National Science Foundation's Industry/University Cooperative Research (I/UCRC) Program

Lessons Learned with Performance Prediction and Design Patterns on Molecular Dynamics

CHREC
NSF Center for High-Performance Reconfigurable Computing

UF UNIVERSITY of FLORIDA

Virginia Tech
VIRGINIA POLYTECHNIC INSTITUTE AND STATE UNIVERSITY

BYU
BRIGHAM YOUNG UNIVERSITY

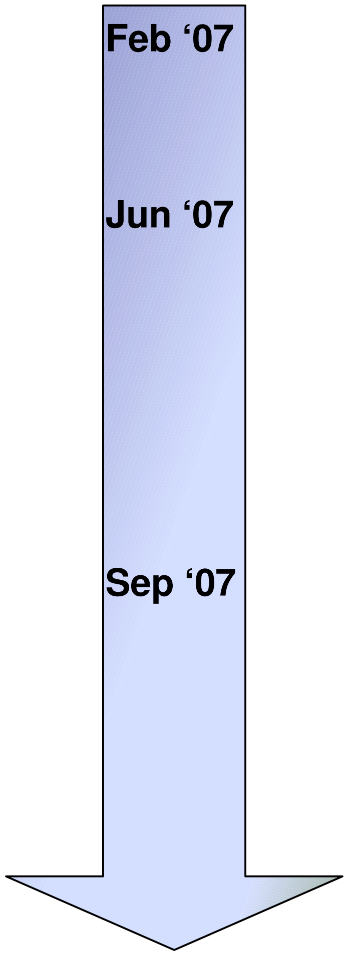
THE GEORGE WASHINGTON UNIVERSITY
WASHINGTON DC

<u>Brian Holland</u>
Karthik Nagarajan
Saumil Merchant
Herman Lam
Alan D. George

ECE Department, University of Florida
NSF CHREC Center

Outline of Algorithm Design Progression

- Algorithm decomposition
  - Design flow challenges
- Performance prediction
  - RC Amenability Test (RAT)
  - Molecular dynamics case study
  - Improvements to RAT
- Design patterns and methodology
  - Introduction and related research
  - Expanding pattern documentation
  - Molecular dynamics case study
- Conclusions



Design Evolution

Design Flow Challenges

- Original mission
  - Create scientific applications for FPGAs as case studies to investigate topics such as portability and scalability
    - Molecular dynamics is one such application
    - Goal is not application implementation but lessons learned from app.
  - Maximize performance and productivity using HLLs and high-performance reconfigurable computing (HPRC) design techniques
    - Applications should have *significant* speedup over SW baseline
- Challenges
  - Ensure speedup over traditional implementations
    - Particularly when researcher is not an RC engineer
  - Explore application design space thoroughly and efficiently
    - Several designs may achieve speedup but which should be used?

Algorithm Performance

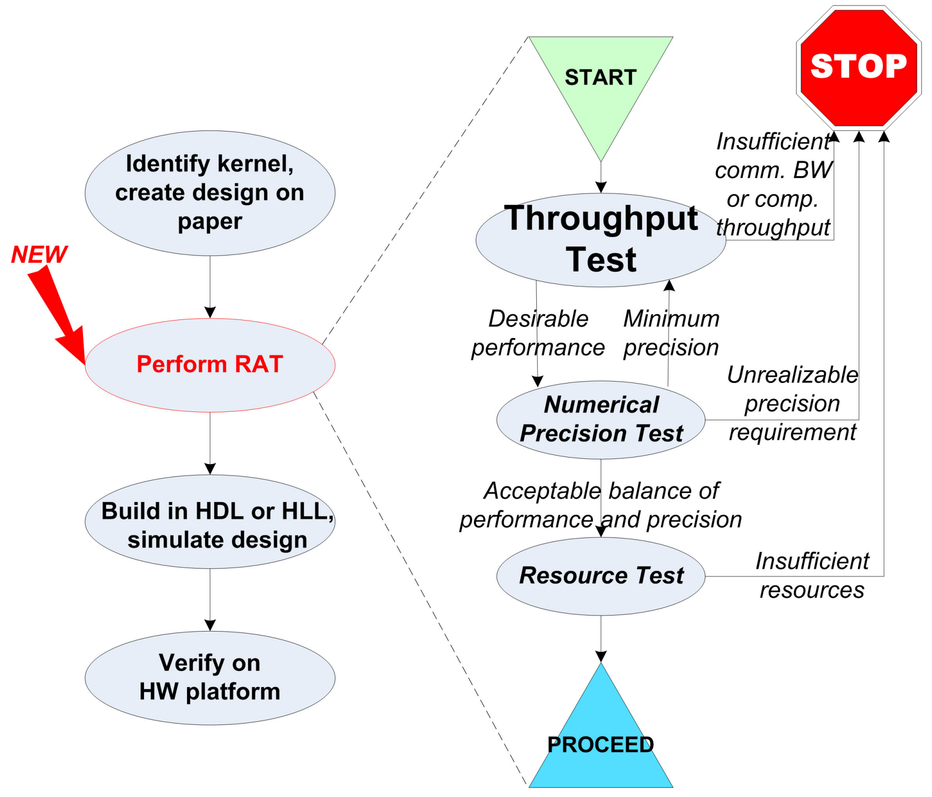
- Motivation
  - (Re)designing applications is expensive
    - Only want to design once and even then, do it most efficiently
  - Scientific applications can contain extra precision
    - Floating point may not be necessary but is used as a SW “standard”
  - Optimal design may overuse available FPGA resources
    - Discovering resource exhaustion mid-development is expensive

- Need
  - Performance prediction
    - Quickly and with reasonable accuracy estimate performance of a *particular* algorithm on a *specific* FPGA platform
    - Use simple analytic models to make prediction accessible to novices

RC Amenability Test (RAT)

"A methodology for fast and accurate RC performance prediction of a specific application on a specific platform before any hardware coding"

- Throughput Test
  - Algorithm and FPGA platform are parameterized
  - Equations are used to predict speedup
- Numerical Precision Test
  - RAT user should explicitly examine impact of reducing precision on computation
  - Interrelated with throughput test
    - Two tests essentially proceed simultaneously
- Resource Utilization Test
  - FPGA resources usage is estimated to determine scalability on FPGA platform



NEW
Identify kernel, create design on paper
Perform RAT
Build in HDL or HLL, simulate design
Verify on HW platform
START
STOP
Throughput Test
Insufficient comm. BW or comp. throughput
Desirable performance
Minimum precision
Numerical Precision Test
Unrealizable precision requirement
Acceptable balance of performance and precision
Resource Test
Insufficient resources
PROCEED

Overview of RAT Methodology

Original RAT Analytic Model

Communication time

$$t_{comm} = t_{read} + t_{write} \qquad t_{read} = \frac{N_{elements} \cdot N_{bytes/element}}{\alpha_{read} \cdot throughput_{ideal}} \qquad t_{write} = \frac{N_{elements} \cdot N_{bytes/element}}{\alpha_{write} \cdot throughput_{ideal}}$$

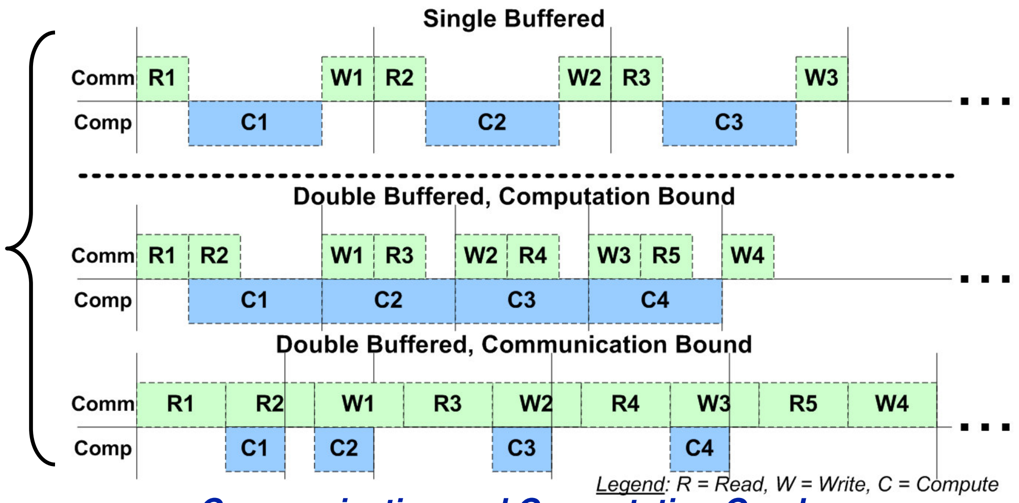
Computation time

$$t_{comp} = \frac{N_{elements} \cdot N_{ops/element}}{f_{clock} \cdot throughput_{proc}}$$

Total RC execution time

$$t_{rc_{SB}} = N_{iter} \cdot (t_{comm} + t_{comp})$$

$$t_{rc_{DB}} \approx N_{iter} \cdot Max(t_{comm}, t_{comp})$$



Single Buffered
Comm
Comp
R1
W1
R2
W2
R3
W3
C1
C2
C3
Double Buffered, Computation Bound
R4
W4
R5
C4
Double Buffered, Communication Bound
Legend: R = Read, W = Write, C = Compute

Communication and Computation Overlap for Single or Double Buffering

Speedup

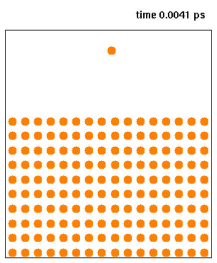
$$speedup = \frac{t_{soft}}{t_{RC}}$$

Application and RC platform attributes are parameterized and used in these equations to estimate performance.

CHREC NSF Center for High-Performance Reconfigurable Computing

UF UNIVERSITY of FLORIDA Virginia Tech BYU BRIGHAM YOUNG UNIVERSITY THE GEORGE WASHINGTON UNIVERSITY

Molecular Dynamics



```
void ComputeAccel() {
  double dr[3],f,fcVal,rrCut,rr,ri2,ri6,r1;
  int j1,j2,n,k;

  rrCut = RCUT*RCUT;
  for(n=0;n<nAtom;n++) for(k=0;k<3;k++) ra[n][k] = 0.0;
  potEnergy = 0.0;

  for (j1=0; j1<nAtom-1; j1++) {
    for (j2=j1+1; j2<nAtom; j2++) {
      for (rr=0.0, k=0; k<3; k++) {
        dr[k] = r[j1][k] - r[j2][k];
        dr[k] = dr[k]-SignR(RegionH[k],dr[k]-RegionH[k])
                    - SignR(RegionH[k],dr[k]+RegionH[k]);
        rr = rr + dr[k]*dr[k];
      }
      if (rr < rrCut) {
        ri2 = 1.0/rr; ri6 = ri2*ri2*ri2; r1 = sqrt(rr);
        fcVal = 48.0*ri2*ri6*(ri6-0.5) + Duc/r1;
        for (k=0; k<3; k++) {
          f = fcVal*dr[k];
          ra[j1][k] = ra[j1][k] + f;
          ra[j2][k] = ra[j2][k] - f;
        }
        potEnergy+=4.0*ri6*(ri6-1.0)- Uc - Duc*(r1-RCUT);
      }
    }
  }
}
```

SW Baseline Code

- Simulation of interactions of a set of molecules over a given time interval
  - Based upon code provided by Oak Ridge National Lab (ORNL)
- Challenges for accurate performance prediction of MD
  - Large simulation datasets
    - Exhaust FPGA's local memory
      - Sets of molecules are often on order of 100,000s of atoms, with dozens of time steps
  - Nondeterministic runtime
    - Molecules beyond a certain threshold are assumed to have zero impact
      - Certain sets require less comp.

CHREC NSF Center for High-Performance Reconfigurable Computing

UF UNIVERSITY of FLORIDA · Virginia Tech · BYU · THE GEORGE WASHINGTON UNIVERSITY

Molecular Dynamics

- Algorithm
  - 16,384 molecule data set
  - Written in Impulse C
  - XtremeData XD1000 platform
    - Altera Stratix II EPS2180 FPGA
    - HyperTransport interconnect
  - SW baseline on 2.4GHz Opteron
- Parameters
  - Dataset Parameters
    - Model volume of data used by FPGA
  - Communication Parameters
    - Model the HyperTransport interconnect
  - Computation Parameters
    - Model computational requirement of FPGA
    - $N_{ops/element}$
      - 164000 ≈ 16384 * 10 ops
      - i.e. each molecule (element) takes 10ops/iteration
    - $Throughput_{proc}$
      - 50
      - i.e. operations per cycle needed for >10x speedup
  - Software Parameters
    - Software baseline runtime and iterations required to complete RC application

Dataset Parameters		
Nelements, input	(elements)	16384
Nelements, output	(elements)	16384
Nbytes/element	(bytes/element)	36
Communication Parameters		
throughput(ideal)	(Mbps)	500
α(input)	0<α<1	0.9
α(output)	0<α<1	0.9
Computation Parameters		
Nops/element	(ops/element)	164000
throughput(proc)	(ops/cycle)	50
f(clock)	(MHz)	75/100/150
Software Parameters		
t(soft)	(sec)	5.76
N	(iterations)	1

RAT Input Parameters of MD

	Predicted	**Predicted**	**Predicted**	**Actual**
f(clock)	**75**	**100**	**150**	**100**
tcomm	2.62E-3	2.62E-3	2.62E-3	1.39E-3
tcomp	7.17E-1	5.37E-1	3.58E-1	8.79E-1
utilcomm	0.4%	0.5%	0.7%	0.2%
utilcomp	99.6%	99.5%	99.3%	99.8%
tRC	7.19E-1	5.40E-1	3.61E-1	8.80E-1
speedup	8	10.7	16	6.6

Performance Parameters of MD

Parameter Alterations for Pipelining

- MD Optimization
  - Each molecular pair's computation should be pipelined
    - Individual molecules have nondeterministic workloads
    - But, pairs of molecules will enter the pipeline at a constant rate
- Parameters
  - Computation Parameters
    - $N_{ops/element}$
      - 16400
      - Strictly number of interactions per element
    - $Throughput_{pipeline}$
      - .333
      - Number of cycles needed to per interaction. i.e. you can only stall pipeline for 2 extra cycles
    - $N_{pipeline}$
      - 15
      - Guess based upon predicted area usage

Dataset Parameters		
Nelements, input	(elements)	16384
Nelements, output	(elements)	16384
Nbytes/element	(bytes/element)	36

Communication Parameters		
throughput(ideal)	(Mbps)	500
α(input)	0<α<1	0.9
α(output)	0<α<1	0.9

Computation Parameters		
Nops/element	**(ops/element)**	**16400**
throughput(pipeline)	**(ops/cycle)**	**0.33333**
Npipelines	**(ops/cycle)**	**15**
f(clock)	(MHz)	75/100/150

Software Parameters		
t(soft)	(sec)	5.76
N	(iterations)	1

Modified RAT Input Parameters of MD

	Predicted	Predicted	Predicted	Actual
f(clock)	**75**	**100**	**150**	**100**
tcomm	2.62E-3	2.62E-3	2.62E-3	1.39E-3
tcomp	7.17E-1	5.37E-1	3.58E-1	8.79E-1
utilcomm	0.4%	0.5%	0.7%	0.2%
utilcomp	99.6%	99.5%	99.3%	99.8%
tRC	7.19E-1	5.40E-1	3.61E-1	8.80E-1
speedup	8	10.7	16	6.6

Performance Parameters of MD

Pipelined Performance Prediction

- Molecular Dynamics
  - If a pipeline is possible, certain parameters become obsolete
    - Number of operations in pipeline (i.e depth) is not important
    - Number of pipeline stalls becomes critical and is much more meaningful for non-deterministic apps
- Parameters
  - $N_{element}$
    - 16384^2
    - Number of molecular pairs
  - $N_{clks/element}$
    - 3
    - i.e. up to two cycles can be stalls
  - $N_{pipelines}$
    - 15
    - Same number of kernels as before

Dataset Parameters		
Nelements	(elements)	$(16384)^2$
Nclks/element	(cycle/element)	3
Npipelines		15
Depthpipeline	cycles	100
f(clock)	(MHz)	100
t(soft)	(sec)	5.76

Dataset Parameters		
tRC	(sec)	0.537
Speedup		10.7

Pipelined RAT Input Parameters of MD

$$t_{RC} = \frac{N_{elements} \cdot N_{clks/element}}{N_{kernels} \cdot f_{clk}} + \frac{Depth_{pipeline}}{f_{clk}} + t_{comm}$$

Modified RC Execution Time Equation

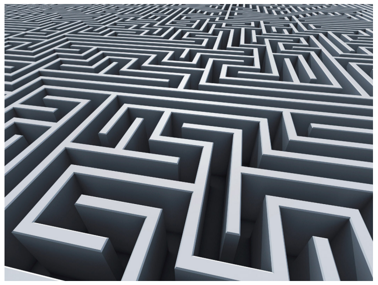
“And now for something completely different”

-Monty Python

(Or is it?)

Image from “Monty Python and the Holy Grail”. For academic purposes only.

Leveraging Algorithm Designs



- Introduction
  - Molecular dynamics provided several lessons learned
    - Best design practices for coding in Impulse C
    - Algorithm optimizations for maximum performance
    - Memory staging for minimal footprint and delay
      - Sacrificing computation efficiency for decreased memory accesses

- Motivations and Challenges
  - Application designs should educate the researcher
    - Successes and mistakes are retained to expedite future apps.
    - Application designs should also train other researchers
  - Unfortunately, new designing can be expensive
    - Collecting application knowledge into *design patterns* provides distilled lessons learned for efficient application

CHREC
NSF Center for High-Performance Reconfigurable Computing

UF UNIVERSITY of FLORIDA
Virginia Tech VIRGINIA POLYTECHNIC INSTITUTE AND STATE UNIVERSITY
BYU BRIGHAM YOUNG UNIVERSITY
THE GEORGE WASHINGTON UNIVERSITY WASHINGTON DC

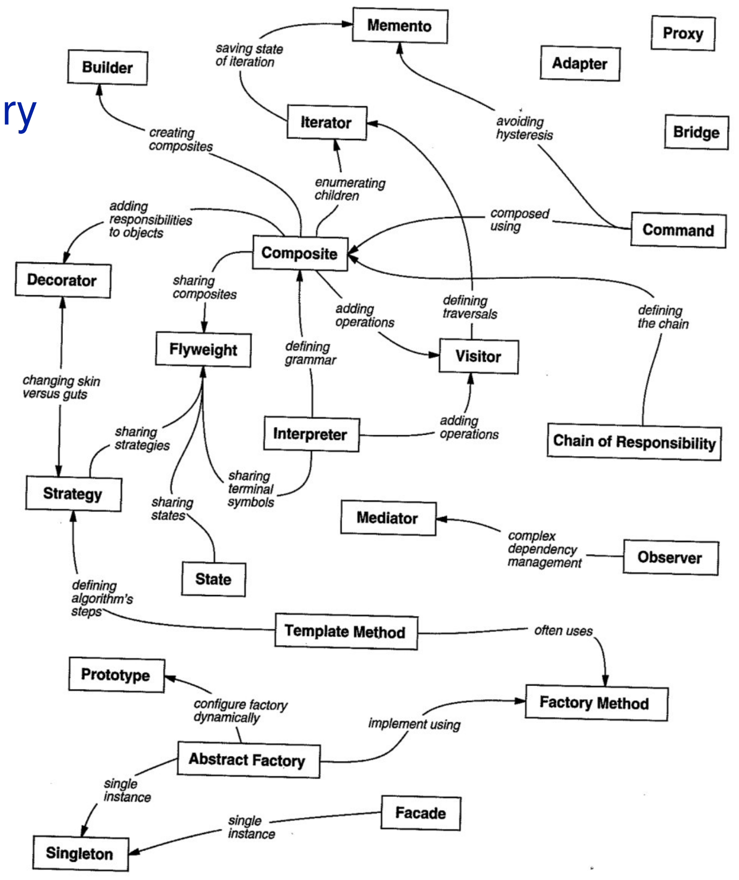
What are Design Patterns?

- Objected-oriented software engineering:
  - "A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design" [1]
- Reconfigurable Computing
  - "Design patterns offer us organizing and structuring principles that help us understand how to put building blocks (e.g., adders, multipliers, FIRs) together." [2]

1. Gamma, Eric, et al., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Boston, 1995.
2. DeHon, Andre, et al., "Design Patterns for Reconfigurable Computing", Proceedings of 12th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04), April 20-23, 2004, Napa, California.

Classification of Design Patterns – OO Textbook [1]

- Pattern categories
  - Creational
    - Abstract Factory
    - Prototype
    - Singleton
    - etc.
  - Structural
    - Adapter
    - Bridge
    - Proxy
    - etc.
  - Behavioral
    - Iterator
    - Mediator
    - Interpreter
    - etc.



- Describing Patterns
  - Pattern name
  - Intent
  - Also know as
  - Motivation
  - Applicability
  - Structure N/A?
  - Participants N/A?
  - Collaborations N/A?
  - Consequences
  - Implementation Useful
  - Sample code Useful
  - Known uses
  - Related patterns

NSF Center for High-Performance Reconfigurable Computing

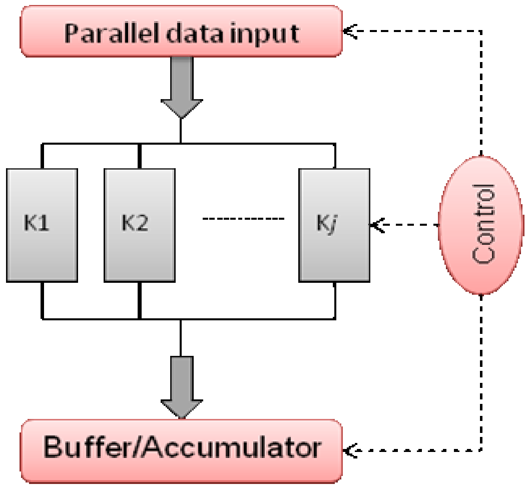
Sample Design Patterns – RC Paper [2]

- 14 pattern categories:
  - Area-Time Tradeoffs
  - Expressing Parallelism
  - Implementing Parallelism
  - Processor-FPGA Integration
  - Common-Case Optimization
  - Re-using Hardware Efficiently
  - Specialization
  - Partial Reconfiguration
  - Communications
  - Synchronization
  - Efficient Layout and Communications
  - Implementing Communication
  - Value-Added Memory Patterns
  - Number Representation Patterns
- 89 patterns identified (samples)
  - Coarse-Grained Time Multiplexing
  - Synchronous Dataflow
  - Multi-threaded
  - Sequential vs. Parallel Implementation (hardware-software partitioning)
  - SIMD
  - Communicating FSMDs
  - Instruction augmentation
  - Exceptions
  - Pipelining
  - Worst-Case Footprint
  - Streaming Data
  - Shared Memory
  - Synchronous Clocking
  - Asynchronous Handshaking
  - Cellular Automata
  - Token Ring
  - etc.

Example – Datapath Duplication

** Replicated computational structures for parallel processing*

- Intent – Exploiting computation parallelism in sequential programming structures (loops)
- Motivation – Achieving faster performance through replication of computational structures
- Applicability – data independent
  - No feedback loops (acyclic dataflow)
- Participants – Single computational kernel



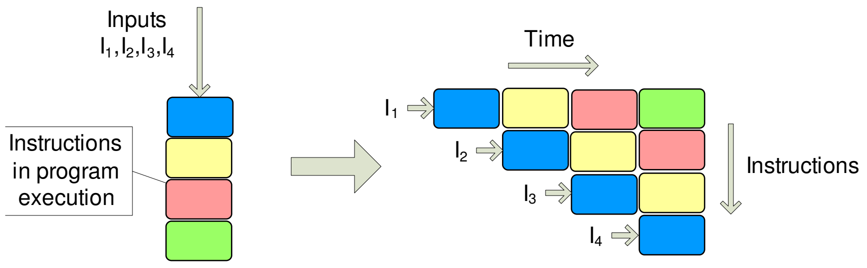
- Collaborations – Control algorithm directs dataflow and synchronization
- Consequences – Area time tradeoff, higher processing speed at cost of increased implementation footprint in hardware
- Known Uses – PDF estimation, BbNN implementation, MD, etc.
- Implementation – Centralize controller orchestrates data movement and synchronization of parallel processing elements

Example Design Pattern: Pipelining

Description

- Name
  - Pipelining (a.k.a. Instruction Pipelining)
- Motivation
  - Instruction throughput could be increased if design allows possibility to execute more instructions per unit of time
  - "Chain"-like instruction execution → increased processing speeds
- Consequences
  - Stall or wasted cycles for non-independent instructions in design
  - Extra registers and flip-flops in data path
- Known uses
  - Algorithm/program with instruction independency in its structure

Structure



Implementation

Pseudo HLL code

```
Void Pipeline_function()
{while(I_i) {-------------
              Instruction 1;
              Instruction 2;
              Instruction 3;
              -------------}}
```

Equivalent VHDL Code

```
entity Pipelining_SampleCode is
    Port (I_i : in std_logic_vector(31 downto 0);
          O_i : in std_logic_vector(31 downto 0)
          ------) end Pipelining_SampleCode;
architecture arch of Pipelining_SampleCode is
--signal declarations for intermediate values / buffers
begin
    temp1<=Instruction 1;
    temp2<=Instruction 2(temp1);
    temp3<=Instruction 3(temp2);
    -------
--Control algorithm for buffer/ dependency management
end arch;
```

CHREC NSF Center for High-Performance Reconfigurable Computing

Example Design Pattern: Memory Dependency

Description

- Name
  - Memory dependency resolution for efficient pipeline implementations
- Motivation
  - Resolve memory dependencies in computations for efficient pipeline implementations
- Applicability
  - Memory dependency may arise due to
    - Multiple reads from same memory in a single clock cycle.
    - Multiple reads and writes to a memory in a single clock cycle.
    - Multiple writes to a memory in a single clock cycle.
  - Memory dependency resolutions
    - Two parallel reads can be implemented using dual-ported memories where possible
    - Modifying operations to serialize memory accesses
- Consequences
  - Increasing number of pipeline stages
    - Not a problem for large number of iterations

```
for (i=0; i<n-1; i++) {
       c[i] = a[i] + a[i+1];
}
```

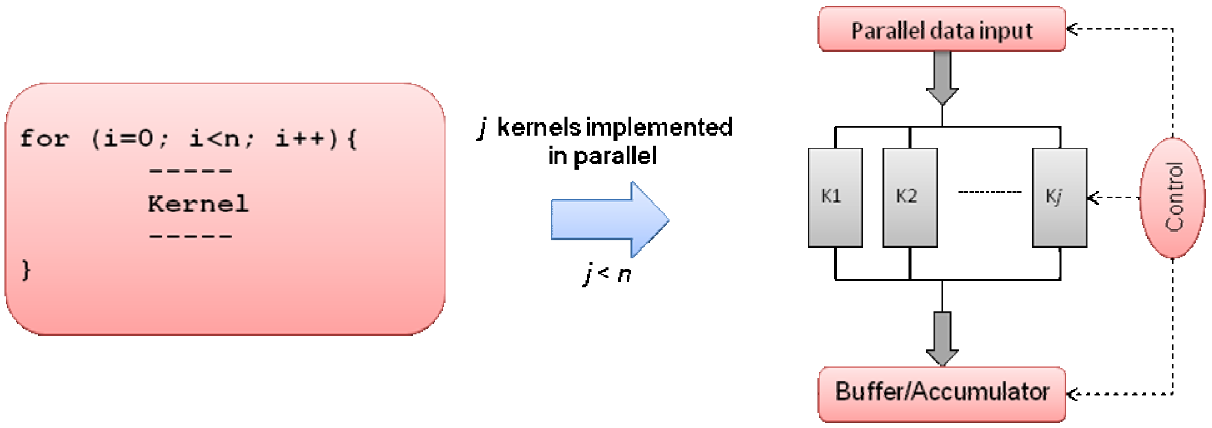
↓

```
for (i=0; i<n; i++) {
       a0 = a1;
       a1 = a[i];
       if(i>0) {c[i] = a0 + a1;}
}
```

CHREC NSF Center for High-Performance Reconfigurable Computing

UF UNIVERSITY of FLORIDA · Virginia Tech · BYU BRIGHAM YOUNG UNIVERSITY · THE GEORGE WASHINGTON UNIVERSITY

System-level Patterns for MD



Visualization of Datapath Duplication

- When design MD, initial goal is decompose algorithm into parallel kernels
  - "Datapath duplication" is a potential starting pattern
  - MD will require additional modifications since computational structure will not divide cleanly

"What do customers buy after viewing this item?"

67% use this pattern

37% alternatively use ….

"May we also recommend:"

Pipelining

Loop Fusion

"On-line Shopping" for Design Patterns

CHREC NSF Center for High-Performance Reconfigurable Computing

UF UNIVERSITY of FLORIDA Virginia Tech BYU THE GEORGE WASHINGTON UNIVERSITY

Kernel-level optimization patterns for MD

```
void ComputeAccel() {
  double dr[3],f,fcVal,rrCut,rr,ri2,ri6,r1;
  int j1,j2,n,k;

  rrCut = RCUT*RCUT;
  for(n=0;n<nAtom;n++) for(k=0;k<3;k++) ra[n][k] = 0.0;
  potEnergy = 0.0;

  for (j1=0; j1<nAtom-1; j1++) {
   for (j2=j1+1; j2<nAtom; j2++) {
    for (rr=0.0, k=0; k<3; k++) {
     dr[k] = r[j1][k] - r[j2][k];
     dr[k] = dr[k]-SignR(RegionH[k],dr[k]-RegionH[k])
                 - SignR(RegionH[k],dr[k]+RegionH[k]);
     rr = rr + dr[k]*dr[k];
    }
    if (rr < rrCut) {
     ri2 = 1.0/rr; ri6 = ri2*ri2*ri2; r1 = sqrt(rr);
     fcVal = 48.0*ri2*ri6*(ri6-0.5) + Duc/r1;
     for (k=0; k<3; k++) {
      f = fcVal*dr[k];
      ra[j1][k] = ra[j1][k] + f;
      ra[j2][k] = ra[j2][k] - f;
     }
     potEnergy+=4.0*ri6*(ri6-1.0)- Uc - Duc*(r1-RCUT);
    }
   }
  }
}
```

Pattern Utilization

🟩 2-D arrays
- SW addressing is handled by C compiler
- HW should be explicit

🟥 Loop fusion
- Fairly straightforward in explicit languages
- Challenging to make efficient in other HLLs

🟦 Memory dependencies
- Shared bank
  - Repeat accesses in pipeline cause stalls
- Write after read
  - Double access, even of same memory location, similarly causes stalls

CHREC NSF Center for High-Performance Reconfigurable Computing

UF UNIVERSITY of FLORIDA Virginia Tech BYU BRIGHAM YOUNG UNIVERSITY THE GEORGE WASHINGTON UNIVERSITY

Design Pattern Effects on MD

	Type	Stall Cycles
🟢	Nested Loop *pipeline depth * outer loop iterations*	d * N
🔴	Possible bank conflict *3 iterations * 1 extra access each*	3
🔵	Accumulation conflicts *Energy calc is longest*	18

```
void ComputeAccel() {
      double dr[3],f,fcVal,rrCut,rr,ri2,ri6,r1;
      int j1,j2,n,k;
      rrCut = RCUT*RCUT;
      for(n=0;n<nAtom;n++) for(k=0;k<3;k++) ra[n][k] = 0.0;
      potEnergy = 0.0;
      for (j1=0; j1<nAtom-1; j1++) {
            for (j2=j1+1; j2<nAtom; j2++) {
                  for (rr=0.0, k=0; k<3; k++) {
                        dr[k] = r[j1][k] - r[j2][k];
                        dr[k] = dr[k] - SignR(RegionH[k],dr[k]-RegionH[k])
                                    - SignR(RegionH[k],dr[k]+RegionH[k]);
                        rr = rr + dr[k]*dr[k];
                  }
                  if (rr < rrCut) {
                        ri2 = 1.0/rr; ri6 = ri2*ri2*ri2; r1 = sqrt(rr);
                        fcVal = 48.0*ri2*ri6*(ri6-0.5) + Duc/r1;
                        for (k=0; k<3; k++) {
                              f = fcVal*dr[k];
                              ra[j1][k] = ra[j1][k] + f;
                              ra[j2][k] = ra[j2][k] - f;
                        }
                        potEnergy+=4.0*ri6*(ri6-1.0)- Uc - Duc*(r1-RCUT);
                  }
            }
      }
}
```

C baseline code for MD

```
for (i=0; i<num*(num-1); i++){
      cg_count_ceil_32(1,0,i==0,num-2,&k);
      cg_count_ceil_32(1,0,i==0,num-2,&j2);
      cg_count_ceil_32(j2==0,0,i==0,num,&j1); if( j2 >= j1) j2++;

      if(j2==0) rr = 0.0;
      split_64to32_flt_flt(AL[j1],&j1y,&j1x);
      split_64to32_flt_flt(BL[j1],&dummy,&j1z);
      split_64to32_flt_flt(CL[j2],&j2y,&j2x);
      split_64to32_flt_flt(DL[j2],&dummy,&j2z);

      if(j1 < j2)   { dr0 = j1x - j2x; dr1 = j1y - j2y; dr2 = j1z - j2z;}
      else          { dr0 = j2x - j1x; dr1 = j2y - j1y; dr2 = j2z - j1z;}

      dr0 = dr0 - ( dr0 > REGIONH0 ? REGIONH0 : MREGIONH0 )
            - ( dr0 > MREGIONH0 ? REGIONH0 : MREGIONH0 );
      dr1 = dr1 - ( dr1 > REGIONH1 ? REGIONH1 : MREGIONH1 )
            - ( dr1 > MREGIONH1 ? REGIONH1 : MREGIONH1 );
      dr2 = dr2 - ( dr2 > REGIONH2 ? REGIONH2 : MREGIONH2 )
            - ( dr2 > MREGIONH2 ? REGIONH2 : MREGIONH2 );

      rr = dr0*dr0 + dr1*dr1 + dr2*dr2;
      ri2 = 1.0/rr; ri6 = ri2*ri2*ri2; r1 = sqrt(rr);
      fcVal = 48.0*ri2*ri6*(ri6-0.5) + Duc/r1;
      fx = fcVal*dr0; fy = fcVal*dr1; fz = fcVal*dr2;

      if(j2 < j1)   { fx = -fx; fy = -fy; fz = -fz; }

      fp_accum_32(fx, k==(num-2), 1, k==0, &ja1x, &err);
      fp_accum_32(fy, k==(num-2), 1, k==0, &ja1y, &err);
      fp_accum_32(fz, k==(num-2), 1, k==0, &ja1z, &err);
      if( rr<rrCut ) {
         comb_32to64_flt_flt(ja1y,ja1x,&EL[j1]);
         comb_32to64_flt_flt(0,ja1z,&FL[j1]);
         fp_accum_32(4.0*ri6*(ri6-1.0) - Uc - Duc*(r1-RCUT),
            i==lim-1,j1<j2, i==0, &potEnergy, &err);
      }
}
```

Carte MD, fully pipelined, 282 cycle depth

Conclusions

- Performance prediction is a powerful technique for improving efficiency of RC application formulation
  - Provides reasonable accuracy for rough estimate
  - Encourages importance of numerical precision and resource utilization in performance prediction
- Design patterns provide lessons-learned documentation
  - Records and disseminates algorithm design knowledge
  - Allows for more effective formulation of future designs
- Future Work
  - Improve connection b/w design patterns and performance prediction
  - Expand design pattern methodology for better integration with RC
  - Increase role of numerical precision in performance prediction