Array Replication to Increase Parallelism in Applications Mapped to Configurable Architectures

Heidi Ziegler, Priyadarshini Malusare, and *Pedro Diniz*

University of Southern California / Information Sciences Institute

4676 Admiralty Way, Suite 1001

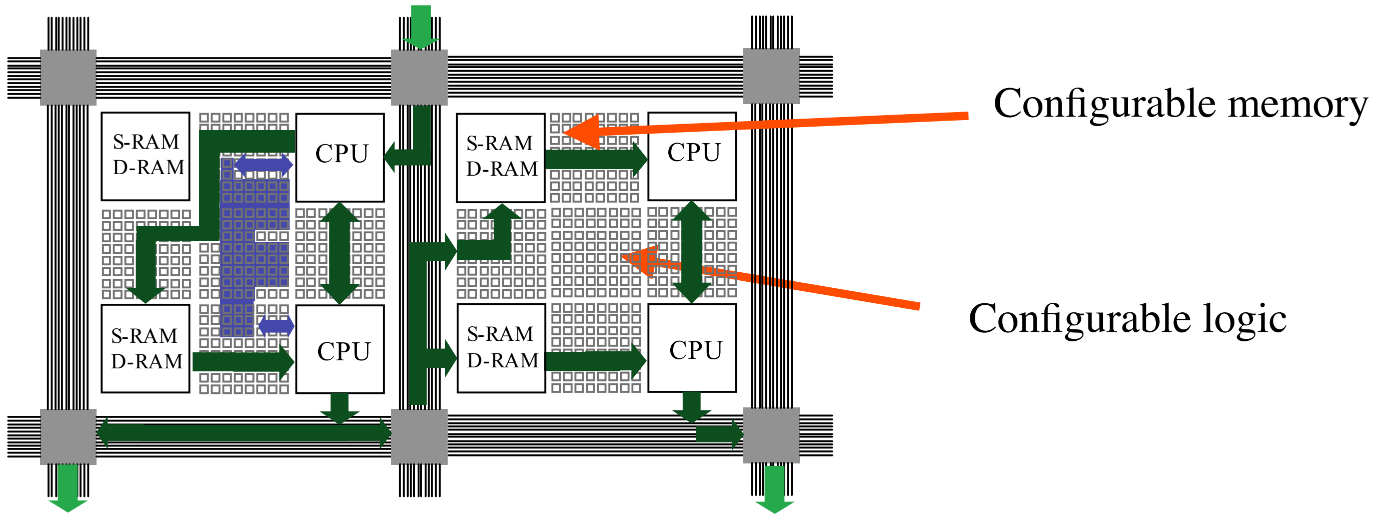
Marina del Rey, CA 90292, USA

{ziegler,priya,pedro}@isi.edu

***This work was partly funded by the National Science Foundation (NSF) under award number CCR-0209228 and NGS-0204040. Heidi Ziegler is also supported by Intel and the Boeing Company.**

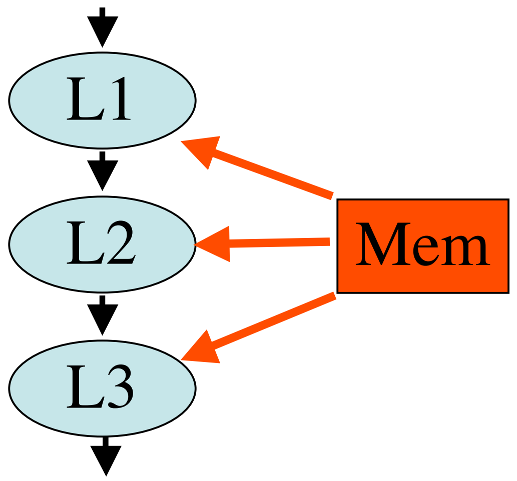
Motivation

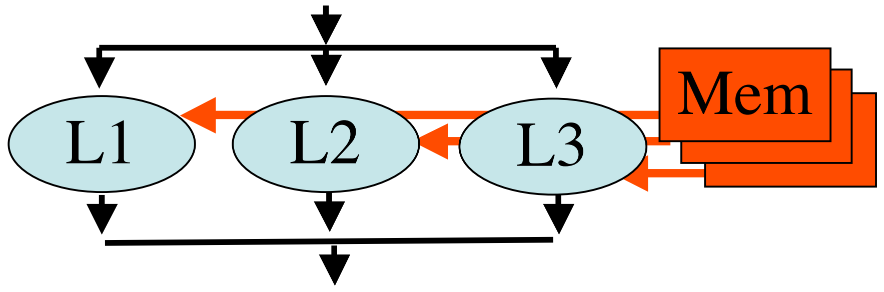
- Emerging architectures have configurable memories
  - Number, size, interconnect
- Opportunities
  - Large on-chip storage area for data
  - Large on-chip read and write bandwidth
  - Relatively low cost to replicate and copy data
- How we make use of these features?



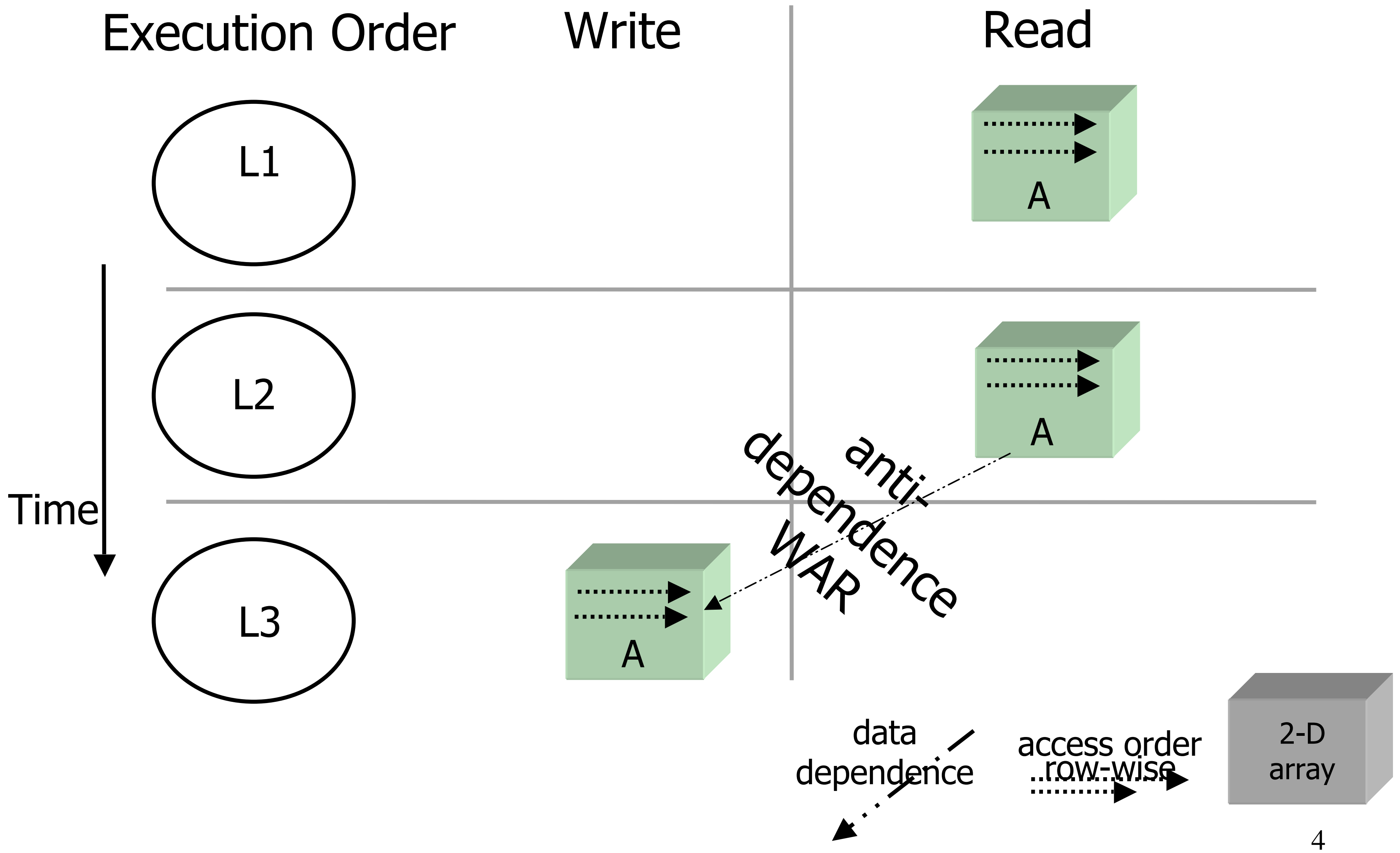
Basic Ideas

- Expose concurrency between loop nests
  - Replicate arrays to eliminate anti- and output-dependences
  - Add synchronization
  - Add update logic
- Eliminate memory contention
  - Replicate arrays
  - Add synchronization
  - Add update logic



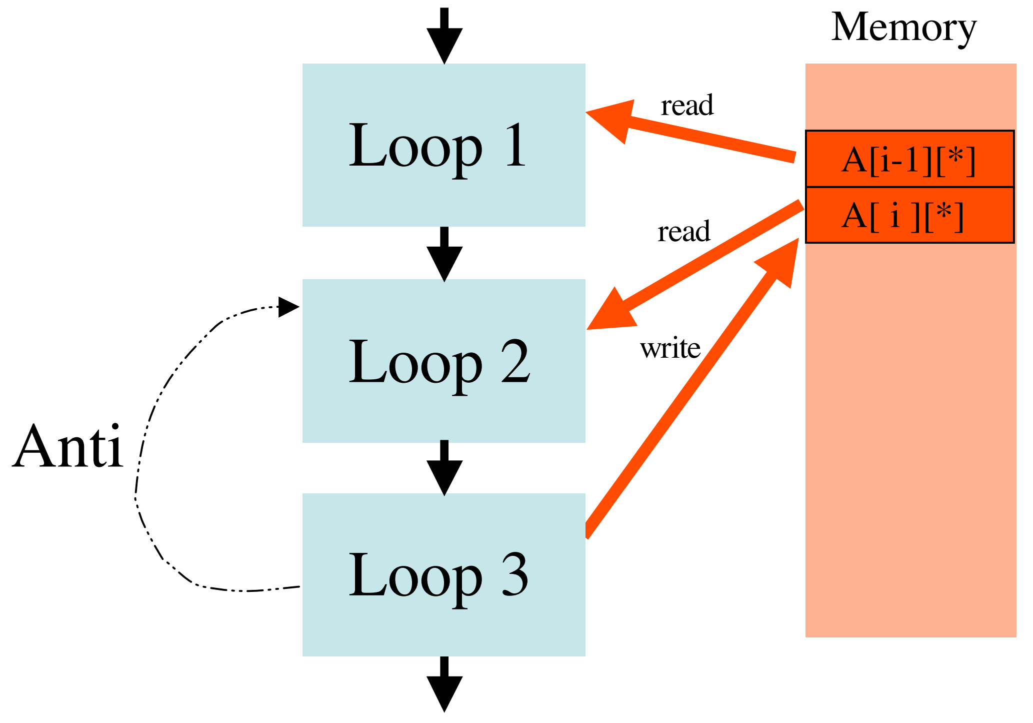


Dependence Definitions



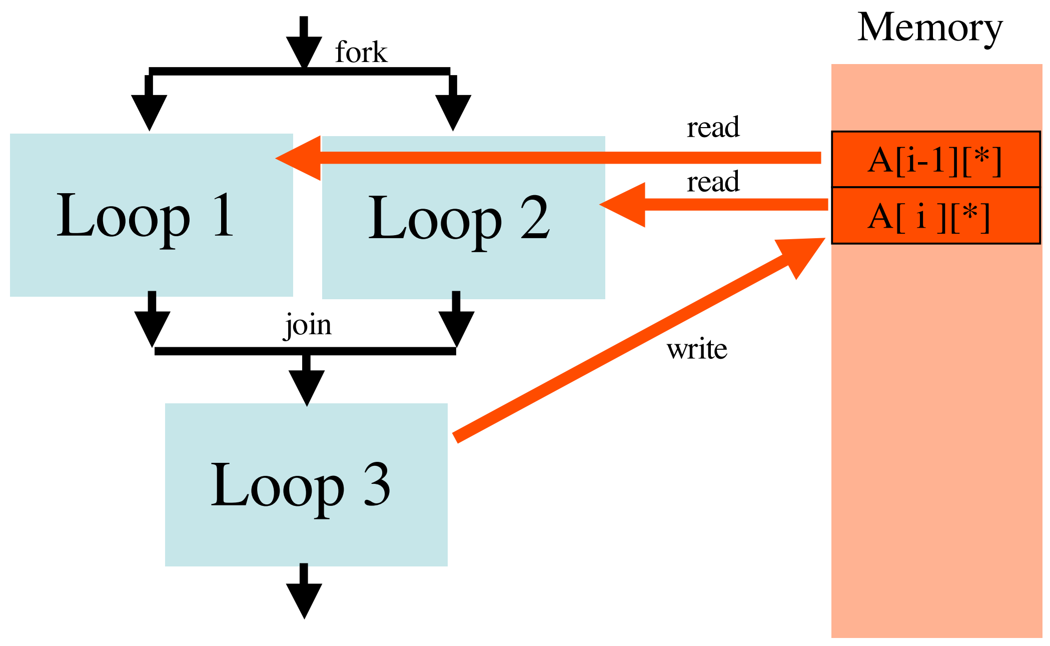
Example Kernel

- Start with all data mapped to the same memory and sequential execution



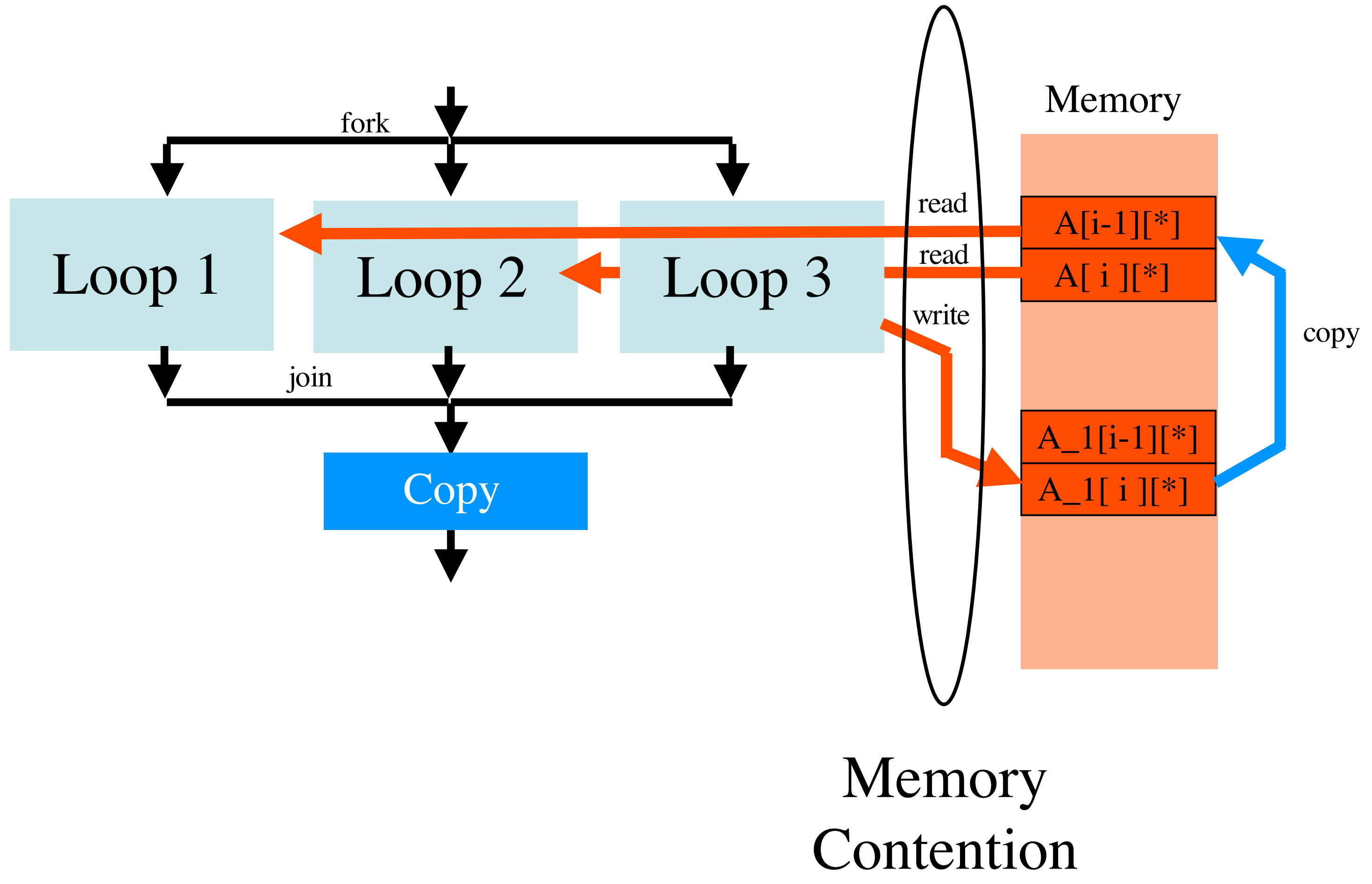
Exploiting Basic Data Independence

- L1 and L2 can be parallelized
- {L1, L2} and L3 can not

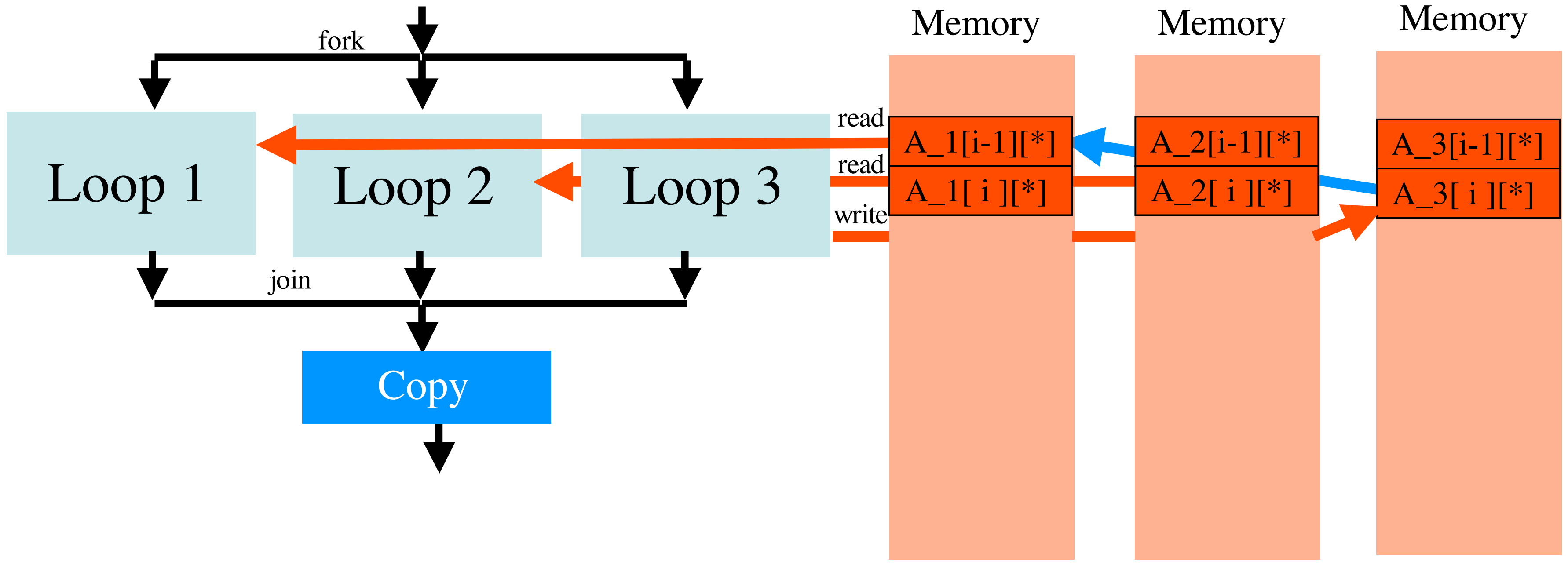


Using Array Renaming

- Create a local copy of array A in order to remove anti-dependence

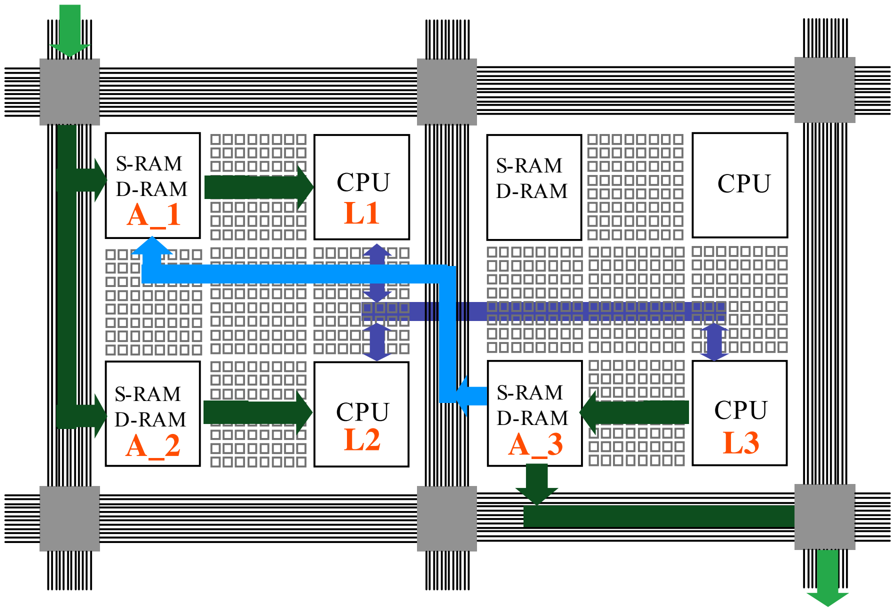


Using Array Renaming & Replication



- No memory contention but more memory space
- Care in updating copies across iterations of control loop

Mapping to a Configurable Architecture



- Many on-chip Memories so that Each Array May be accessed in Parallel
- Replication Operation done by Writing to Memories in Tandem using the same Bus

Array Data Access Descriptor

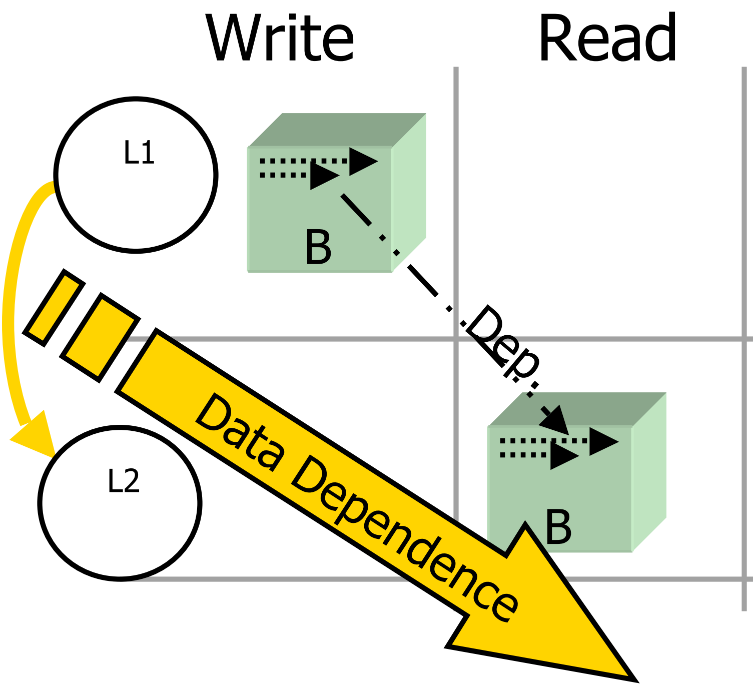
$$\{ER, WR\}_{n_i}^{A} = \left\langle \begin{array}{l} lb_1 < d1 < ub_1 \\ \ldots\ldots \\ lb_x < dx < ub_x \end{array} \right\rangle$$

Set describes **basic data access information**

n_i	program point / loop nest
A	array name
ER, WR	exposed read or write array access
lb, ub	lower and upper bound of each dimension $d1 \ldots dx$
	accessed array section, integer linear inequalities

Compiler Analysis: Data Dependence

Solve for data dependences



$$WR_{L1}^{B} = \left\langle \begin{array}{l} lb_1 \le d1 \le ub_1 \\ lb_2 \le d2 \le ub_2 \end{array} \right\rangle$$

$$ER_{L2}^{B} = \left\langle \begin{array}{l} lb_1 \le d1 \le ub_1 \\ lb_2 \le d2 \le u2_1 \end{array} \right\rangle$$

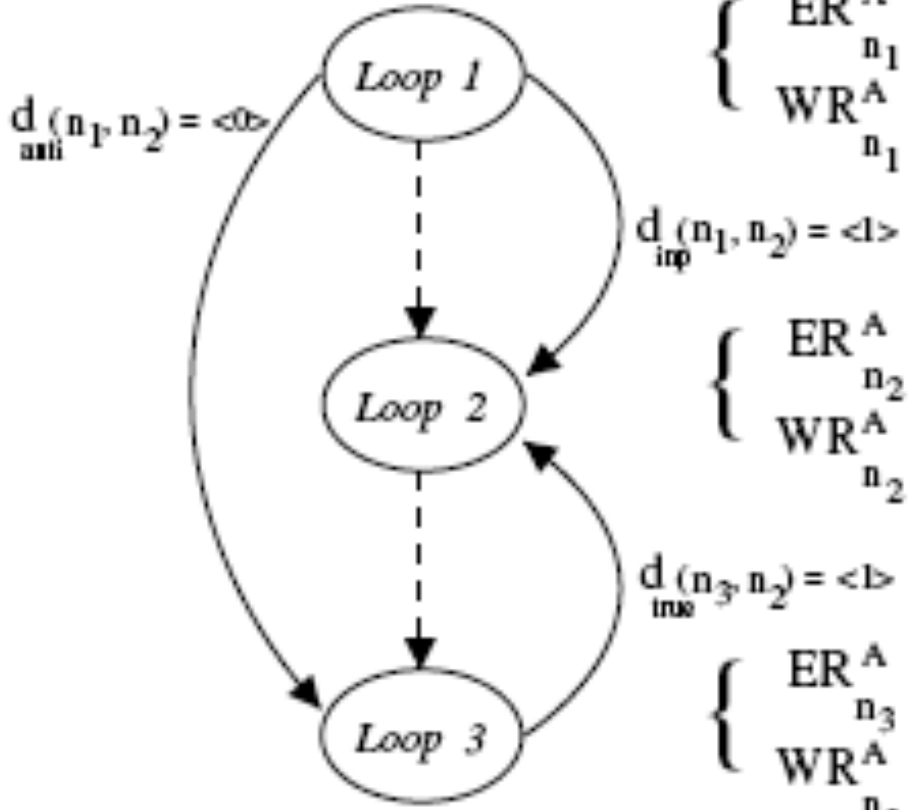
$$data\_dependence \Rightarrow f\left(WR_{ni}^{B}, ER_{nj}^{B}\right)$$

Compiler Analysis: Outline

- Outline:
  - Identify Control Loop
    - CFG with Coarse-grain task information
  - Extract Data Dependence Information
    - Exposed Read and Write Information
    - Array Section for Affine Array Accesses
  - Extract Parallel Regions
    - Using Array Renaming to Eliminate Anti-dependences
  - Identify Array Copies for Reduced Contention
    - Currently Replicate Write Arrays (partial replication)
    - Replicate All Write and Read Arrays (full replication)

- Status:
  - Compiler Analysis Implemented in SUIF
  - Code Generation and Translation to VHDL Still Manual

Analysis Example



Loop 1
Loop 2
Loop 3
$d_{anti}(n_1, n_2) = <0>$
$d_{inp}(n_1, n_2) = <1>$
$d_{true}(n_3, n_2) = <1>$

$$\begin{cases} ER^{A}_{n_1} = \{ i \le d0 \le i,\ 0 \le d1 \le N-1 \} \\ WR^{A}_{n_1} = \{ \} \end{cases}$$

$$\begin{cases} ER^{A}_{n_2} = \{ i-1 \le d0 \le i-1,\ 0 \le d1 \le N-1 \} \\ WR^{A}_{n_2} = \{ \} \end{cases}$$

$$\begin{cases} ER^{A}_{n_3} = \{ \} \\ WR^{A}_{n_3} = \{ i \le d0 \le i,\ 0 \le d1 \le N-1 \} \end{cases}$$

n_3 post dominates $\{n_1, n_2\}$

Experimental Methodology

- Goal
  - Evaluate Cost/Benefit of Array Renaming and Replication
  - Configurable logic device - field programmable gate array (FPGA)
  - Use of Many Memory Blocks for Array Storage

- Synthetic Kernels
  - HIST: 3 loop nests; 3 arrays
  - BIC: 4 loop nests; 4 arrays (most array data)
  - LCD: 3 loop nests; 2 arrays

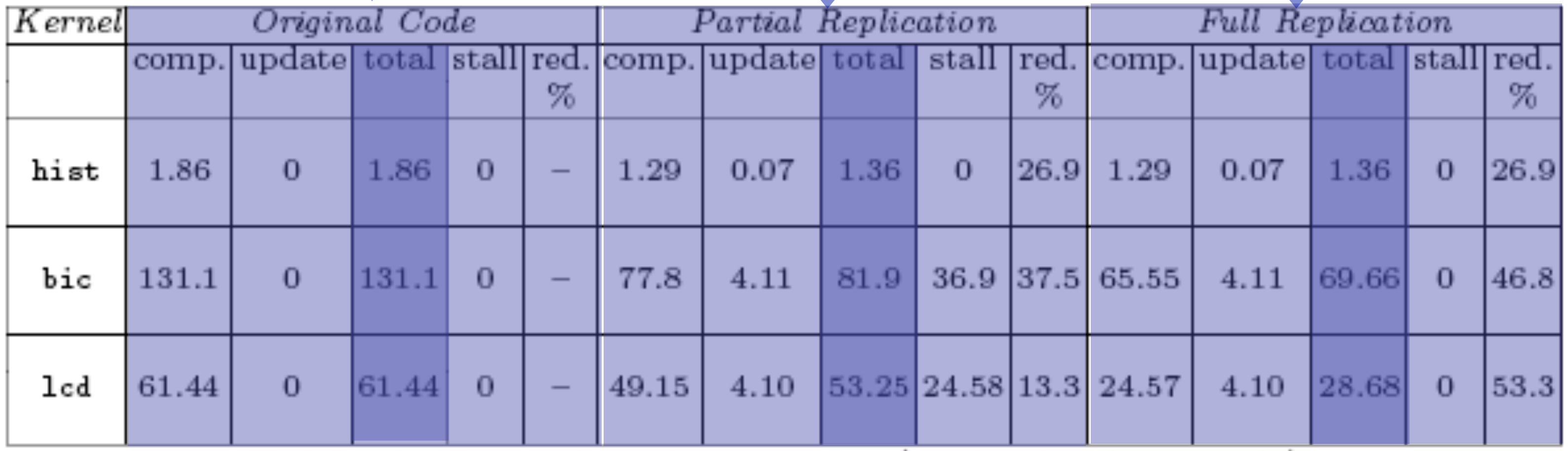
- Methodology
  - Analyze using SUIF and Transform Benchmarks Manually
  - Loop level execution times and Memory Schedules from Monet^{TM}
  - Simulate total execution times using loop level inputs
  - Manual Replication Transformation

Execution Time Results

Original Code

Partial Replication
(replication of written arrays)

Full Replication
(replication of all arrays)



Kernel	*Original Code*					*Partial Replication*					*Full Replication*				
	comp.	update	total	stall	red. %	comp.	update	total	stall	red. %	comp.	update	total	stall	red. %
hist	1.86	0	1.86	0	–	1.29	0.07	1.36	0	26.9	1.29	0.07	1.36	0	26.9
bic	131.1	0	131.1	0	–	77.8	4.11	81.9	36.9	37.5	65.55	4.11	69.66	0	46.8
lcd	61.44	0	61.44	0	–	49.15	4.10	53.25	24.58	13.3	24.57	4.10	28.68	0	53.3

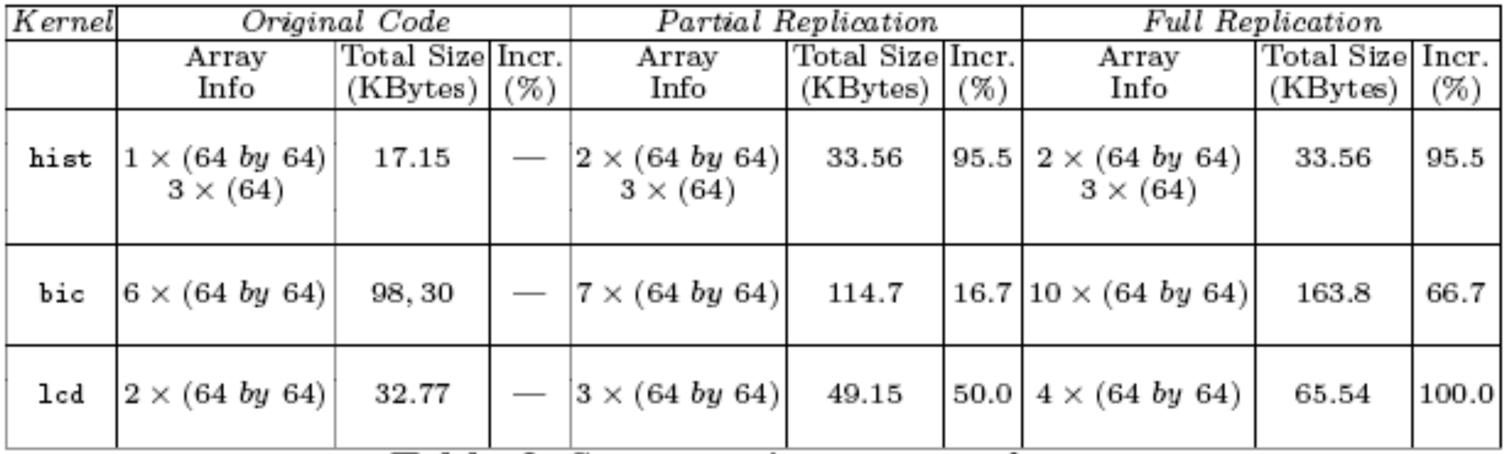
Table 1. Execution time results (cycles in thousands).

Kernel

Execution Cycles (simulation)
- computation
- update of replicas
- total execution
- stall due to memory contention
- overall reduction (percentage)

Fully replicated code versions achieve speedups between 1.4 and 2.1

Storage Requirement Results



Kernel	*Original Code*			*Partial Replication*			*Full Replication*		
	Array Info	Total Size (KBytes)	Incr. (%)	Array Info	Total Size (KBytes)	Incr. (%)	Array Info	Total Size (KBytes)	Incr. (%)
`hist`	$1 \times (64\ by\ 64)$ $3 \times (64)$	17.15	—	$2 \times (64\ by\ 64)$ $3 \times (64)$	33.56	95.5	$2 \times (64\ by\ 64)$ $3 \times (64)$	33.56	95.5
`bic`	$6 \times (64\ by\ 64)$	98, 30	—	$7 \times (64\ by\ 64)$	114.7	16.7	$10 \times (64\ by\ 64)$	163.8	66.7
`lcd`	$2 \times (64\ by\ 64)$	32.77	—	$3 \times (64\ by\ 64)$	49.15	50.0	$4 \times (64\ by\ 64)$	65.54	100.0

Table 2. Space requirements results.

Kernel

Space Requirements
- size in KBytes
- increment

Fully replicated code versions require storage increase by a factor of 2

Discussion

- Overhead of Updating Copies can be Negligible
  - Provided Enough Bandwidth for Updates
  - Small Number of Replicas
- Memory Contention
  - Even with a Small Number of Arrays can be Substantial
  - Scheduling could Mitigate this Issue somewhat…
- Preliminary Results Reveal:
  - Removal of anti-dependences can enable substantial increases in execution speed the cost of modest increase in storage
  - Increase in Space can be non-negligible if Initial Footprint is Small

Related Work

- Array Privatization
  - Eigenmann *et al*. LCPC 1991; Li ICS 1992; Tu *et al*. LCPC 1993
- Fine-grain Memory Parallelism
  - So *et*. *al*. CGO 2004
- Pipelining and Communication for FPGAs
  - Tseng PPoPP 1995; Ziegler *et*. *al*. DAC 2003

- This work:
  - Relaxes constraints on previous analyses
    - Loop Nests rather than Statements
    - Coarser-grain & Loop Carried Dependences
  - Combines the transformations to take advantage of configurable architecture characteristics
    - such as many on-chip memories
    - low cost array replication

Conclusion

- This paper:
  - Describes a Simple Loop Nests Analysis for Task-Level Parallelism
    - Uses Renaming and Replication to Eliminate Dependences
    - Across loop nests with selected replication strategies
    - Array Section Analysis to identify replication regions for each Array
  - Results target configurable architectures with
    - Many on-chip memories
    - Programmable Chip Routing
  - Results
    - Need to be Expanded to Larger Kernels
    - Respectable Speedups with Modest Space Increase.

Thank You