Cork: Dynamic Memory Leak Detection for Garbage-Collected Languages

Aaron Brown
Colin Kern

4/24/2007

Outline

- Motivation
  - Memory Management Errors
  - Garbage Collectors
- Solution: Cork
  - Type Points-From Graphs
  - Detecting Heap Growth
  - Correlating to Data Structures
  - Efficiency and Scalability

Outline

- Results
  - Case Studies
  - Effect of Parameters
  - Overhead
- Related Work
- Conclusions

Motivation: Memory Management Errors

- Three types of errors occur with explicit memory management
  - Dangling pointers
  - Lost pointers
  - Unnecessary references

Motivation: Dangling Pointers

- Data being pointed to is freed
- Pointer is dereferenced

```
int main()
{
   int* x;
   x = malloc(sizeof(int));
   *x = 5;
   free(x);          //x is now a dangling pointer
   printf("%d", *x);  //ERROR
}
```

Motivation: Lost Pointers

- All pointers pointing to heap data are removed, but the data was not freed.

```
int main()
{
   int *x, *y, *z;
   x = malloc(sizeof(int));
   y = x;
   *x = 5;
   z = malloc(sizeof(int));
   x = y = z;     //The memory x and y pointed to still
                  //exists.
}
```

Motivation: Unnecessary References

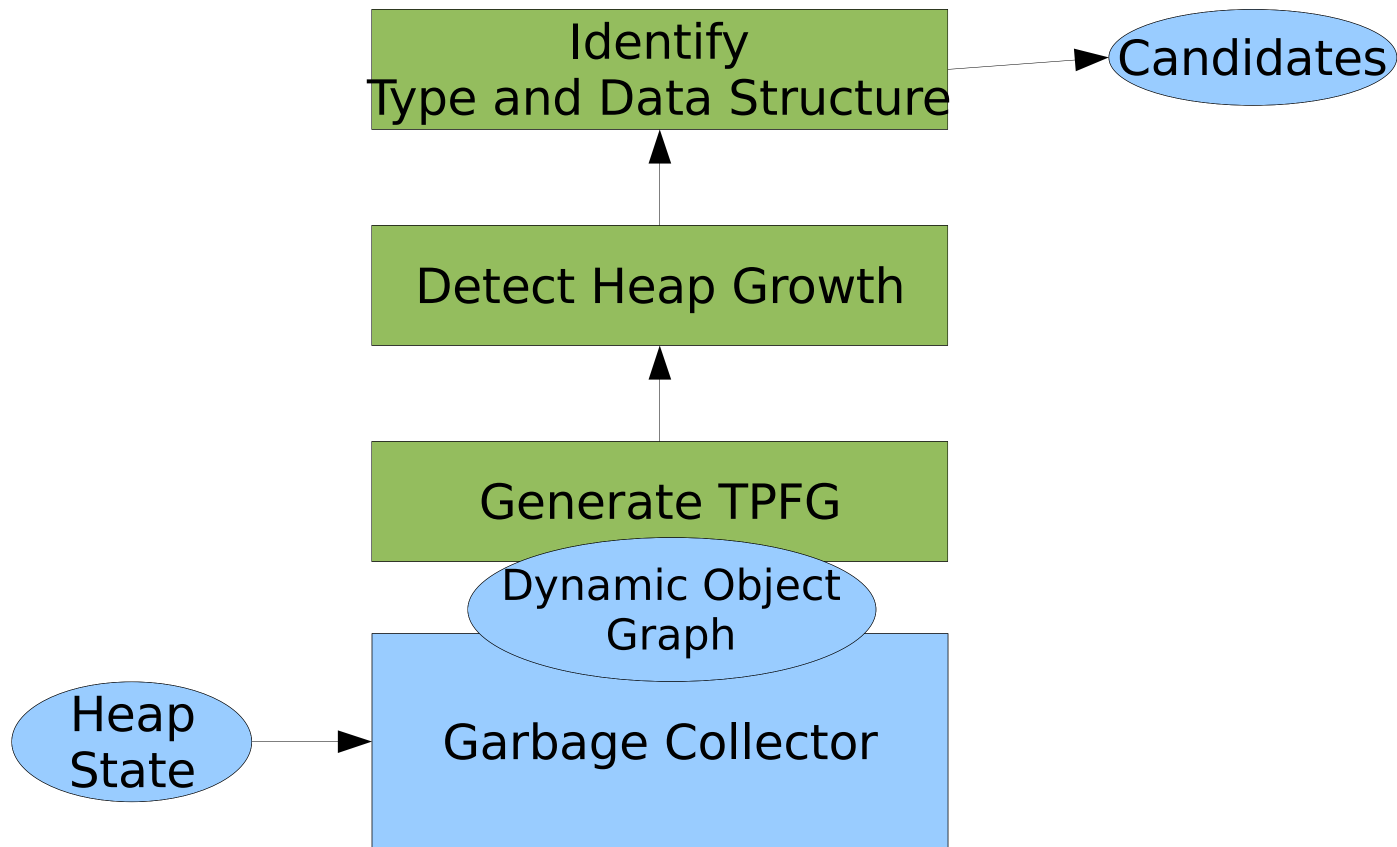
- Pointers and memory are kept for data that is not used again.
- A type of memory leak.

```
int main()
{
   int *x;
   x = malloc(sizeof(int));
   *x = 5;
   printf(“%d”, *x);
   ...           //The program continues, but x is not
                //used again.
}
```

Motivation: Garbage Collection

- Garbage collection:
  - Fixes dangling pointers
  - Fixes lost pointers
  - DOES NOT FIX unnecessary references.
- Memory leaks cause performance degradation and possible crashing.

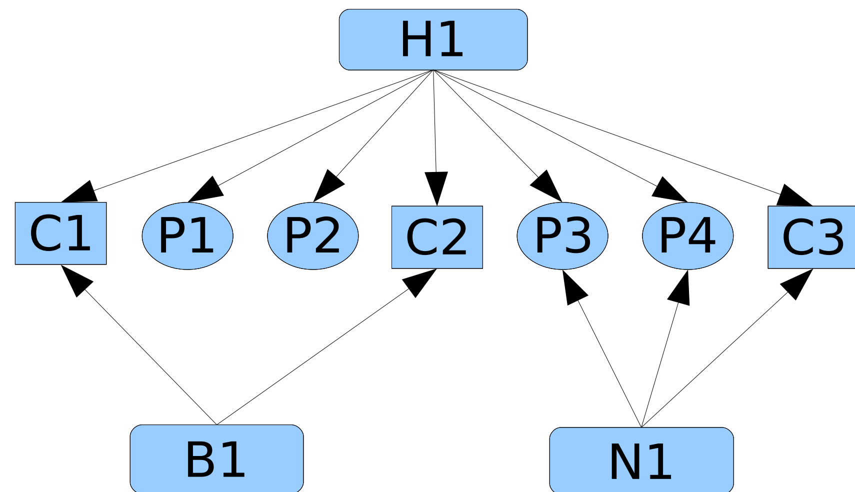
Approach: Cork



Dynamic Object Graph

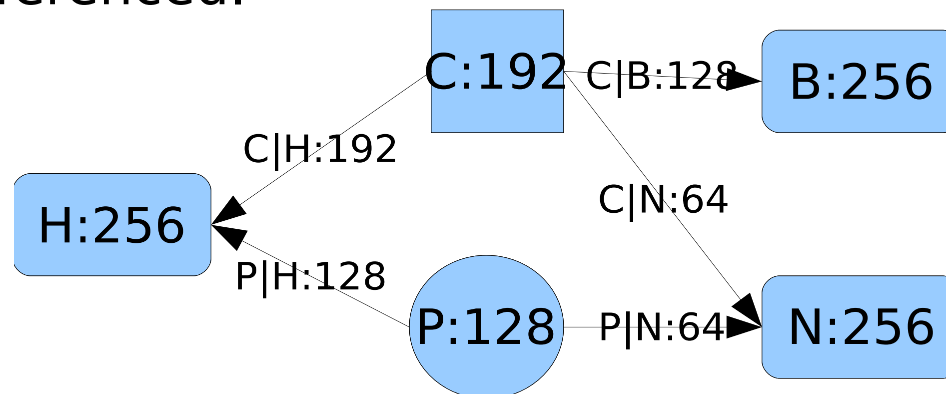
- Generated by garbage collector
- Node: A data object allocated in the heap
- Edge: A pointer from one object in the heap to another object.

Type	Symbol	Size
HashTable	H	256
Queue	N	256
Queue	B	256
Company	C	64
People	P	32



TPFG Generator

- TPFG: Type Points-From Graph
- Node: A type defined by the program
  - The number indicates the amount of space of all currently dynamically allocated objects of this type.
- Edge: Indicates one type being referenced by another type
  - The number indicates the amount of space of all the objects being referenced.



Advantages of TPFG

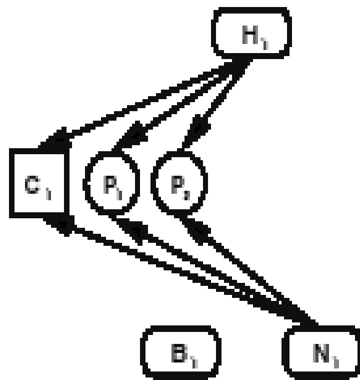
- Minimizes cost by building during the scanning phase of garbage collection.
- Uses volume instead of simple counters.
  - Helps to detect array growth when the number of arrays does not change.
- Using volume also causes larger types to be weighted heavier that smaller types.
  - Is this good?

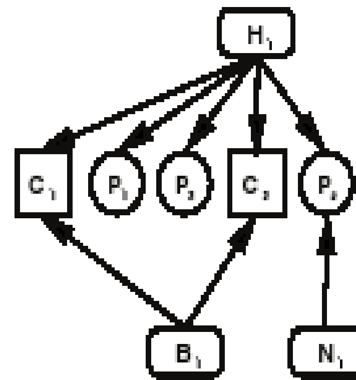
Detecting Heap Growth

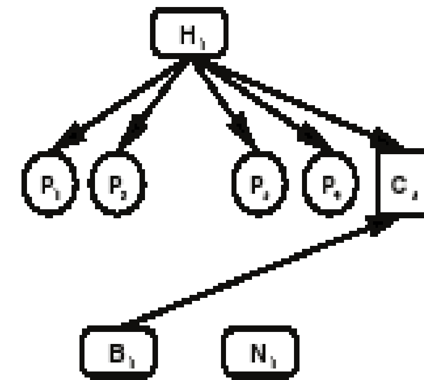
- TPFG produced each time the garbage collector executes.
- Differences between consecutive TPFGs are analyzed.
- Types that are growing are considered candidate leaks.
- Candidates are ranked using the Ratio Ranking Technique (RRT).

Heap Growth Example

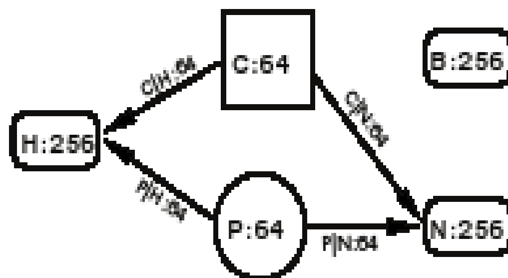
Object Points–to Graphs



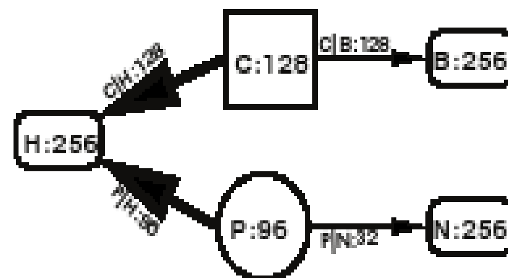




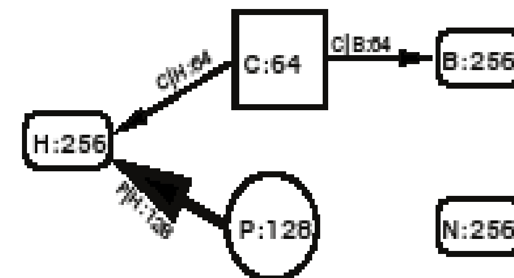
Type Points–From Graphs



(a) After collection 1



(b) After collection 2



(c) After collection 3

Ratio Ranking Technique

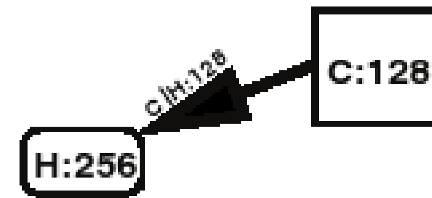
- Ranks according to ratio of volumes between two consecutive TPFGs.
- Uses a threshold value to choose candidate leaks.
- Uses a decay factor, $0 < f < 1$, to adjust for jitter.

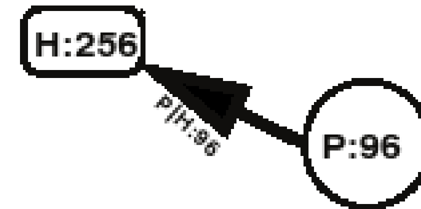
Correlating Data Structures and Allocation Sites

- Cork attempts to correlate growing nodes and edges in the TPFGs with
  - The corresponding type
  - The data structure containing that type
  - The location where the structure was allocated
- To do this, Cork produces “slices” of the TPFG.

Generating a Slice

- Start at a candidate node
- Follow growing edges
- Stop when the current node is not growing and there are no growing edges leaving the node





Efficiency and Scalability

- Cork only needs to keep the previous 4 TPFGs to be accurate.
- Cork uses the TIB (Type Information Block). TIB or an equivalent structure is required by Java and C#.
- In theory, the number of edges in a TPFG grows quadratically, but in practice it is linear.

Cork in Other Collectors

- Cork is compatible with any scanning collector.
- Cork performs its analysis only when the entire heap is scanned.
- If the entire heap is never scanned, Cork can combine data from multiple scans as long as the combined data considers the entire heap.

Cork in Other Languages

- Cork is designed to work with polymorphically typed languages
  - Java, C#
- Cork could be modified to use other types of languages
  - Other methods would be required to track global type information
  - This would increase the overhead of Cork

Outline

- Results
  - Case Studies
  - Effect of Parameters
  - Overhead
- Related Work
- Conclusions

Methodology

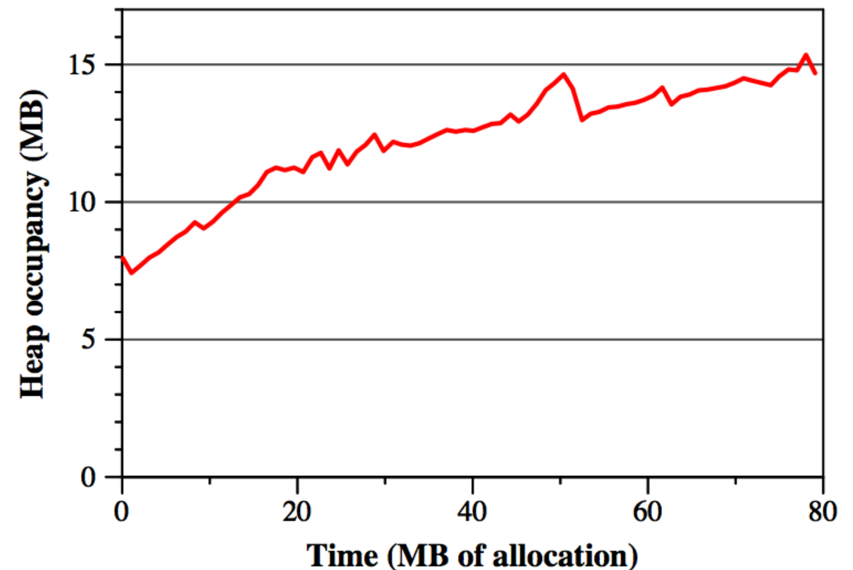
- Implementation
  - Used the MMTk
    - Memory Management Toolkit
    - Allows the testing of new garbage collection algorithms
  - Ran under Jikes RVM
    - Research Virtual Machine
    - Testbed for VM technologies

Empirical Questions

- Can Cork be used to successfully find bugs?
- Is the overhead low enough to be generally deployable?
- How do the parameters affect Cork?

Case Study – FOP

- FOP
  - Renders XSL-FO documents
  - 150k lines of code
- Problem
  - Excessive heap usage



Case Study – FOP

- Cork Reports
  - Leak report points to the ArrayList object

GC No.	Rank	Type
15	2104.17	Ljava/util/ArrayList;
	135.26	Lorg/apache/fop/fo/LengthProperty;
	115.64	Lorg/apache/fop/datatypes/AutoLength;
18	2106.28	Ljava/util/ArrayList;
21	2108.24	Ljava/util/ArrayList;

Case Study – FOP

- Cork Reports
  - ArrayList Slice Report and Allocation Sites report narrow the search

Type
Ljava/util/ArrayList;
← Lorg/apache/fop/layout/inline/WordArea;
← Ljava/lang/Object; []
← Ljava/util/ArrayList;
← Lorg/apache/fop/layout/LineArea;
← Ljava/lang/Object; []
← Ljava/util/ArrayList;
← Lorg/apache/fop/layout/inline/WordArea;
← Ljava/lang/Object; []
← Ljava/util/ArrayList;
← Lorg/apache/fop/layout/inline/InlineSpace;
← Ljava/lang/Object; []
← Ljava/util/ArrayList;
...

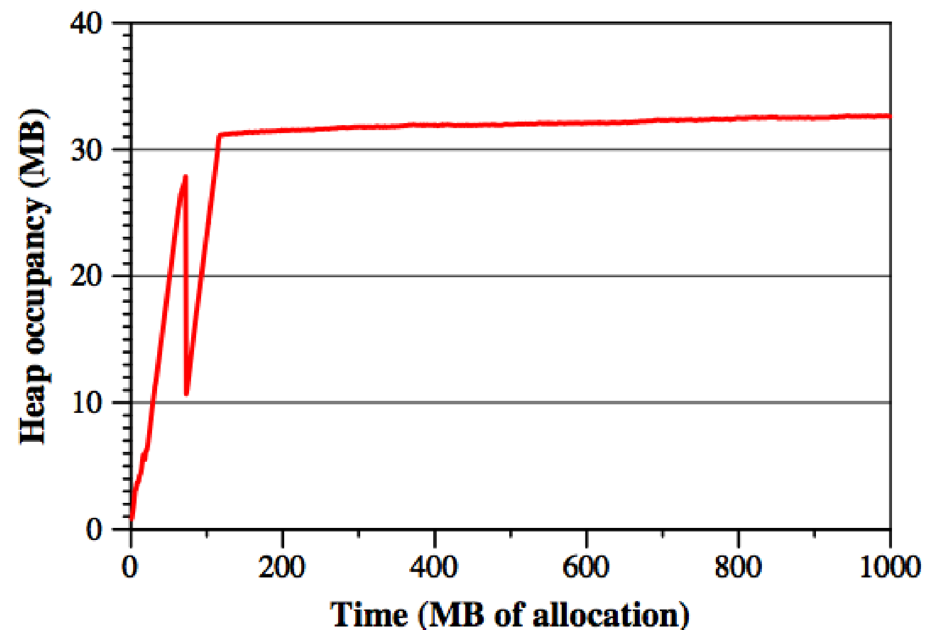
Method	**bcidx**
Ljava/util/ArrayList;	
too numerous to be useful (45)	
Lorg/apache/fop/layout/inline/WordArea;	
.../render/pdf/PDFRenderer;.renderWordArea...	355
.../render/pdf/PDFRenderer;.renderWordArea...	450
.../render/pdf/PDFRenderer;.renderWordArea...	555
.../render/pdf/PDFRenderer;.renderWordArea...	811
.../render/pdf/PDFRenderer;.renderWordArea...	820
Lorg/apache/fop/layout/LineArea;	
.../layout/BlockArea;.getCurrentLineArea...	26
.../layout/BlockArea;.createNextLineArea...	45

Case Study – FOP

- Observations
  - Heap growth was not a memory leak
    - Side-effect of XSL-FO specification
    - Developers concurred with the analysis

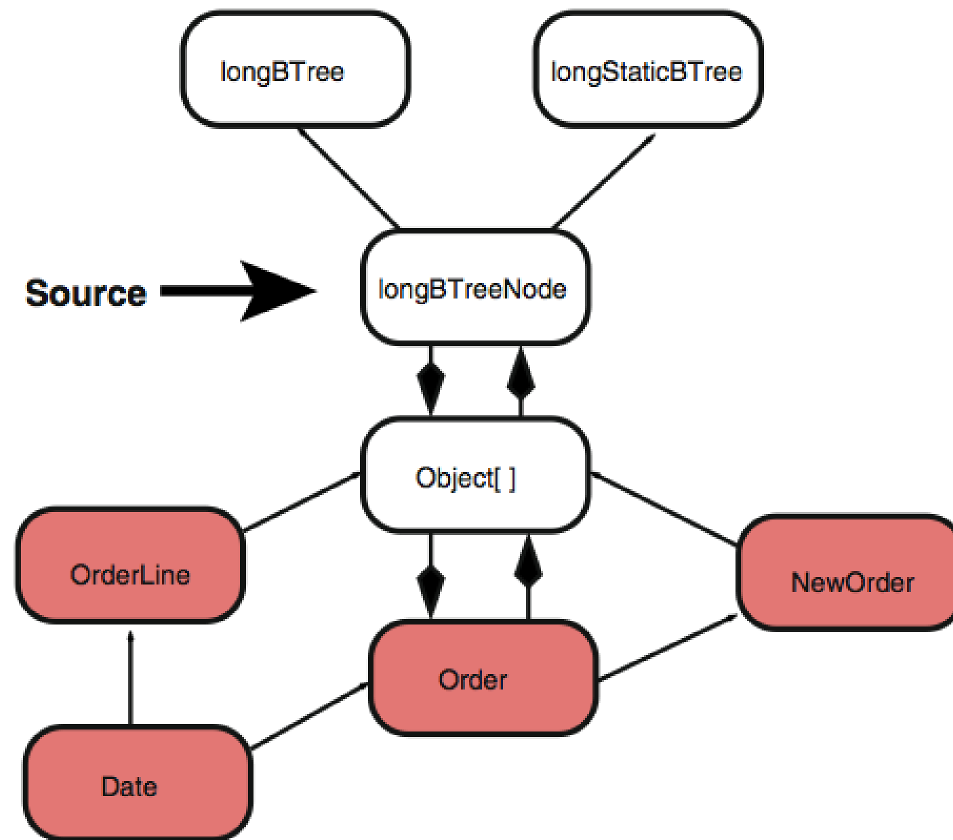
Case Study – SPECjbb2000

- SPECjbb2000
  - Business Middleware Benchmark
  - Emulates a 3-tier system with the bulk of the benchmark handling the middleware portion
  - ~30k Lines of code
- Problem
  - Known memory leak



Case Study – SPECjbb2000

- Cork Slice Report



Case Study – SPECjbb2000

- Observations
  - Orders added to Btree, never removed
- Time to Fix
  - 1 day of work to find and fix the bug

Case Study – Eclipse

- Eclipse
  - IDE/Software Framework/Kitchen Sink
  - ~2 Million Lines of code
- Problem
  - Unresolved memory leak
  - Bug #115789
- Wrote a script to induce the memory leak

Case Study – Eclipse

- Cork Slice Report

HashMap$
HashIterator
HashMap
HashMap$
HashEntry[ ]
Source
HashMap$
HashEntry
ResourceCompareInput
ResourceCompareInput$
MyDiffNode
ListenerList
ResourceCompareInput$
FilteredBufferedResourceNode
ElementTree
ArrayList
RuleBasedCollator
Folder
File
Object[ ]
ElementTree$
ChildIDsCache
Path

Case Study – Eclipse

- Observations
  - Homegrown reference counting implementation
- Time to Fix
  - 3.5 days of work to find and fix the bug
  - The authors weren't familiar with the codebase

Overhead

- Ran a series of 15 benchmarks
  - SPECjvm, DaCapo, SPECjbb2000, Eclipse
- Performance Overhead
  - 1.4-4% longer execution on average
  - The worst case saw ~18% longer execution time
- Space Overhead
  - 0.145% extra heap usage on average
  - The worst case saw 0.5% extra heap usage

Effect of Threshold Parameters

Threshold: 0/50		0%	10%	25%	50%		0%	10%	25%	50%
False Positives		930	443	213	89		176	42	8	1
False Negatives		0	0	0	0		0	0	0	2
Correct		7	7	7	7		7	7	7	4

Threshold: 100/200		0%	10%	**25%**	50%		0%	10%	25%	50%
False Positives		102	13	**0**	0		67	3	0	0
False Negatives		0	0	**0**	5		0	0	1	5
Correct		7	7	**7**	2		7	7	6	2

- Tunes Cork's sensitivity to different patterns in heap usage
- If too low, it will generate false positives
- If too high, it will miss leaks

Effect of Decay Factor

	0%	5%	10%	**15%**	20%	25%
False Positives	0	0	0	**0**	0	2
False Negatives	5	0	0	**0**	0	0
Correct	2	7	7	**7**	7	7

- Prevents CORK from missing objects due to temporary decreases in growth in an otherwise growing heap usage
- If too low, it will miss valid leaks
- If too high, it will generate false positives

Other Approaches

- Compile-time analysis
  - Can be used to find double frees and missing frees
  - Heine and Lam, 2003
- Offline analysis
  - Heap differencing
    - De Pauw, et al., 1998, De Pauw, et al., 2000
  - Allocation tracking
    - Hastings and Joyce 1992, Serrano and Baum 2000, Shaham, et al. 2000, Campan, et al. 2002

Other Approaches

- Online analysis
  - Expected lifetime tracking
    - Qin et al. 2002
  - Tracking object “staleness”
    - Chilimbi and Hauswirth 2004, Bond and McKinley 2006
- Leakbot
  - Online leak detection system that analyzes an application's heap on a separate processor
  - Mitchell and Sevitzky 2003, Gupta and Palanki 2005

Conclusions

- Cork is a low-overhead memory leak detector
  - Could be deployed on production applications
- Effectively identifies the source of the leak
  - Few false positives if configured properly