A Multi-Paradigm C++-based Hardware Description Language

Chad D. Kersey (cdkersey@gatech.edu)

Advisor: **Sudhakar Yalamanchili**
Acting Advisor: **Hyesoon Kim**
Committee: **Saibal Mukhodpadhyay, Tom Conte, Tushar Krishna, Rich Vuduc, Jeff Young**

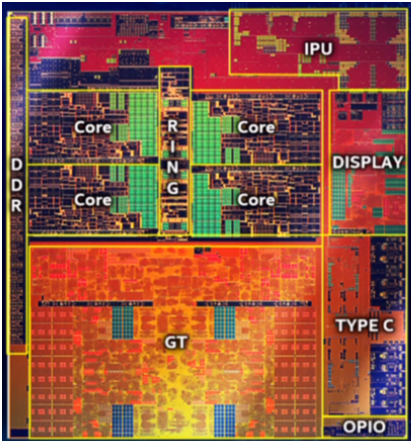
Introduction

Overview

- Hardware description languages
  - Generators
  - Hierarchical Design
  - Register Transfer Level
  - High-Level Synthesis
- All intended to reduce workload for ASIC and FPGA design.
- Also important target for generating, validating, and developing models for system-level simulation.

Overview: Accelerator-Rich Architectures

- Accelerators are an integral part of computer architectures.
- Modern processors incorporate a diverse array of accelerator cores.
- Each accelerator introduces a unique design challenge; these are not simply tiled designs.
- Designer productivity is crucial for achieving performance goals.



10nm Intel Ice Lake core showing significant area devoted to accelerator cores.

Overview: HDL-Based Design

Accelerators pose significant design, verification, and validation task:

- Need to quickly find lower bounds on performance, upper bounds on area and TDP costs.
- High-level synthesis may be well-suited for this initial sanity check.

Using HLS leads to additional challenges:

- Can we use our HLS model as the basis for a full design?
- How do we interface our prototype with models of existing designs?
    - Implement interfaces between our HLS and our existing design?
    - Now we have a new set of interfaces to maintain!
- Best case: our tool supports both HLS and a low-level paradigm. (e.g. SystemC), but what if we want to use a different paradigm?

Overview: Conflicting HDLs

- A design may lend itself well to a third tool, e.g. Bluespec.
- But the majority of the design may already be completed using another HDL.
- With traditional HDLs we would have to add an interface layer.
  - E.g. a Verilog module produced as the output of another tool.
  - Adds one more interface to maintain/keep consistent.
- If our language includes support for generators, however, is it possible to use the generator to implement the required paradigm within the parent language?

Statement of Problem

Popular HDLs do not offer an extensible set of design paradigms and seamless integration between them. Of those that are extensible, none offer a full range of paradigms from gate-level design through HLS.

Background

Background: Extensibility

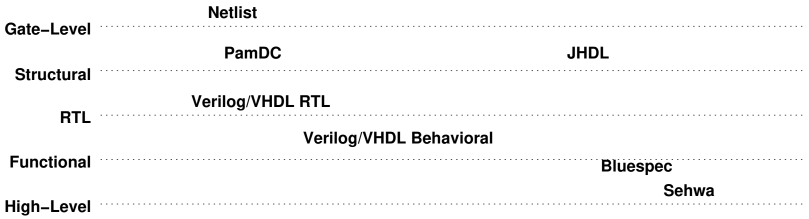
A specific definition of HDL *extensibility* is used in the context of this dissertation:

Criteria for Extensibility

- New hardware description paradigms may be added.
- Interoperability between paradigms.
- Signal types compatible across design paradigms.

- Extensibility is the solution to the problem of interoperability.
- Generative HDLs in high-level languages (MyHDL, Chisel, CHDL) are extensible.

Background: HDL Menagerie



HDLs using many approaches have been developed:

- Traditional HLS approaches do not allow generators; poor interoperability with other paradigms.
- System C: RTL, TLM, and HLS in one; generators supported in elaboration stage; not in synthesizable dialects.
- MyHDL is an extensible Python-based HDL; best described as "SystemC in Python". Extensible because synthesis and simulation environment are the same.
- Chisel is a generative HDL, and has already been extended to support RTL (`when()` blocks) and GAA.

Background: HDL Menagerie

System	**Language**	**Structural**	**RTL**	**Behavioral**	**High-Level**
		Fixed			
PMS/ISP [6]	—	×	×		
Sehwa [46]	Lisp				×
Bambu [48]	C++				×
Gaut [17]	C++				×
Trident [58]	C++				×
LegUp [12]	C++				×
Handel-C [4]	C			×	×
SystemC [44]	C++	×	×	×	×
		Extensible			
PamDC [9]	C++	×			
CHDL(2001) [35]	C++	×	×		
Java Gen. [13]	Java	×			
JHDL [8]	Java		×	×	
Chisel [5]	Scala	×	×		
Extended Chisel [25]	Scala	×	×	×	
Cλash [36]	Haskell	×	×		
Bluespec [42]	SystemVerilog	×	×	×	
MyHDL [18] [28]	Python	×	×	×	
PyMTL [37]	Python	×	×	×	
CHDL	C++	×	×	×	×

Paradigms supported by sampling of HDLs. SystemC provides all paradigm types here, but the set is fixed.

Background: Thesis Statement

Thesis Statement

By adopting a general-purpose language with strong support for construction of domain specific languages, such as C++, as a hardware description language and building a layered set of abstractions around a core of simple primitives, we can produce interoperable designs using a diverse set of paradigms, from gate-level description to high-level synthesis.

Outline of Talk

- Introduction
- Background
- CHDL - The core library, supporting netlist introspection.
- Harmonica - Data parallel core implemented using CHDL.
- CHDL-GAA - Implementation of GAA using CHDL.
- Cheetah - Pipeline-oriented HDL.
- Conclusions

Design of CHDL[1]

[1] C. Kersey and S. Yalamanchili. An Introspective Approach to Architecting Hardware Using C++, OpenSuCo 2017

CHDL: Analogous Structures

CHDL is:

- Generator-based: like PamDC and Chisel.
- Structural: implements all logic as simple primitives.
- Introspective: design can be accessed and modified post-generation.

Analogous Structures

CHDL Structure	**Hardware Structure**
C++ Function	Module
Function Call	Module Instantiation
Program Execution	Elaboration, Simulation

CHDL: Features

CHDL, the core library, provides:

- Data types representing nodes and vectors of signals.
- Functions to instantiate basic logic operations.
- Functions to perform basic integer arithmetic on vectors of signals.
- Operator overloads for logical, bitwise, arithmetic, and comparison operations.
- API for accessing and modifying the netlist of logic primitives.
- Function for dumping the netlist of logic primitives as synthesizable Verilog.
- A set of simple optimizations.
- Technology mapping to standard cell libraries.

CHDL: Features

CHDL-STL, the template library, provides:

- Support for structured signal types.
- Extended support for numeric types including fixed and floating point real numbers.
- Type-independent generators for Bloom filters, queues, and stacks.
- A set of memory interface types and a variety of memory system component generators.
- Implementation of RTL description, including optional `IF`/`ELSE` macros.

CHDL: Flow

CHDL is a Generative HDL:

- All CHDL designs are elaborated down to simple primitives.
- The netlist of primitives is then simulated or emitted.

Use of CHDL:

1. Design is created as C++ program.
2. C++ program is run, building in-memory netlist.
3. Netlist is simulated, emitted as Verilog, or technology mapped.

Use of CHDL

Primitive	**Description**
`Inv()`	Inverter
`Nand()`	2-input nand
`Reg()`	D flip-flop
`Memory()`	SRAM bank

Input:

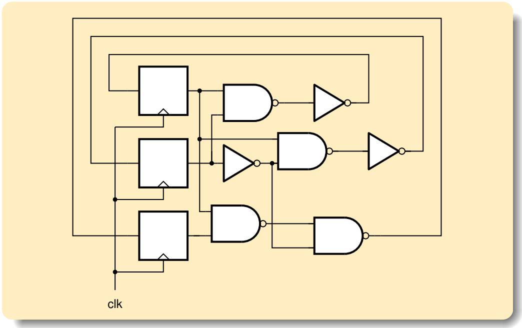
```
bvec<8> x;
x = Reg(x + Lit<8>(1));
```

Output:

- Netlist with 8 DFFs.
- CLA adder optimized to incrementer.

CHDL: Netlist Introspection

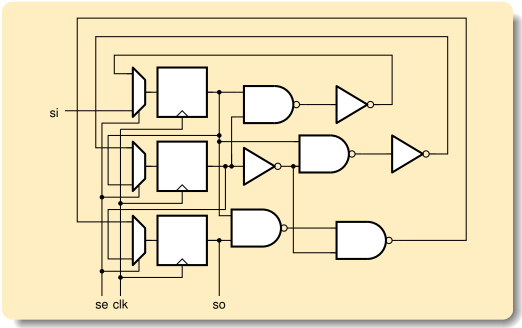
- CHDL provides an API for manipulating the netlist of primitives.
- Has been used to implement novel optimizations:
    - Sub-module caching.
    - Register retiming.
- Also used to implement power emulation and scan chain insertion.



Scan chain insertion and addition of BIST may be performed using netlist introspection.

CHDL: Netlist Introspection

- CHDL provides an API for manipulating the netlist of primitives.
- Has been used to implement novel optimizations:
  - Sub-module caching.
  - Register retiming.
- Also used to implement power emulation and scan chain insertion.

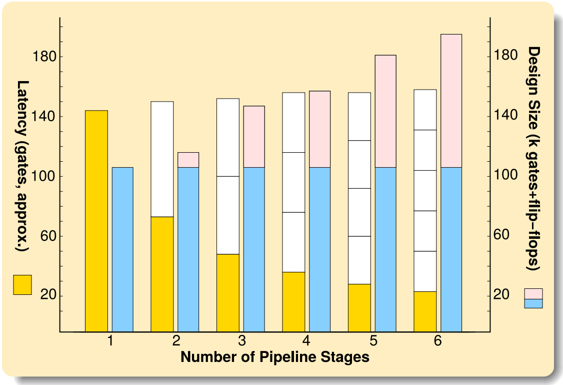


Scan chain insertion and addition of BIST may be performed using netlist introspection.

CHDL: Netlist Introspection

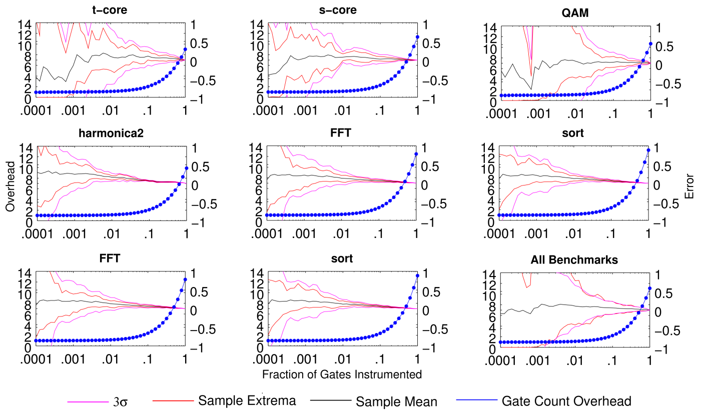
Register retiming, a common optimization, has been implemented using CHDL's netlist introspection:

- Allows addition of pipeline stages by adding empty pipeline stages.
- Selective optimization to avoid retiming debugging signals.
- Independent of built-in CHDL optimizations.
- Can selectively re-timing logic prior to scan.



Logic depth and cell count as a function of number of pipeline stages in a retimed design.

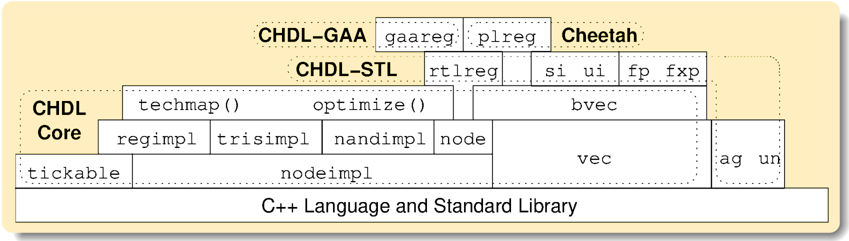
CHDL: Netlist Introspection



Power emulation has also been implemented using CHDL's netlist introspection:

- Uses CHDL technology mapping algorithm.
- Generates global pipelined sum tree (Wallace tree).
- Static sampling to trade accuracy/area.

CHDL: Components



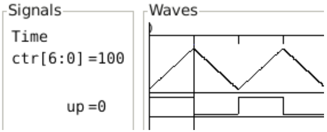
CHDL is composed of multiple component libraries:

- CHDL core library
  - Primitive logic gates, node and vector data types.
  - Logical operator overloads provided for `node`.
  - Arithmetic, bitwise, and comparison operator overloads provided for `bvec<N>`.
  - Optimization, technology mapping, netlist introspection.
- CHDL Template Library
  - Additional arithmetic types and operations.
  - Structured data types.
  - RTL register types and operations.

CHDL: Example

RTL for Alternate Up-Down Counter

```
rtl_reg<node> up(Lit(1));
rtl_reg<bvec<7>> ctr;

IF(up) {
  IF (ctr == Lit<7>(99)) {
    up = Lit(0);
  } ENDIF;
  ctr++;
} ELSE {
  IF (ctr == Lit<7>(1)) {
    up = Lit(1);
  } ENDIF;
  ctr--;
} ENDIF;
```



- Say we want to count by 1 to 100 and back to 0.
- More complicated structures easier to express as RTL.
- CHDL-RTL provided as part of the CHDL template library.
- Optional macros for clarity.

CHDL: Conclusions

In this section we have seen:

- CHDL is a generative C++ based HDL.
- Provides netlist introspection, used to implement:
  - Module caching
  - Retiming
  - Power emulation
- Generator-based paradigm, extended to RTL in CHDL template library.

The Harmonica Core Design[2]

[2] C. Kersey, et al. Lightweight SIMT Core Designs for Intelligent 3D Stacked DRAM, MEMSYS 2017

Harmonica: HARP

Harmonica implements the HARP instruction sets:

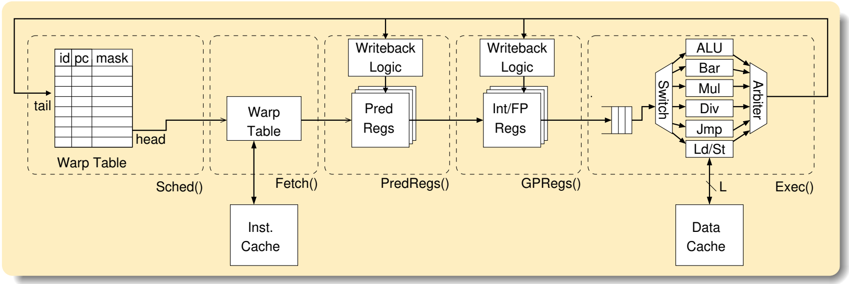
- Project to produce Heterogeneous Architecture Research Prototype.
- Parameterized instruction sets e.g. 4w8/8/32/16:
    - **4**-word instruction and machine word/virtual address.
    - **w**ord-encoded instructions, not byte-encoded.
    - **8** GP and **8** predicate registers per thread.
    - **32** threads per warp and **16** total warps.
- RISC architectures supporting exceptions and hardware interrupts.
- Instructions to control thread/warp spawn.
- Instructions to handle control flow divergence.

Harmonica: Use of CHDL

Harmonica is entirely implemented in CHDL.

- Uses structured signal support from template library.
- RTL-like design style.
- Uses C++ template support to allow parameterization of:
  - Machine word size.
  - Register file size.
  - Number of threads/warps.
- Pipeline registers use CHDL template library `buffer`.

Harmonica: Core Design



Harmonica Stats

Property	Value
Code Size (lines)	2094
Instruction Set	51
Pipeline Depth	6+

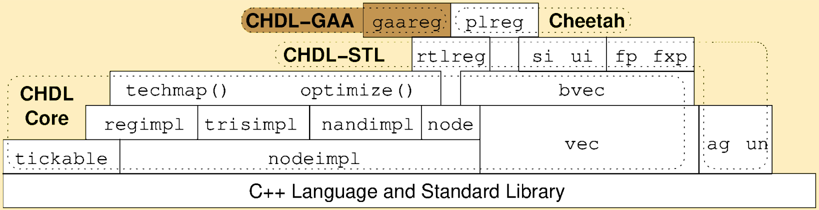
- Small code base and instruction set.
- Organized as one module per major pipe stage.
- Memory system may dominate pipeline latency.

Harmonica: Conclusions

We have seen that Harmonica is:

- A SIMT RISC core.
- Entirely implemented in CHDL.
- A parameterized architecture enabling design space exploration.
- Enabled by CHDL's core and template library features.

Guarded Atomic Actions for CHDL[3]



[3] Planned Submission to DAC 2020

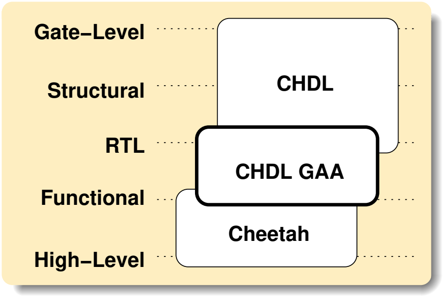
Guarded Atomic Actions

Guarded Atomic Actions:

- GAA allows modules to interact by invoking *method*s instead of asserting a `valid` signal and waiting for a `ready` signal.
- Enables code reuse while maintaining atomicity; method can be invoked from multiple places in requesting module simultaneously.
- Eliminates need for custom arbiter/scheduler implementation for ready/valid signals ($\sim$ 100 lines per module for fair scheduler for arbitrary number of requesters).
- This implementation can be combined with RTL or CHDL generators.

Guarded Atomic Actions

Guarded atomic actions:



GAA sits between Cheetah and the CHDL core and template libraries in terms of level of abstraction.

- Groups of assignments and method invocations organized into *rules*.
- Rule firing also protected by guard predicates.
- Atomicity guaranteed; a rule must fire eventually if its predicate is satisfied.
- Fairness determined by particulars of implementation.
  - CHDL implementation implements a fair scheduler.

Guarded Atomic Actions

Features

Bluespec Feature	**C++/CHDL Feature**
module	class/struct
method	function
Verilog signal types	CHDL signal types
register	`gaareg<T>`
rule	`gaarule`

- Many GAA features mapped to C++/CHDL features by convention.
- Special templated register type; similar to CHDL-RTL.
- Interoperable; `gaareg<T>` holds CHDL signals.
- Rules may be generated algorithmically.
- Explicit `gaa_generate()` function.

Guarded Atomic Actions

- One value method, `Get()`.
- Three action methods:
  - `Set()`
  - `Inc()`
  - `Clear()`
- No explicit guard predicates.

```
struct counter {
  void Set(bvec<8> val) {
    Action().
      Assign(ctr, val);
  }

  void Inc() {
    Action().
      Assign(ctr, ctr + Lit<8>(1));
  }

  bvec<8> Get() { return ctr; }
  void Clear() { Set(Lit<8>(0)); }

  gaareg<bvec<8> > ctr;
};
```

Guarded Atomic Actions: Evaluation

GAA Examples

Description	Lines
Generic GCD; Euclid's Algorithm	31
Project 3D points onto plane	54
N dining philosophers	14
Sieve of Eratosthenes	48

- Examples have short line counts.
  - Rely on CHDL data type/operator implementations.
  - Ready/valid and register write conflict avoidance automated.
  - Fair arbitration between requesters with no additional code.
  - Use of GAA eliminates ~ 100 lines per module.
- Generic; GCD can be done on integers or polynomials in $GF(2^p)$.

Guarded Atomic Actions

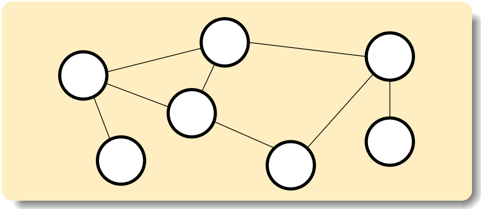
Scheduling in GAA:

- Atomicity provided by eliminating simultaneous writes.
- If conflicting rules fire on same cycle, one must be chosen.
- Static priority scheme is a reasonable option; designer may enforce fairness.

Scheduling in CHDL-GAA:

- Atomicity and fairness both enforced.
- Two algorithms available:
    - Both rotate priorities and provide for fairness.
    - **Dynamic** scheduling algorithm selects all runnable rules.
    - **Static** scheduling algorithm selects runnable rule *sets*, chosen by graph coloring.

Guarded Atomic Actions: Scheduling



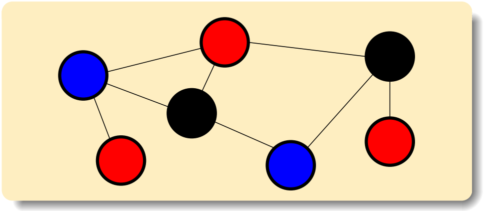
Static scheduling algorithm:

- Construct graph:
    - Rules as nodes.
    - Edges for conflicts.
- Color graph.
- Generate scheduler.
    - Max one color per cycle.
    - Choose based on priority.
    - Rotate priorities for fairness.

Properties of static scheduling:

- Rules are statically assigned to sets.
- Firable set chosen based on priority.
- Trades area vs dynamic scheduler for performance.
- Performance suffers as % of rules firing decreases.

Guarded Atomic Actions: Scheduling



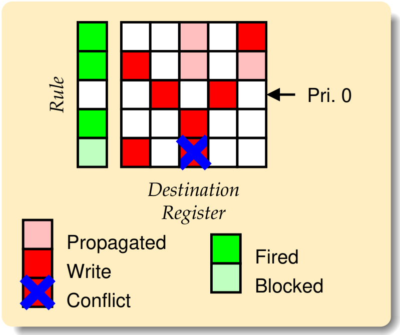
Static scheduling algorithm:

- Construct graph:
  - Rules as nodes.
  - Edges for conflicts.
- Color graph.
- Generate scheduler.
  - Max one color per cycle.
  - Choose based on priority.
  - Rotate priorities for fairness.

Properties of static scheduling:

- Rules are statically assigned to sets.
- Firable set chosen based on priority.
- Trades area vs dynamic scheduler for performance.
- Performance suffers as % of rules firing decreases.

Guarded Atomic Actions: Scheduling



Dynamic scheduler:

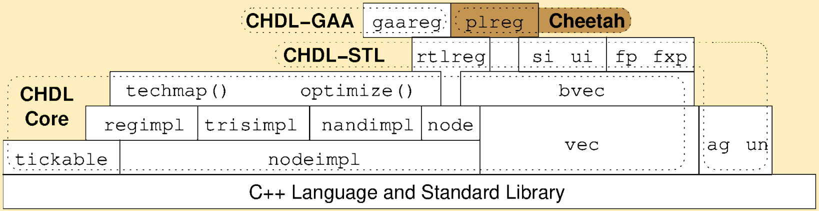
- Matrix of rules and registers.
- Writes propagated in priority order.
- Priority 0 row rotated.

- Trades area and complexity for performance in certain cases.
- Highest-priority rule on cycle *t* is lowest-priority on next cycle.
- Relies on optimizations to produce a high-performance hardware implementation:
  - If no rules write the same register, scheduler should be optimized away.
  - If rules are mutually exclusive, scheduler should be optimized away.

Guarded Atomic Actions: Conclusions

- GAA can be implemented as a combination of generators and new template classes on top of CHDL.
- Steps have to be taken to ensure atomicity and fairness. CHDL-GAA provides two options:
  - Static scheduler; graph coloring based approach.
  - Dynamic scheduler; schedules rules individually.
- GAA enables re-use of code by automating ready/valid signal interfaces.

Cheetah: A Pipeline-Oriented HDL[4]



[4] Planned Submission to DAC 2020

Cheetah: Pipelined Designs

In pipelined designs:

- Signals may have different names as they propagate through.
  - Harmonica spends 56 lines describing inter-stage interfaces.
  - These must be manually updated each time a signal is added.
  - Stages must pass signals they do not use.
- Stage inputs may require arbiters and multiplexers.
- Stall signals may require custom handling.
- Buffers, if added, must be interfaced as well.

Productivity can be realized by automating pipelined designs in the same way that GAA automates interfaces.

Cheetah

Cheetah is a pipeline-oriented HDL:

- Generates pipelines from algorithmic description.
- Basic block in input treated as a pipeline stage.
- Many threads may be active at a time; one per pipeline stage.
- Special signal type `plvar<T>` for pipeline-carried values.
- Relies on CHDL's generator and DSL support.

Feature	**Description**
`PlSpawn()`	Set valid signal for stage; spawn “thread”.
`PlLabel()`	Create a named pipeline stage.
`PlStage()`	Create anonymous pipeline stage.
`PlJmp()`	Conditional jump to named pipeline stage.
`PlBuf()`	*n*-entry pipeline buffer.

Cheetah: Example

Pipelined multiply with FIFO (ready/valid) interface.

- FIFO input to pipeline interface.
- Pipeline stages can be labeled or anonymous.
- `PlStall()` returns stall signal.

```
typedef fp32_t word_t;
const int N = sz<word_t>::value;

plvar<word_t> a, b, p;

PlLabel("start"); {
  word_t in_a, in_b;
  node in_ready = !PlStall();
  OUTPUT(in_ready);
  Flatten(in_a) = Input<N>("in_a");
  Flatten(in_b) = Input<N>("in_b");
  a.set(in_a);
  b.set(in_b);
  PlSpawn(Input("in_valid"));
}
```

Cheetah: Example

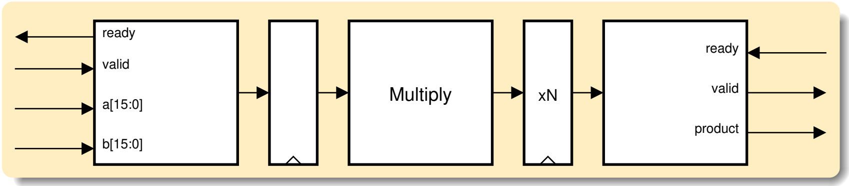
Pipelined multiply with FIFO (ready/valid) interface.

- Additional anonymous stages for retiming.
- Final stage interfaces FIFO output to pipeline.

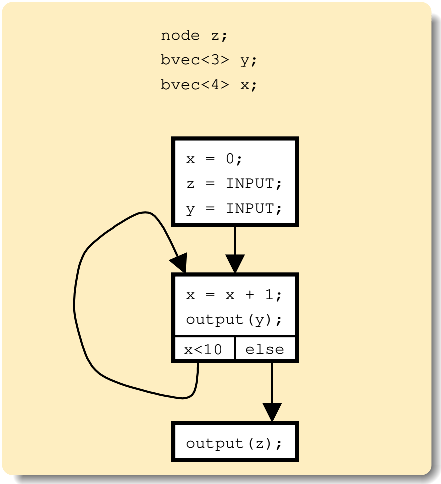
```
const int EX_STG = 10;

PlLabel("mul");
  p.set(a.get() * b.get());

for (int i = 0; i < EX_STG; ++i)
  PlStage();

PlLabel("finish"); {
  bvec<N> out_p = Flatten(p.get());
  node out_valid = PlValid();
  OUTPUT(out_p);
  OUTPUT(out_valid);
  PlStall(Input("out_ready"));
}
```

Pipelined multiply example:

- Uses CHDL-STL for arithmetic functions.
- Most lines devoted to interface.
- Relies on register retiming for performance.
- Pipeline registers automatically inserted.
- Additional buffers may be added with `Buffer()`.
- Simplified diagram excluding stall signals.



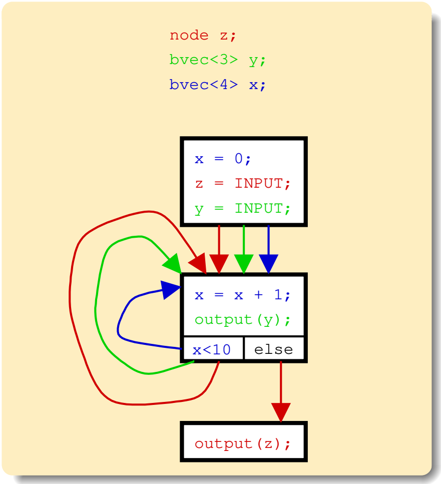
Cheetah: Liveness Analysis



Liveness analysis ensures pipeline registers only generated as necessary.

- Liveness analysis is used for pipeline register/buffer construction.
- Performed at bit granularity. Only live bits are included in pipeline registers.
- All signals in a successor block's live-in will be provided by a predecessor's live-out.
- Note: Inner loop is prioritized to avoid deadlock.

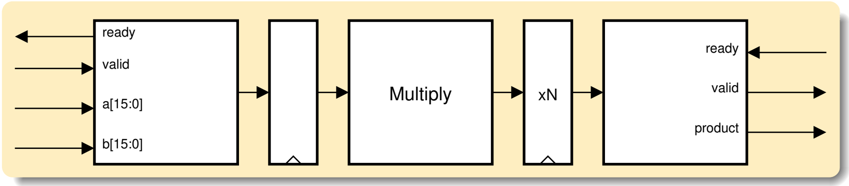
Cheetah: Liveness Analysis



Liveness analysis ensures pipeline registers only generated as necessary.

- Liveness analysis is used for pipeline register/buffer construction.
- Performed at bit granularity. Only live bits are included in pipeline registers.
- All signals in a successor block's live-in will be provided by a predecessor's live-out.
- Note: Inner loop is prioritized to avoid deadlock.

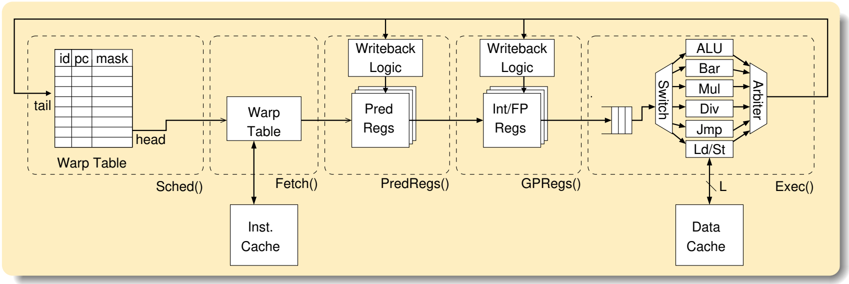
Cheetah



The multiply example contains no cycles, fan-in, or fan-out; a typical design, e.g. Harmonica, does.

- Multiply example contains no conditional branches, cycles.
- Consider design of Harmonica core:
  - Dispatch to multiple functional units.
  - Cycle of warps through system.
- Cheetah automates stalling, steers signals with multiplexers.

Cheetah



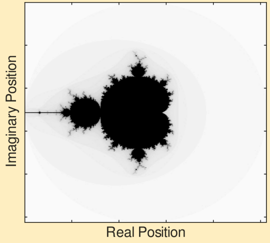
The multiply example contains no cycles, fan-in, or fan-out; a typical design, e.g. Harmonica, does.

- Multiply example contains no conditional branches, cycles.
- Consider design of Harmonica core:
  - Dispatch to multiple functional units.
  - Cycle of warps through system.
- Cheetah automates stalling, steers signals with multiplexers.

Cheetah: Mandelbrot Set

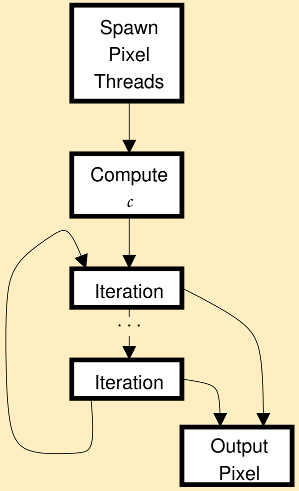
Mandelbrot Set

- Mathematical curiosity with surprisingly complex structure.
- Simple iterative definition. Set of complex numbers c for which $z_0 = 0$, $z_{i+1} = z_i^2 + c$ does not diverge.
- Divergence proven if $|z_i| \geq 2$ Most implementations iteration-limited.



- Mandelbrot set provides example with control flow.
- Each point takes multiple trips through pipeline.
- Pixels are emitted as absolute value exceeds 2 or iteration count exceeded. (i.e. chaotically)

Cheetah: Mandelbrot Set



Pipelined architecture for visualizing Mandelbrot set.

- Templated complex type `cpx<T>`.
- Fixed or floating point, uses CHDL-STL numeric types.
- Multiple iterations may be performed per trip through pipeline. Parameter selects number of iterations.
- Number of iterations and stages per iteration can be set as parameters.
- Spawn loop passes integers to pipeline that computes c.
- Could dispatch to available iteration unit. Matters less for high iteration limits.

Cheetah: Evaluation

Cheetah Examples

Description	**Lines**
Simple single-issue processor	107
Mandelbrot visualizer	58

- Line counts similar to C implementations.
- Complementary to GAA; GAA automates interfaces, Cheetah automates pipelining.
- Instruction set processor example has 10 instruction types; only supports real number (fixed or float) arithmetic.

Cheetah: Conclusions

- A high-level paradigm may be implemented as a DSL in a generative HDL.
- If we treat pipeline stages as analogous to basic blocks, we can use liveness analysis to insert pipeline registers.
- We can still precisely control the hardware implementation in a high-level paradigm like a pipeline-oriented HDL.
- This is effective both for fixed-function hardware and instruction set processors.

Concluding Remarks

Future Directions

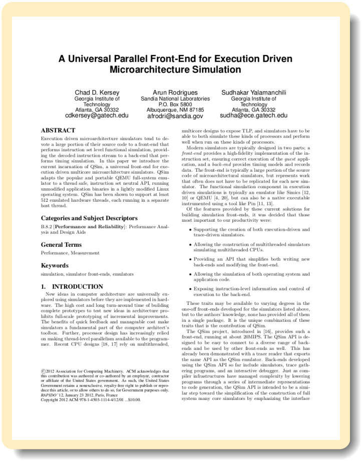
- CHDL is extensible to the point that high-level algorithmic description can be elaborated into gates, but these still don't *look* the same as C++ implementations of the same algorithms.
- "Homoiconic" languages (e.g. Lisp) could support user-transparent HLS; this could be brought to C++ too with a compile-time parsing step.
- A serial complement to Cheetah in which each basic block becomes a clock cycle in a state machine is also in development.
- Combining this with Cheetah could allow explorations of design spaces including both pipelined and multicycle implementations of various pieces.

Engineering Contributions

A wide range of work has been done feeding in to this program of research:

- QSim generic simulation interface and QEMU-based front-end.
- HARP assembly language toolchain and benchmark suite.
- CHDL core implementations including Iqyax (MIPS1-compatible) and Harmonica.
- CHDL/SST integration to the point that multiple Iqyax cores could run with a shared, coherent cache.
- Prototype CHDL/SystemC (simulator) integration.
- CHDL-GAA and Cheetah layers for CHDL.

Most Relevant Publications



A Universal Parallel Front-End for Execution Driven Microarchitecture Simulation

Chad D. Kersey
Georgia Institute of Technology
Atlanta, GA 30332
cdkersey@gatech.edu

Arun Rodrigues
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185
afrodri@sandia.gov

Sudhakar Yalamanchili
Georgia Institute of Technology
Atlanta, GA 30332
sudha@ece.gatech.edu

ABSTRACT

Execution driven microarchitecture simulators tend to devote a large portion of their source code to a front-end that performs instruction set level functional simulation, providing the decoded instruction stream to a back-end that performs timing simulation. In this paper we introduce the current incarnation of QSim, a universal front-end for execution driven multicore microarchitecture simulators. QSim adapts the popular and portable QEMU full-system emulator to a thread safe, instruction set neutral API, running unmodified application binaries in a lightly modified Linux operating system. QSim has been shown to support at least 512 emulated hardware threads, each running in a separate host thread.

Categories and Subject Descriptors

B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids

General Terms

Performance, Measurement

Keywords

simulation, simulator front-ends, emulators

1. INTRODUCTION

New ideas in computer architecture are universally explored using simulators before they are implemented in hardware. The high cost and long turn-around time of building complete prototypes to test new ideas in architecture prohibits full-scale prototyping of incremental improvements. The benefits of quick feedback and manageable cost make simulators a fundamental part of the computer architect's toolbox. Further, processor design has increasingly relied on making thread-level parallelism available to the programmer. Recent CPU designs [18, 17] rely on multithreaded, multicore designs to expose TLP, and simulators have to be able to both simulate these kinds of processors and perform well when run on these kinds of processors.

Modern simulators are typically designed in two parts; a *front-end* provides a high-fidelity implementation of the instruction set, ensuring correct execution of the *guest* application, and a *back-end* provides timing models and records data. The front-end is typically a large portion of the source code of microarchitectural simulators, but represents work that often does not have to be replicated for each new simulator. The functional simulation component in execution driven simulations is typically an emulator like Simics [12, 10] or QEMU [4, 20], but can also be a native executable instrumented using a tool like Pin [11, 13].

Of the features provided by these current solutions for building simulation front-ends, it was decided that those most important to our productivity were:

- Supporting the creation of both execution-driven and trace-driven simulators.
- Allowing the construction of multithreaded simulators simulating multithreaded CPUs.
- Providing an API that simplifies both writing new back-ends and modifying the front-end.
- Allowing the simulation of both operating system and application code.
- Exposing instruction-level information and control of execution to the back-end.

These traits may be available to varying degrees in the one-off front-ends developed for the simulators listed above, but to the authors' knowledge, none has provided all of them in a single package. It is the unique combination of these traits that is the contribution of QSim.

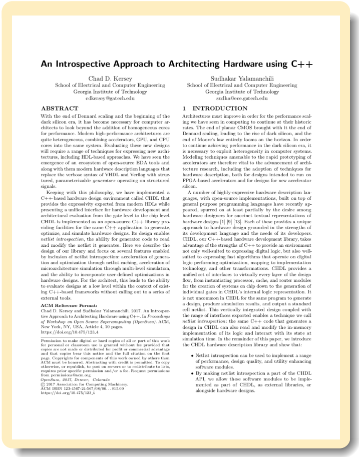
The QSim project, introduced in [16], provides such a front-end, running at about 20MIPS. The QSim API is designed to be easy to connect to a diverse range of back-ends and be used by other front-ends as well. This has already been demonstrated with a trace reader that exports the same API as the QSim emulator. Back-ends developed using the QSim API so far include simulators, trace gathering programs, and an interactive debugger. Just as compiler infrastructures have managed complexity by lowering programs through a series of intermediate representations to code generation, the QSim API is intended to be a similar step toward the simplification of the construction of full system many core simulators by emphasizing the interface

C. Kersey, A. Rodrigues, and S. Yalamanchili

A Universal Parallel Front-end for Execution Driven Microarchitecture Simulation, RAPIDO 2012

- Instruction set independent, ergo universal, API for architectural modeling.
- Provides interface between instruction set and timing simulation.
- Front-end used for high-level Harmonica simulator.

Most Relevant Publications



An Introspective Approach to Architecting Hardware using C++

Chad D. Kersey
School of Electrical and Computer Engineering
Georgia Institute of Technology
cdkersey@gatech.edu

Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology
sudha@ece.gatech.edu

ABSTRACT

With the end of Dennard scaling and the beginning of the dark silicon era, it has become necessary for computer architects to look beyond the addition of homogeneous cores for performance. Modern high-performance architectures are quite heterogeneous, combining accelerators, GPU, and CPU cores into the same system. Evaluating these new designs will require a range of techniques for expressing new architectures, including HDL-based approaches. We have seen the emergence of an ecosystem of open-source EDA tools and along with them modern hardware description languages that replace the verbose syntax of VHDL and Verilog with structured, parameterizable *generators* operating on structured signals.

Keeping with this philosophy, we have implemented a C++-based hardware design environment called CHDL that provides the expressivity expected from modern HDLs while presenting a unified interface for hardware development and architectural evaluation from the gate level to the chip level. CHDL is implemented as an open-source C++ library providing facilities for the same C++ application to generate, optimize, and simulate hardware designs. Its design enables *netlist introspection*, the ability for generator code to read and modify the netlist it generates. Here we describe the design of our library and focus on several features enabled by inclusion of netlist introspection: acceleration of generation and optimization through netlist caching, acceleration of microarchitecture simulation through multi-level simulation, and the ability to incorporate user-defined optimizations in hardware designs. For the architect, this leads to the ability to evaluate designs at a low level within the context of existing C++-based frameworks without calling out to a series of external tools.

ACM Reference Format:
Chad D. Kersey and Sudhakar Yalamanchili. 2017. An Introspective Approach to Architecting Hardware using C++. In *Proceedings of Workshop on Open Source Supercomputing (OpenSuco)*. ACM, New York, NY, USA, Article 4, 10 pages.
https://doi.org/10.475/123_4

1 INTRODUCTION

Architectures must improve in order for the performance scaling we have seen in computing to continue at their historic rates. The end of planar CMOS brought with it the end of Dennard scaling, leading to the rise of dark silicon, and the end of Moore's law entirely looms on the horizon. In order to continue achieving performance in the dark silicon era, it is necessary to exploit heterogeneity in computer systems. Modeling techniques amenable to the rapid prototyping of accelerators are therefore vital to the advancement of architecture research, including the adoption of techniques for hardware description, both for designs intended to run on FPGA-based accelerators and for designs for new accelerator silicon.

A number of highly-expressive hardware description languages, with open-source implementations, built on top of general purpose programming languages have recently appeared, spurred on at least partially by the desire among hardware designers for succinct textual representations of hardware designs [1] [9] [13]. Each of these provides a unique approach to hardware design grounded in the strengths of its development language and the needs of its developers. CHDL, our C++-based hardware development library, takes advantage of the strengths of C++ to provide an environment not only well-suited to expressing digital logic, but also well-suited to expressing fast algorithms that operate on digital logic performing optimization, mapping to implementation technology, and other transformations. CHDL provides a unified set of interfaces to virtually every layer of the design flow, from instantiating processor, cache, and router modules for the creation of systems on chip down to the generation of individual gates in CHDL's internal logic representation. It is not uncommon in CHDL for the same program to generate a design, produce simulation results, and output a standard cell netlist. This vertically integrated design coupled with the range of interfaces exported enables a technique we call *netlist introspection*: the same C++ code that generates a design in CHDL can also read and modify the in-memory implementation of its logic and interact with its state at simulation time. In the remainder of this paper, we introduce the CHDL hardware description library and show that:

- Netlist introspection can be used to implement a range of performance, design quality, and utility enhancing software modules.
- By making netlist introspection a part of the CHDL API, we allow these software modules to be implemented as part of CHDL, as external libraries, or alongside hardware designs.

C. Kersey and S. Yalamanchili

An Introspective Approach to Architecting Hardware Using C++, OpenSuCo 2017

- Introduced the concept of netlist introspection.
- Serves as document of CHDL in general as well.

Most Relevant Publications

Lightweight SIMT Core Designs for Intelligent 3D Stacked DRAM

Chad D. Kersey
Georgia Institute of Technology
School of ECE
cdkersey@gatech.edu

Hyesoon Kim
Georgia Institute of Technology
College of Computing
hyesoon@cc.gatech.edu

Sudhakar Yalamanchili
Georgia Institute of Technology
School of ECE
sudha@ece.gatech.edu

ABSTRACT

In this work we present an analysis of the Harmonica stream multiprocessor, a light-weight, parameterized, open-source single-instruction-multiple-thread (SIMT) core designed for integration within 3D-stacked DRAM. We evaluate the range of Harmonica designs afforded by the architecture's parameter space in the role of a vault-level accelerator, augmenting a design similar to the Micron Hybrid Memory Cube into an array of compact accelerated DRAM channels. In this role, with a small SRAM cache, Harmonica cores are capable of providing the requisite small footprint, energy efficiency, latency tolerance, and bandwidth demand to perform well. The instruction set and microarchitecture of Harmonica are both novel, providing a lightweight interface for thread creation within the SIMT model and a simple design that issues a single warp per cycle, simplifying the register file design compared to high-performance GPUs, and providing parameters for attributes from the number of warps and threads per warp to the number of general purpose registers per thread. For our suite of analytics-oriented benchmarks, Harmonica cores consuming on the order of 100mW of power maintain a demand for an average of 12GB/s of bandwidth while tolerating the latency present in a DRAM-based memory system.

CCS CONCEPTS

• **Computer systems organization → Single instruction, multiple data**;

KEYWORDS

SIMT, near-memory computing, 3D DRAM, accelerator architectures

ACM Reference format:
Chad D. Kersey, Hyesoon Kim, and Sudhakar Yalamanchili. 2017. Lightweight SIMT Core Designs for Intelligent 3D Stacked DRAM. In *Proceedings of 2017 International Symposium on Memory Systems, Alexandria, VA, October 03–05 2017 (MEMSYS'17)*, 11 pages.
https://doi.org/

1 INTRODUCTION

The computing industry is in the midst of a confluence of technology and application trends due to which the cost, execution time, and energy requirements of applications are being dominated by the memory system. With the end of Dennard scaling, the power per socket will remain roughly flat and therefore sustaining performance scaling will require continuing improvements in energy efficiency. In certain HPC applications the cost of data movement is already exceeding the cost of computation [14]. Further, the growth of signal pins on a socket has not kept pace with the number of cores on each die leading to reduced memory bandwidth per core, further challenging performance scaling. These trends have been exacerbated by the explosive emergence of applications in domains such as machine learning, image processing, graph analytics, and query processing characterized by poor temporal and spatial locality, low operation density, irregular control flow, and irregular memory access patterns. As a result, we see increased stresses on memory bandwidth and latency with negative effects on performance. Consequently, we are in seeing a re-emergence of the need for moving processing to memory to reduce data movement energy, mitigate the impact of poor locality, and increase memory bandwidth available to computations.

While research in the 1990s addressed the idea of placing processing in memory (PIM), the continued evolution of Moores Law and related architectural advances and market forces of the day precluded the need for such architectures. Further, the integration of logic in a DRAM process presented its own challenges. However, with the slowing down of Moore's Law, the emergence of 3D packaging provides a vehicle for integrating logic-optimized and DRAM-optimized dice thereby providing a means to sustain performance scaling by pushing back the memory bandwidth and power walls. However, placement of compute accelerators in memory is subject to distinct constraints and tradeoffs between power, performance, and area. Our goal in the design and implementation our SIMT architecture is to develop an architectural template than can be tuned to match different 3D memory organizations.

In this paper we define a family of lightweight single instruction multiple thread (SIMT) architectures referred to as the Heterogeneous Architecture Research Prototype (HARP). The HARP infrastructure defines a family of instruction set architectures parameterized by aspects such as the word size, the number of general-purpose and predicate registers, and the size and number of synchronously executed thread

C. Kersey, H. Kim, and S. Yalamanchili

Lightweight SIMT Core Designs for Intelligent 3D Stacked DRAM, MEMSYS 2017

- Analyzed Harmonica in role of near-memory accelerator.
- Area, power modeling performed using CHDL.

Conclusions

We have seen that:

- HDL-based design does not offer many opportunities for open-ended multi-paradigm design without duplicated design or maintenance effort.
- CHDL provides:
  - A generator-based C++ HDL that is *extensible*.
  - Support for a variety of novel features by allowing netlist introspection, including scan chain insertion.
  - Extended features that include RTL support, GAA, and pipeline-oriented high-level synthesis via Cheetah.
- This thesis contributes specific examples of accelerator and processor designs built using CHDL (Harmonica and Iqyax) and proposes tools and approaches to automate the development of complex designs.

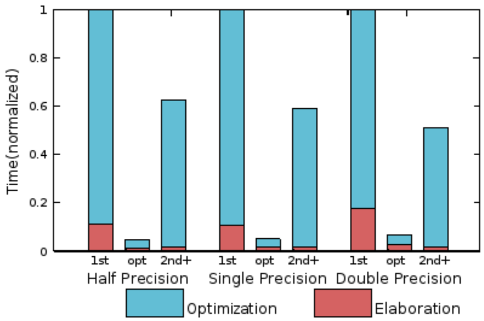
Bonus Slides!

Background: MyHDL

MyHDL is an extensible Python-based HDL:

- Best described as "SystemC in Python".
  - Especially considering SystemC is Verilog in C++.
- Because it is Python, better support for, e.g., reflection.
- Academic work (Jaic et al. 2015) brought support for structured signals.
- May dump fully-elaborated design as synthesizable VHDL or Verilog.
- No support for HLS; may emit behavioral code.
- Good support for domain specific languages although none implemented yet.

CHDL: Netlist Introspection



Module caching improves the performance of the elaboration and optimization phases.

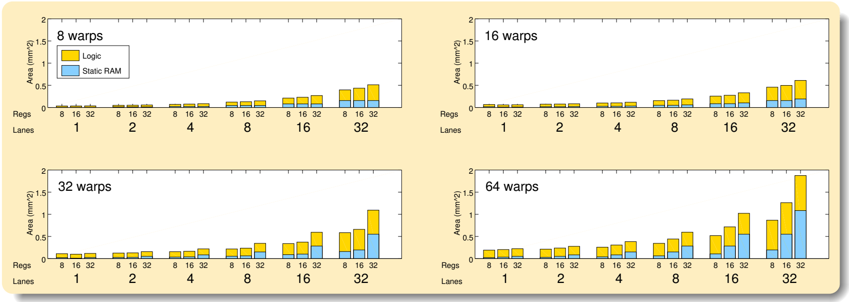
Module caching is a technique which:

- Stores cached, optimized netlists of submodules to disk.
- Improves performance on subsequent runs.

Harmonica: Harpbench

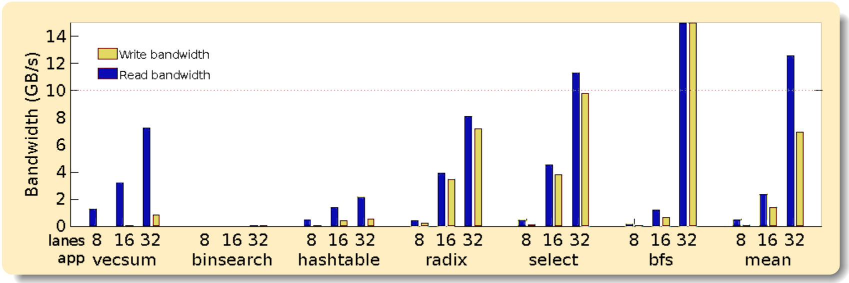
Description	Data	Size
Breadth-first search	PA road network.	1090920 nodes, 3083796 edges
Radix sort	Random integers	1048576 elements
Binary search.	Random integers.	1048576 elements, 1048576 lookups
Hash table lookup	Random integers	1048576 elements, 1048576 lookups
Sum integer vector	Random integers	16777216 elements
Select from table	Random values	1048576 elements, 1037940 matching rows

Harmonica: Area



- Logic/SRAM area in FreePDK15
- Covers a wide range of values depending on lane/reg/warp count.
- Note SRAM area dominates for large thread counts.

Harmonica: Bandwidth



- 32-lane version saturates available bandwidth on some benchmarks.
- At an area of approx. 1 sq. mm per core.
- Bandwidth utilization is cache-dependent.

Cheetah: Pipeline Scheduling

Some considerations are taken by Cheetah to ensure the pipeline does not deadlock or generate cyclic combinational logic:

- Stall signals are propagated back along trees; this means that the internal stall signal and the stall signal being presented to an upstream block may be produced by different logic.
- Priority may be set for any edge in the pipeline graph.
- Inner loops are given higher priority by default.

Cheetah: Instruction Set Processors

Cheetah was designed with instruction set based accelerators in mind:

- Mis-speculations, forwarding results, etc. can be broadcast using non-`plval` CHDL signals.
- May be used for full designs or individual functional units and combined with other paradigms.
- Dissertation includes floating point processor example with simple branch prediction.