

# Iteration Disambiguation for Parallelism Identification in Time- Sliced Applications

Shane Ryoo, Christopher I. Rodrigues,  
and Wen-mei W. Hwu

Center for Reliable and High-Performance Computing  
Department of Electrical and Computer Engineering

University of Illinois at Urbana-Champaign



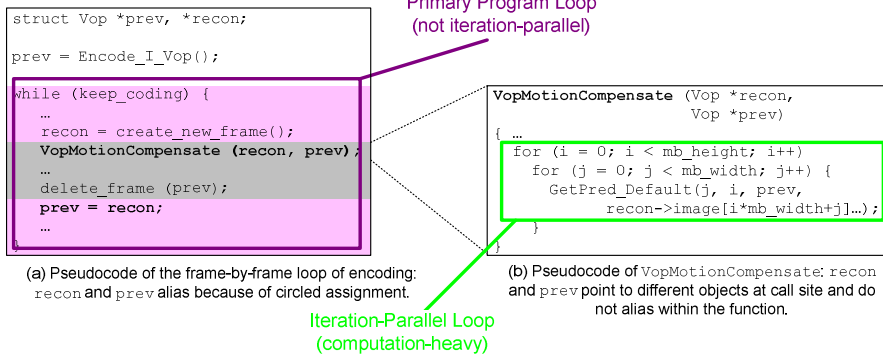
## *Time-Sliced Applications in C*

- Examples: media and scientific simulation applications
- Iterate through a large outer loop
  - The output of an iteration (time slice) is the input of the next iteration
  - Thus, serial dependences across iterations
- In inner loops:
  - Large amounts of computation: majority of execution time
  - Abundant iteration-level parallelism



20<sup>th</sup> Workshop on LCPC, October 11<sup>th</sup>, 2007

## A Basic Code Example



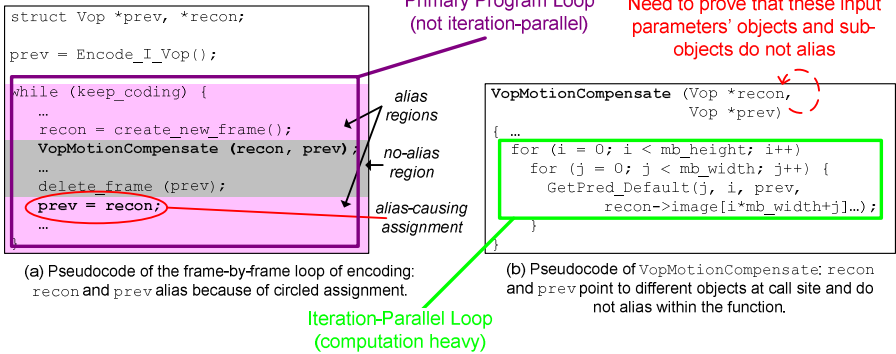
## Common Allocation Patterns

Allocation Pattern	Proposed Solutions
Statically allocated buffers; output is copied to input	Generic points-to pointer analysis
Statically allocated buffers; swapped among pointers	Alias pairs [Landi & Ryder], Connection analysis [Ghiya & Hendren]
Dynamically allocated buffers; Swapped among pointers	Connection analysis, <b>Iteration disambiguation</b>

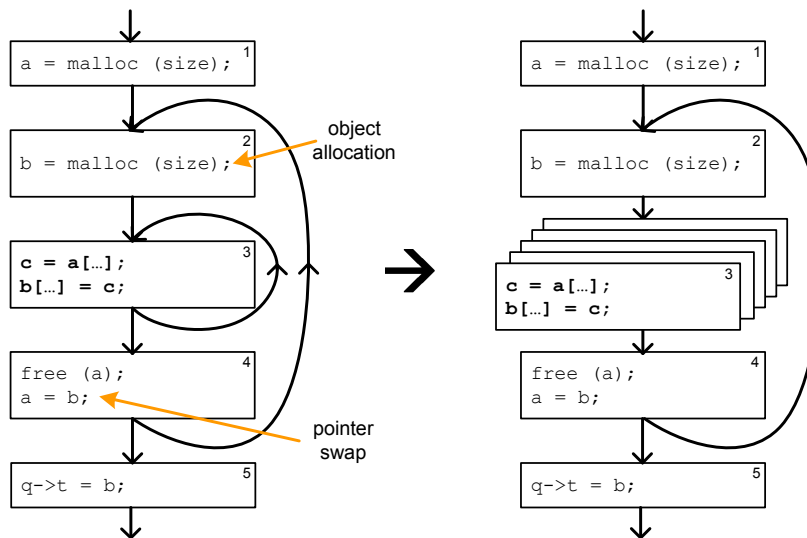
The third is becoming more prevalent, since it is a logical and easily implemented pattern when there are multiple older buffers

- Multiple time slices (H.264 video)
- Multiple buffers per time slice (some MPEG-4 modes)

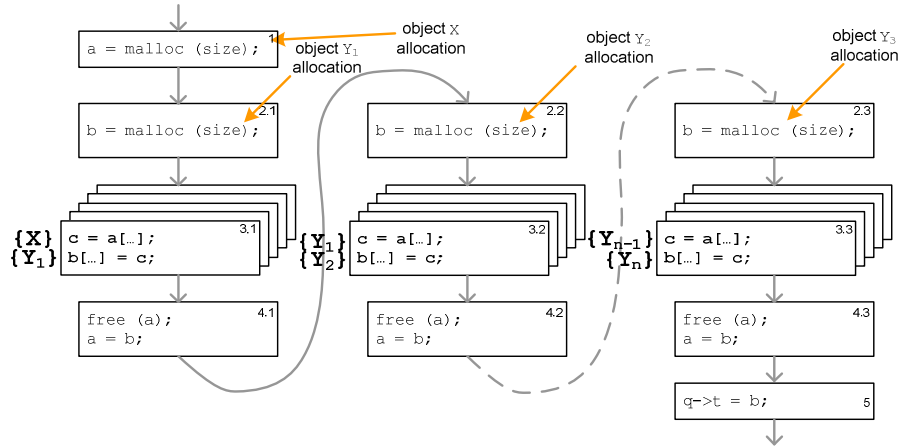
## Code Example Revisited



## A Similar Example



## What's the Problem?



Examining a dynamic trace, it is obvious that the inner loop can be parallelized...



20th Workshop on LCPC, October 11th, 2007

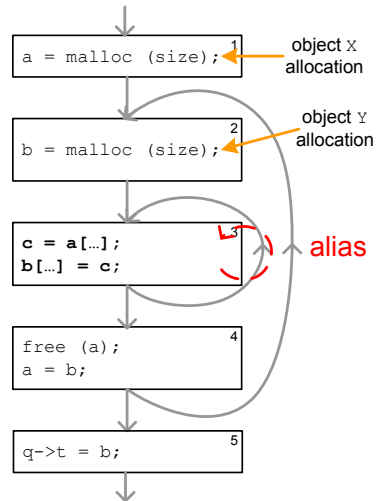
## Problem Exposed

But from a static standpoint, it's not quite so clear.

There is a real conflict between the loads of *a* and stores of *b* because the iterations are aggregated in the static representation.

$\{X, Y_1, \dots, Y_{n-1}\}$   
 $\{Y_1, Y_2, \dots, Y_n\}$

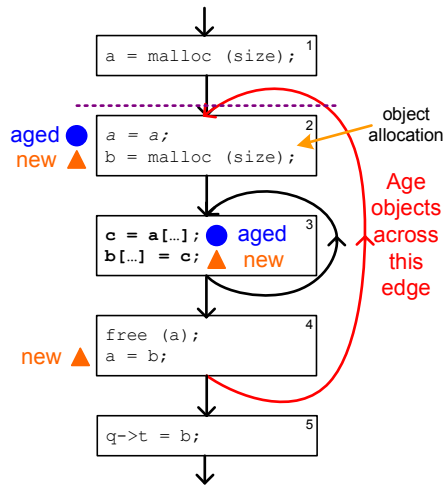
Is there a way to indicate that *a* is an older version of *b* and the two do not conflict within an iteration?



20th Workshop on LCPC, October 11th, 2007

## Aging

- Label the most recently allocated object `new`
- Previously allocated objects are called `aged`
- `Aged` and `new` objects do not alias in the scope defined by aging points (at the top of loops containing the allocation)
- This notation sufficiently expresses the relationship between the pointers



## Iteration Disambiguation: Cycle-Sensitive Analysis

Goal: disambiguate the objects within the limitations of the static code representation

- Pointers to the objects alias at some point
  - Need to define region where they do or do not alias
  - Fine-grained precision is expensive and often unnecessary
- Applications usually have multiple levels of function calls
  - Analysis must be interprocedural in scope
  - For scalability, avoid maintaining multiple contexts of analysis information

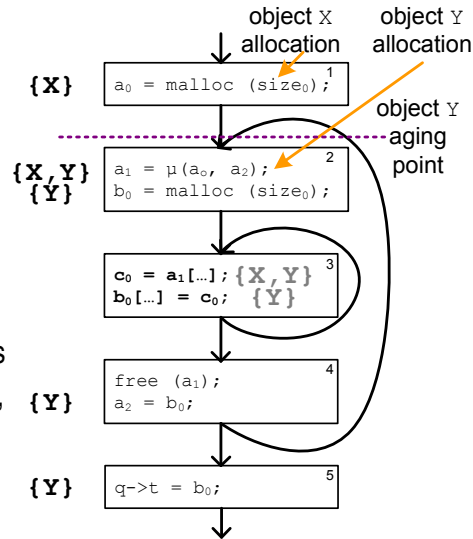
# *Analysis*

## *Properties of the Analysis*

- Core idea: distinguish the most recently allocated object (*new*) from older objects (*aged*) within the scope of an iteration of the large program loop
- Built as an extension of a points-to pointer analysis
- Track references that are passed via local variables
- Monotonic dataflow
- Make conservative estimates for references obtained through memory

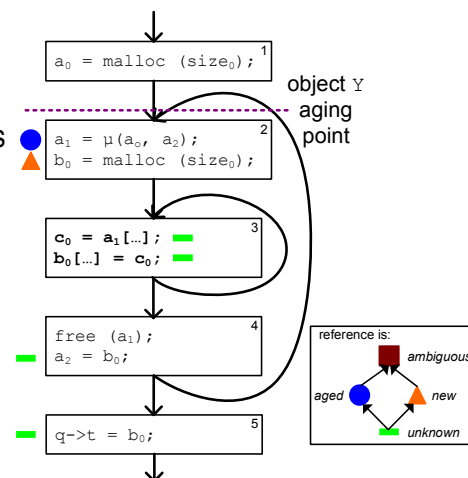
## Setup

- Run a points-to pointer analysis
  - Heap-sensitive pointer analysis preferable for better resolution
- Convert to SSA form
  - $\mu$ -functions used to mark entry to loops
- Aging points are placed at the entry of functions and loops that contain allocations of the objects
- Since  $b$  never points to object  $X$ , we do not have to analyze that object to find the parallelism in block 3



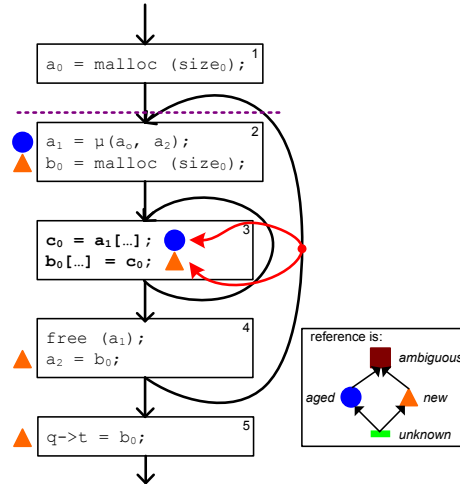
## Predicate Initialization

- Pointers which receive the returned address of allocation calls are marked *new*
- The destination of a  $\mu$ -function is marked *aged* for an object if it is located at an aging point for the object
- An input parameter of a function is marked *aged* if the function contains an allocation of the object



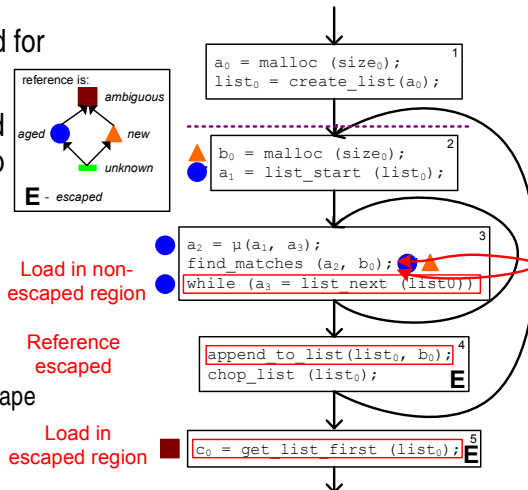
## Propagation

- Ages are propagated forward in the code, stopping at aging points
- If aged and new references meet at a  $\phi$ -function, the result is ambiguous
- If two pointers have different, non-ambiguous age types for each object they point to, they do not alias within a limited scope



## Extending the Analysis

- Many object references are stored to memory and loaded for later use
- Extension: references loaded from memory are aged if no new references have escaped to memory
- Need to propagate escaped reference dataflow
  - Start where new and ambiguous references escape
  - End at aging points

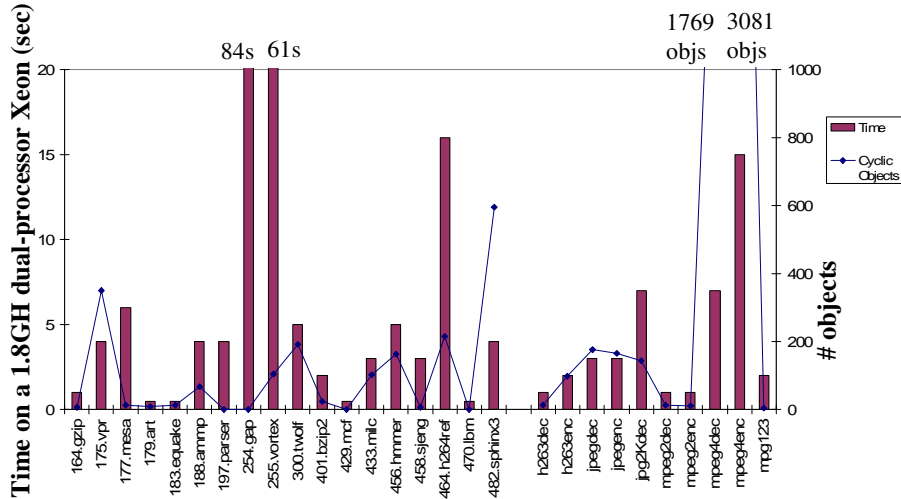


## ***What's Lacking?***

- Iteration disambiguation only addresses top-level objects
  - Most objects containing heavily-processed data are children objects
  - We need to prove that children objects are unique to a parent
    - Multi-dimensional array analysis (Wu et al., ICS 2002)
    - A more general tree-structure analysis (ongoing work)
- Even having information about child objects, this is insufficient to discover useful parallelism in contemporary programs
  - Arrays indices are non-affine due to variable (image) sizes
  - Cannot prove lack of array overrun
    - Requires knowledge of value bounds and relationships
    - Some codes prevent proof of parallelism because they do not properly screen out incorrect inputs

## ***Results***

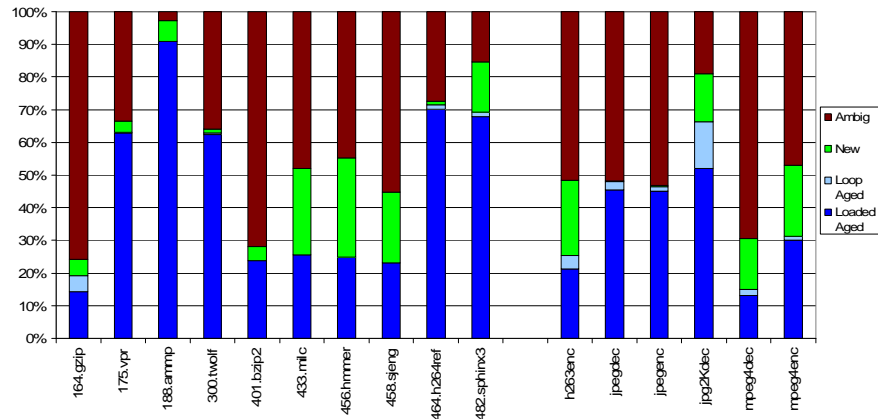
## Iteration Disambiguation Scalability



20<sup>th</sup> Workshop on LCPC, October 11<sup>th</sup>, 2007

## Iteration Disambiguation Results

Percentage of static count of sources of memory operations. Only-new object references omitted.



New references are independent from aged references.  
Neither are independent from ambiguous references.



20<sup>th</sup> Workshop on LCPC, October 11<sup>th</sup>, 2007

## **Conclusion**

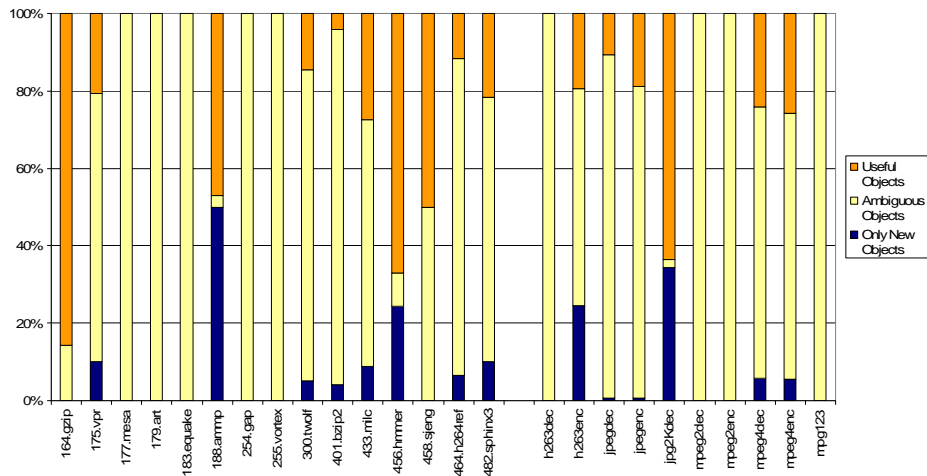
- We have presented an analysis that:
  - Runs quickly
  - Is sufficient to identify the region-specific independence of top-level objects in the applications we have examined
  - Is only one of several analyses that are needed to fully parallelize applications
- To extract iteration-level parallelism, we need
  - An additional analysis to propagate relationships to sub-objects
  - Methods to handle non-affine array accesses

## **Questions?**

## Recursion Support

- Although iteration disambiguation is not designed to address recursion, we would like to handle it gracefully
- Recursive code allows code to “backwards” across aging points
  - New references become *aged* on function entry
  - If returned, the *aged* references may co-exist with *new* references pointing to the same object
  - Same problem with newly-*aged* references loaded from memory
- Current solution: when returning from a function that may have allocated the object,
  - Make *aged* references ambiguous when returned
  - Start *escaped* region at function return

## Object Classification



Some objects never escape to memory and don't have a real cyclic dependence.

Others we can't draw any conclusions from due to the lack of both new and aged references.