

Characterize by:

Size of change
Time constant of change
Space of configurations

Use existing technologies and tricks for parallel dynamic load balancing, throttling, mapping, scheduling …
But with the *additional challenge of greater degrees of parallelism and more severe resource constraints.*

If finding a static mapping for PCA hardware is so hard, how can we expect to achieve the same result with an online, distributed, and low-overhead mapper?

Dynamic Compilation: Can it Apply to HPEC?

PCA Technology Program Objective Subset	Dynamic Compilation Capabilities (as demonstrated by R-JIT™ for Java)
Hardware unknown until mission (e.g., failures, other resident apps).	Optimization at mission initiation.
Mission parameters (#targets, etc.) unknown until runtime.	Runtime profiling and profile-driven optimization.
Runtime assembly and selection of library components.	Cross-module optimization at runtime simultaneously with class loading.
Desire for a morphware "virtual machine"	Java virtual machine.
Changing mission parameters, objectives, and system resources at runtime.	Runtime recompilation and insertion of new code into running systems.

Runtime Compilation Issues / Technologies

- How can a compiler be made fast & resource-efficient enough to work effectively at runtime?
  - Efficient data-structures and IR.
  - Integrated optimizations: do multiple mapping and optimization tasks in the same phase, limit phase iterations.
  - Apply optimization selectively to high-priority sections.
  - Sequence from light to heavy optimization based on priority.
- Nevertheless, heavy parallelism and constraint nature of PCA architectures presents a new level of compilation challenge to perform at runtime.
  - Optimizing application might take hours or more (e.g., solving complex constraint system using integer programming).
  - We can start by apply the Java strategies to these new compiler problems
    - E.g., achieve variation in optimization intensity for modulo scheduling by searching first for schedules in looser initiation intervals.
  - New approaches
    - E.g., pre-analyzed or partially compiled code "finished" at runtime.

Dynamic Component Selection: Two Possible Approaches

Component Assemblies

- Collections of precompiled components (different problem sizes, algorithms).
- Decorated with meta-information (XML) or selection control scripts.
- Loader mediates meta-information and constraints to select component.

Object-Oriented Approach (Java/CLOS)

- Component selection is handler selection.
- Loader and compiler tightly integrated.
- Express meta-information within the application in the application language.
- Static conditions at call site inferred from program context.
- Dynamic conditions at call site handled using runtime code generation (e.g., profile-motivated speculation and generation of call-site predicates).
- Expression within programming language allows seamless inter-module analysis and optimization.

The object-oriented and dynamic compilation approach has structured and tested ways of handling this. Advantages in engineering economy, simplicity, and reliability.

Insertion of New Code into Running Systems

- The challenge of dynamic class loading in Java, and how to implement:
  - Class hierarchy in a Java application is not static.
  - For competitive performance, JIT must perform inter-module optimizations using the speculation that class hierarchy is static.
  - The action of a dynamic class load invalidates that speculation.
  - How to transform in-progress invocations (thread queue, call stack?)
    - How do we transfer PC?
    - How do we transform optimized state?
  - One approach (Sun's Hotspot) is dynamic decompilation facility.
    - Hairy, slow, complex semantics. How to validate?
  - Reservoir's approach: pseudo-conditionals and simple code-rewriting.
- These technologies can apply to PCA
  - To "morph" to pre-compiled configurations over small mission sets.
  - To generate "sockets" for morphs to runtime generated configurations for large or open mission sets.

Reliability of Dynamic Compilation

Concerns about reliability are justifiable:

- The complexity of dynamic compiler transformations exceed the complexity of modern dynamic instruction issue hardware.
- In practice, Reservoir has found that other leading commercial release JVMs fail to pass our random Java test suite (R-JVV™).

Reservoir's Approach to Dynamic Compilation Testing

- Intense application of randomly generated tests.
- IR maintenance of extra information to detect optimization failures, with graceful recovery.
- Coverage analysis increases proportion of compiler that is exercised.
- Limit deployment to a single application.

➔ Reservoir's mainframe system R-DYN™ deployment with only one bug found in 4 years (Bank 2001)

Still, much R&D required specifically in compiler validation: e.g., Model-driven testing, compiler specification checking, proof-carrying code.

Summary

- New programmable DSP architectures can achieve performance near fixed function hardware (e.g. Imagine, 20 GOPS).
- ... but introduce major compiler challenges!
- New streaming languages can help.
- Automated mapping for multiple cores/distributed memories is critical research area.
- Advancing automatic mapping technology for dynamic compilation has the potential to solve other important problems.

References

- Bank et. al. 2001 “Dynamic Optimization in the Mainframe World” FDDO-4.
- Dally et. al. 2002: Stanford’s Imagine Web Site: http://cva.stanford.edu/imagine/project/im_perf.html
- Kapasi et. al. 2000 “Efficient Conditional Operations for Data-parallel Architectures” MICRO-33.
- Knobe, Sarkar 1998 “Array SSA form and its use in Parallelization”, SOSP.
- Lee 2002 “Embedded Software” To appear in *Advances in Computers* (M. Zelkowitz, editor), Vol. 56, Academic Press, London.
- Mattson 2001 “A Programming System for the Imagine Media Processor” Stanford Ph.D. Thesis.
- Rixner et. al. 1998 “A Bandwidth-Efficient Architecture for Media Processing” ISCA-31.