Addressing Software Challenges for Smart Infrastructures using Software Persistent Memory

Raju Rangaswami

School of Computing and Information Sciences
College of Engineering and Computing
Florida International University

IFIP 10.4 WG Meeting
January 13th, 2011

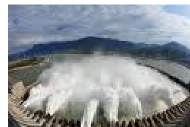
Requrements for Smart Infrastructures

Requrements for Smart Infrastructures

Self-management

- Self-management is based on intelligence

Requrements for Smart Infrastructures



Self-management

- Self-management is based on intelligence
- Intelligence demands memory

Requrements for Smart Infrastructures

Self-management

- Self-management is based on intelligence
- Intelligence demands memory
- Fault-tolerant intelligence demands *persistent memory*

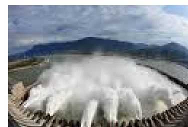
Simulating Smart Infrastructures

Simulating Smart Infrastructures

Large-scale simulations

- Massively parallel and long-running (days, weeks)

Simulating Smart Infrastructures



Large-scale simulations

- Massively parallel and long-running (days, weeks)
- Demand tolerance to software, system, or power failure

Simulating Smart Infrastructures

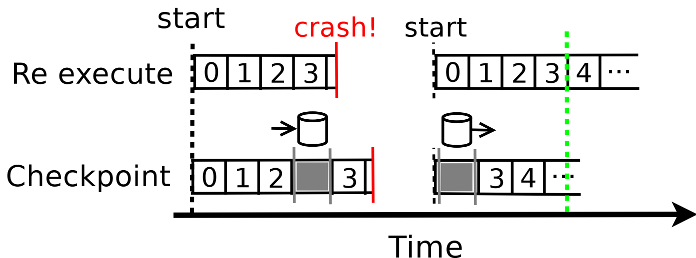
Large-scale simulations

- Massively parallel and long-running (days, weeks)
- Demand tolerance to software, system, or power failure
- Need for *persistent memory*

The Problem

Software, System, or Power Failure

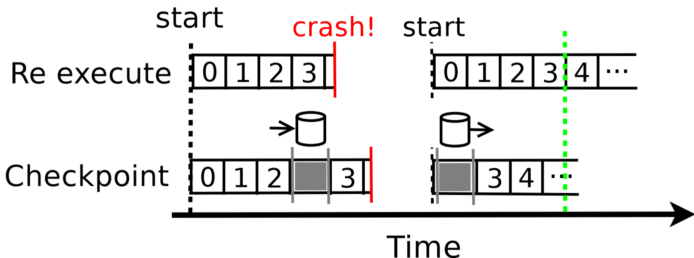
- Solution I: Software re-execution
- Solution II: Recover from a full system or process checkpoint



The Problem

Software, System, or Power Failure

- Solution I: Software re-execution
- Solution II: Recover from a full system or process checkpoint

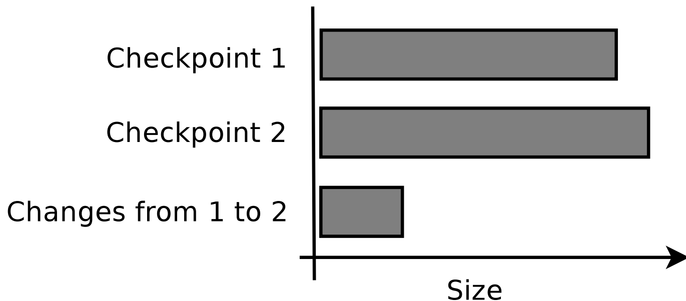


These solutions still incur substantial overhead.

Problems With Existing Checkpointing Solutions

Program characteristics

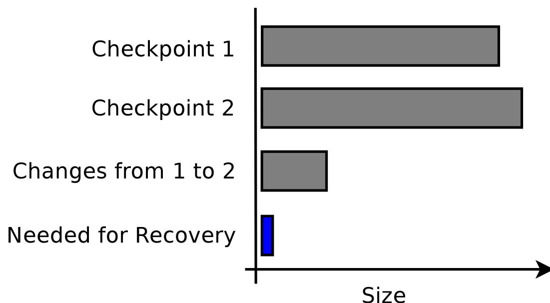
- Programs usually modify a subset of data



Problems With Existing Checkpointing Solutions

Program characteristics

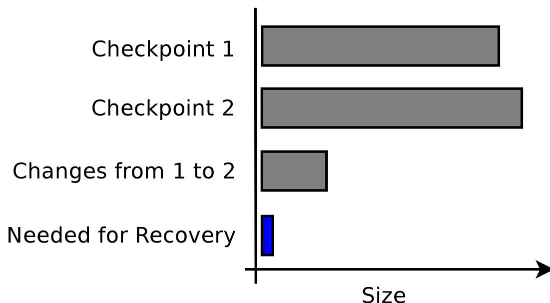
- Programs usually modify only a subset of data at any time
- Programs only need a subset of data for recovery



Problems With Existing Checkpointing Solutions

Program characteristics

- Programs usually modify only a subset of data at any time
- Programs only need a subset of data for recovery



Manual tracking is cumbersome, inefficient, and less portable.

Making a Balanced BTree Persistent

The developer would need to:

- Know all data to persist (*cumbersome*)

Making a Balanced BTree Persistent

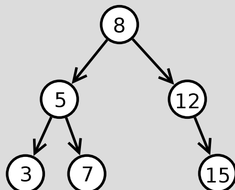
The developer would need to:

- Know all data to persist (*cumbersome*)
- Keep track of changes to data (*cumbersome*)

Making a Balanced BTree Persistent

The developer would need to:

- Know all data to persist (*cumbersome*)
- Keep track of changes to data (*cumbersome*)

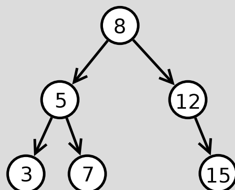


Initial Tree

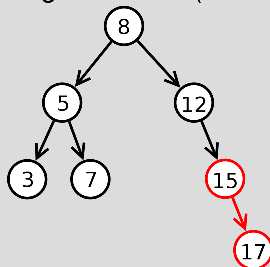
Making a Balanced BTree Persistent

The developer would need to:

- Know all data to persist (*cumbersome*)
- Keep track of changes to data (*cumbersome*)



Initial Tree

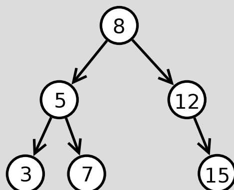


Before Balancing

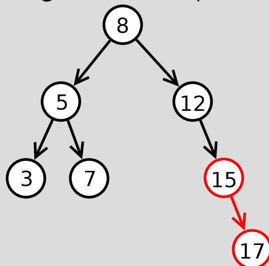
Making a Balanced BTree Persistent

The developer would need to:

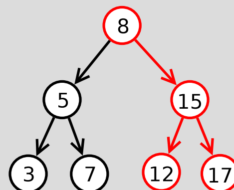
- Know all data to persist (*cumbersome*)
- Keep track of changes to data (*cumbersome*)



Initial Tree



Before Balancing

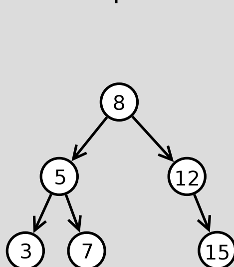


Final Tree

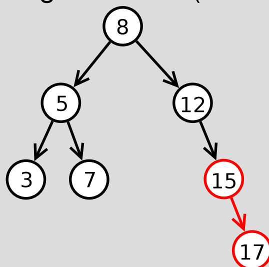
Making a Balanced BTree Persistent

The developer would need to:

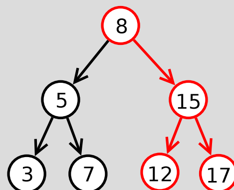
- Know all data to persist (*cumbersome*)
- Keep track of changes to data (*cumbersome*)



Initial Tree



Before Balancing



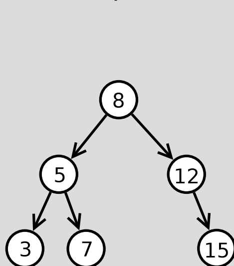
Final Tree

- Keep track of complex dependencies (*cumbersome*)

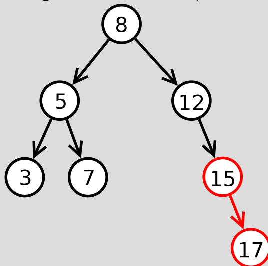
Making a Balanced BTree Persistent

The developer would need to:

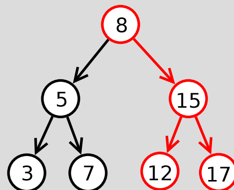
- Know all data to persist (*cumbersome*)
- Keep track of changes to data (*cumbersome*)



Initial Tree



Before Balancing



Final Tree

- Keep track of complex dependencies (*cumbersome*)
- Serialize arbitrary data structures (*cumbersome*)

Software Persistent Memory (*SoftPM*)

Applications allocate, and interact with, persistent memory in much the same way as they work with volatile memory.

SoftPM Goals

Developer specifies:

- Top-level data structures to persist
- When to create checkpoints
- How to recover execution from restored data in memory

SoftPM Goals

Developer specifies:

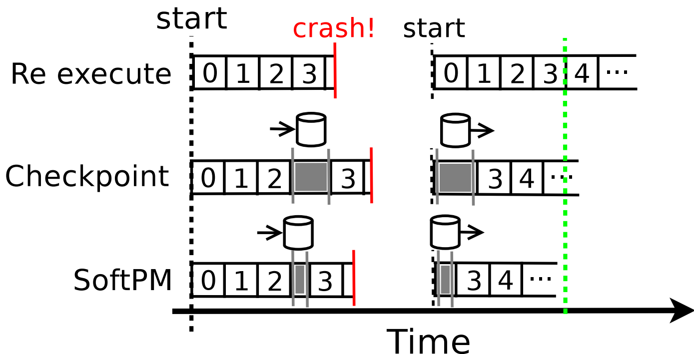
- Top-level data structures to persist
- When to create checkpoints
- How to recover execution from restored data in memory

The solution will:

- Require minimal source code changes
- Automatically detect data to be checkpointed
- Track changes to data
- Minimize checkpoint creation and restoration time

SoftPM Basic Approach

- Implements application directed checkpointing
- Data to be checkpointed is automatically discovered
- Fast checkpoints and recovery



The Container Abstraction

Simple container

- Self-describing unit of persistent memory
- One or more containers per application of arbitrary size
- Applications create a container and specify a root structure
- All data reachable from the root automatically discovered
- A container is restoreable with a single operation

The Container Abstraction

Simple container

- Self-describing unit of persistent memory
- One or more containers per application of arbitrary size
- Applications create a container and specify a root structure
- All data reachable from the root automatically discovered
- A container is restoreable with a single operation

Versioned container

- Each persistence point creates new container version
- Past versions are restoreable with a single operation
- Past versions are browsable
- Container version is branch-able

Using *SoftPM* : An Example

```
typedef struct l {
  int v;
  struct l *n;
} list;
```

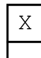
- ```
  a() {
    list *l = malloc(...);
    l->v = X;
    list *e1 = malloc(...);
    e1->v = Y;
    l->n = e1;
    list *e2 = malloc(...);
    e2->v = Z;
    e1->n = e2;
    ⋮
  }
  ```
  ```

Using *SoftPM* : An Example

```
typedef struct l {
  int v;
  struct l *n;
} list;
a() {
▶ list *l = malloc(...);
  l->v = X;
  list *e1 = malloc(...);
  e1->v = Y;
  l->n = e1;
  list *e2 = malloc(...);
  e2->v = Z;
  e1->n = e2;
  ⋮
}
```

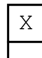
Using *SoftPM* : An Example

```
typedef struct l {
  int v;
  struct l *n;
} list;
a() {
  list *l = malloc(...);
► l->v = X;
  list *e1 = malloc(...);
  e1->v = Y;
  l->n = e1;
  list *e2 = malloc(...);
  e2->v = Z;
  e1->n = e2;
  ⋮
}
```



Using *SoftPM* : An Example

```
typedef struct l {
  int v;
  struct l *n;
} list;
a() {
  list *l = malloc(...);
  l->v = X;
  list *e1 = malloc(...);
  e1->v = Y;
  l->n = e1;
  list *e2 = malloc(...);
  e2->v = Z;
  e1->n = e2;
  ⋮
}
```



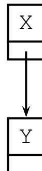
Using *SoftPM* : An Example

```
typedef struct l {
  int v;
  struct l *n;
} list;
a() {
  list *l = malloc(...);
  l->v = X;
  list *e1 = malloc(...);
```

```
  e1->v = Y;
  l->n = e1;
  list *e2 = malloc(...);
  e2->v = Z;
  e1->n = e2;
  ⋮
}
```

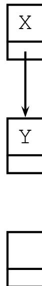
Using *SoftPM* : An Example

```
typedef struct l {
  int v;
  struct l *n;
} list;
a() {
  list *l = malloc(...);
  l->v = X;
  list *e1 = malloc(...);
  e1->v = Y;
▶ l->n = e1;
  list *e2 = malloc(...);
  e2->v = Z;
  e1->n = e2;
  ⋮
}
```



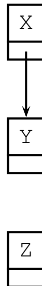
Using *SoftPM* : An Example

```
typedef struct l {
  int v;
  struct l *n;
} list;
a() {
  list *l = malloc(...);
  l->v = X;
  list *e1 = malloc(...);
  e1->v = Y;
  l->n = e1;
▶ list *e2 = malloc(...);
  e2->v = Z;
  e1->n = e2;
  ⋮
}
```



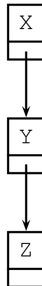
Using *SoftPM* : An Example

```
typedef struct l {
  int v;
  struct l *n;
} list;
a() {
  list *l = malloc(...);
  l->v = X;
  list *e1 = malloc(...);
  e1->v = Y;
  l->n = e1;
  list *e2 = malloc(...);
▶ e2->v = Z;
  e1->n = e2;
  ⋮
}
```



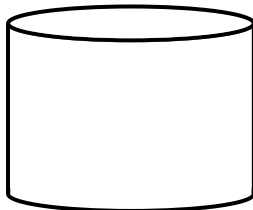
Using *SoftPM* : An Example

```
typedef struct l {
  int v;
  struct l *n;
} list;
a() {
  list *l = malloc(...);
  l->v = X;
  list *e1 = malloc(...);
  e1->v = Y;
  l->n = e1;
  list *e2 = malloc(...);
  e2->v = Z;
▶ e1->n = e2;
  ⋮
}
```



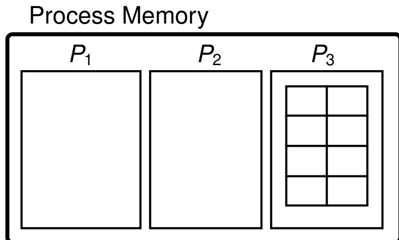
Creating a Persistence Point

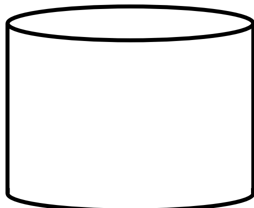
```
struct pcont {
  list *L;
} C;
a() {
  if (!(cid = pCRestore(&C)))
    cid = pCAlloc(&C);
  list *l = malloc(...);
  C->L = l;
  l->v = X;
  list *e1 = malloc(...);
  e1->v = Y;
  l->n = e1;
  list *e2 = malloc(...);
  e2->v = Z;
  e1->n = e2;
  pPoint(cid);
  }
  ⋮
}
```



Creating a Persistence Point

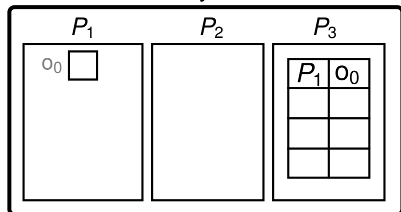
```
struct pcont {
  list *L;
} C;
a() {
▶ if (!(cid = pCRestore(&C)))
    cid = pCAlloc(&C);
  list *l = malloc(...);
  C->L = l;
  l->v = X;
  list *e1 = malloc(...);
  e1->v = Y;
  l->n = e1;
  list *e2 = malloc(...);
  e2->v = Z;
  e1->n = e2;
  pPoint(cid);
  }
  ⋮
}
```

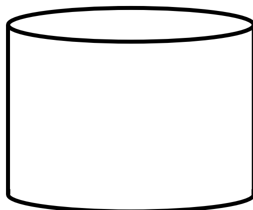




Creating a Persistence Point

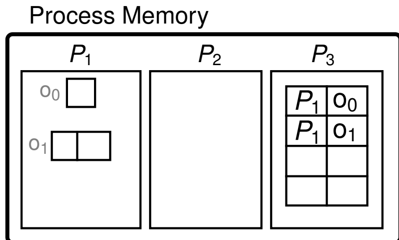
```
struct pcont {
  list *L;
} C;
a() {
  if (!(cid = pCRestore(&C)))
    cid = pCAlloc(&C);
  list *l = malloc(...);
  C->L = l;
  l->v = X;
  list *e1 = malloc(...);
  e1->v = Y;
  l->n = e1;
  list *e2 = malloc(...);
  e2->v = Z;
  e1->n = e2;
  pPoint(cid);
  }
  ⋮
}
```

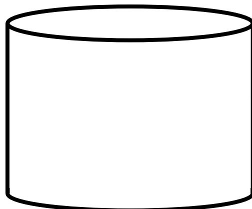




Creating a Persistence Point

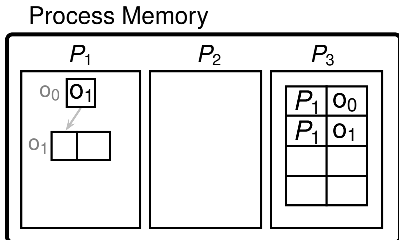
```
struct pcont {
  list *L;
} C;
a() {
  if (!(cid = pCRestore(&C)))
    cid = pCAlloc(&C);
  list *l = malloc(...);
  C->L = l;
  l->v = X;
  list *e1 = malloc(...);
  e1->v = Y;
  l->n = e1;
  list *e2 = malloc(...);
  e2->v = Z;
  e1->n = e2;
  pPoint(cid);
  }
  ⋮
}
```

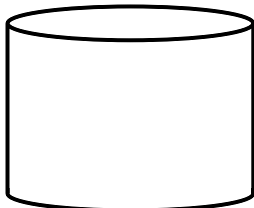




Creating a Persistence Point

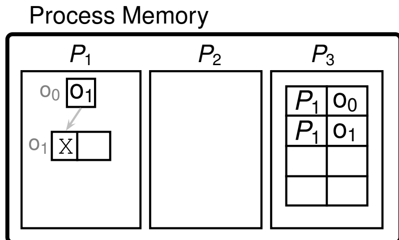
```
struct pcont {
  list *L;
} C;
a() {
  if (!(cid = pCRestore(&C)))
    cid = pCAlloc(&C);
  list *l = malloc(...);
▶ C->L = l;
  l->v = X;
  list *e1 = malloc(...);
  e1->v = Y;
  l->n = e1;
  list *e2 = malloc(...);
  e2->v = Z;
  e1->n = e2;
  pPoint(cid);
  }
  ⋮
}
```

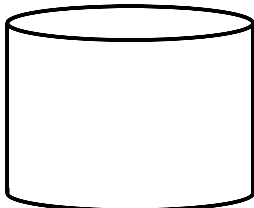




Creating a Persistence Point

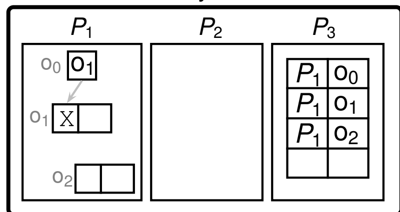
```
struct pcont {
  list *L;
} C;
a() {
  if (!(cid = pCRestore(&C)))
    cid = pCAlloc(&C);
  list *l = malloc(...);
  C->L = l;
▶ l->v = X;
  list *e1 = malloc(...);
  e1->v = Y;
  l->n = e1;
  list *e2 = malloc(...);
  e2->v = Z;
  e1->n = e2;
  pPoint(cid);
  }
  ⋮
}
```

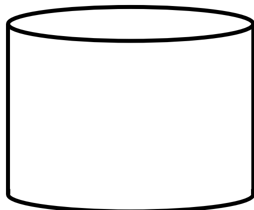




Creating a Persistence Point

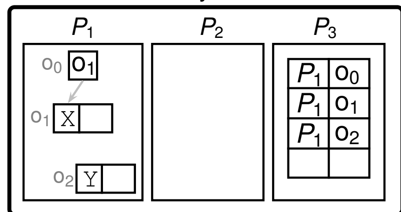
```
struct pcont {
  list *L;
} C;
a() {
  if (!(cid = pCRestore(&C)))
    cid = pCAlloc(&C);
  list *l = malloc(...);
  C->L = l;
  l->v = X;
▶ list *e1 = malloc(...);
  e1->v = Y;
  l->n = e1;
  list *e2 = malloc(...);
  e2->v = Z;
  e1->n = e2;
  pPoint(cid);
  }
  ⋮
}
```

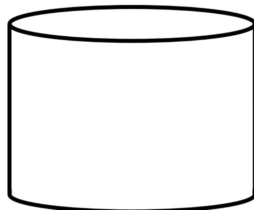




Creating a Persistence Point

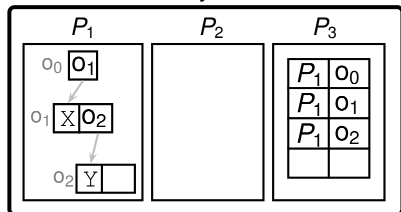
```
struct pcont {
  list *L;
} C;
a() {
  if (!(cid = pCRestore(&C)))
    cid = pCAlloc(&C);
  list *l = malloc(...);
  C->L = l;
  l->v = X;
  list *e1 = malloc(...);
▶ e1->v = Y;
  l->n = e1;
  list *e2 = malloc(...);
  e2->v = Z;
  e1->n = e2;
  pPoint(cid);
  }
  ⋮
}
```

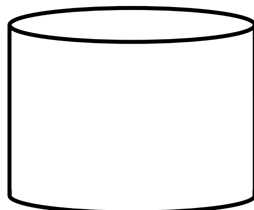




Creating a Persistence Point

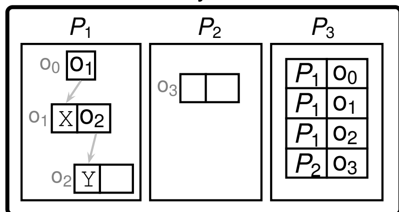
```
struct pcont {
  list *L;
} C;
a() {
  if (!(cid = pCRestore(&C)))
    cid = pCAlloc(&C);
  list *l = malloc(...);
  C->L = l;
  l->v = X;
  list *e1 = malloc(...);
  e1->v = Y;
▶ l->n = e1;
  list *e2 = malloc(...);
  e2->v = Z;
  e1->n = e2;
  pPoint(cid);
  }
  ⋮
}
```

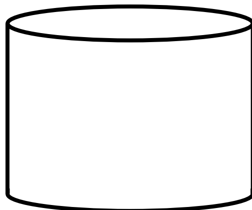




Creating a Persistence Point

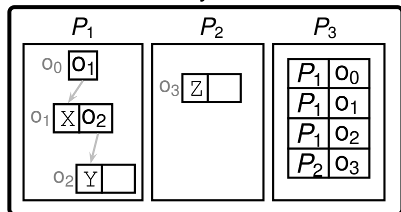
```
struct pcont {
  list *L;
} C;
a() {
  if (!(cid = pCRestore(&C)))
    cid = pCAlloc(&C);
  list *l = malloc(...);
  C->L = l;
  l->v = X;
  list *e1 = malloc(...);
  e1->v = Y;
  l->n = e1;
▶ list *e2 = malloc(...);
  e2->v = Z;
  e1->n = e2;
  pPoint(cid);
  }
  ⋮
}
```

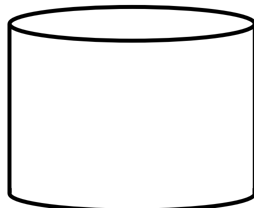




Creating a Persistence Point

```
struct pcont {
  list *L;
} C;
a() {
  if (!(cid = pCRestore(&C)))
    cid = pCAlloc(&C);
  list *l = malloc(...);
  C->L = l;
  l->v = X;
  list *e1 = malloc(...);
  e1->v = Y;
  l->n = e1;
  list *e2 = malloc(...);
▶ e2->v = Z;
  e1->n = e2;
  pPoint(cid);
  }
  ⋮
}
```

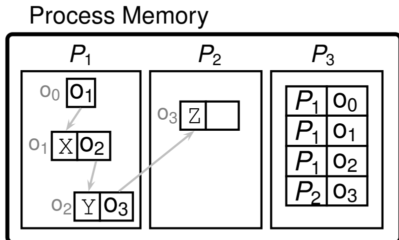


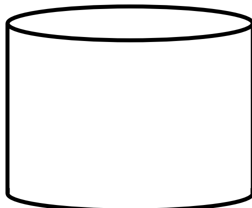


Persistent Medium

Creating a Persistence Point

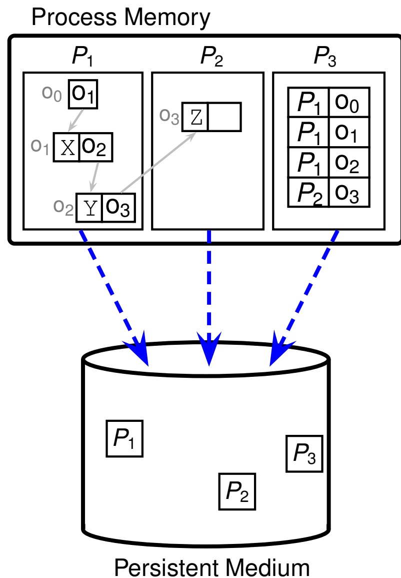
```
struct pcont {
  list *L;
} C;
a() {
  if (!(cid = pCRestore(&C)))
    cid = pCAlloc(&C);
  list *l = malloc(...);
  C->L = l;
  l->v = X;
  list *e1 = malloc(...);
  e1->v = Y;
  l->n = e1;
  list *e2 = malloc(...);
  e2->v = Z;
▶ e1->n = e2;
  pPoint(cid);
  }
  ⋮
}
```





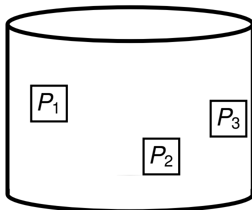
Creating a Persistence Point

```
struct pcont {
  list *L;
} C;
a() {
  if (!(cid = pCRestore(&C)))
    cid = pCAlloc(&C);
  list *l = malloc(...);
  C->L = l;
  l->v = X;
  list *e1 = malloc(...);
  e1->v = Y;
  l->n = e1;
  list *e2 = malloc(...);
  e2->v = Z;
  e1->n = e2;
▶ pPoint(cid);
  }
  ⋮
}
```



Restoring a Persistence Point

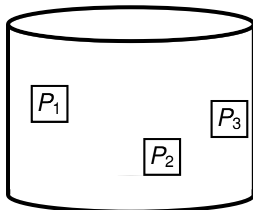
```
struct pcont {
  list *l;
} C;
a() {
  if (!(cid = pCRestore(&C)))
     ⋮
  }
  ⋮
}
```



Restoring a Persistence Point

```
struct pcont {
  list *l;
} C;
a() {
▶ if (!(cid = pCRestore(&C)))
      ⋮
  }
  ⋮
}
```

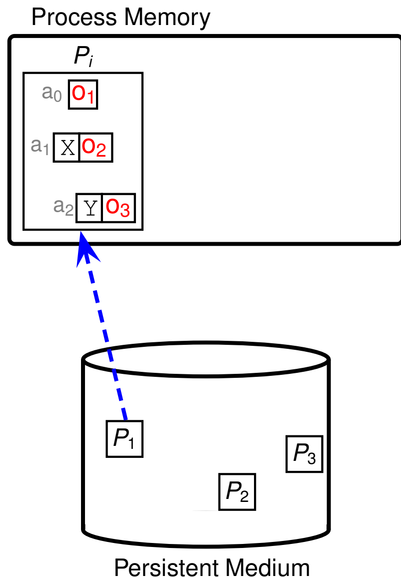
Prev	Curr



Restoring a Persistence Point

```
struct pcont {
  list *l;
} C;
a() {
  if (!(cid = pCRestore(&C)))
     ⋮
  }
  ⋮
}
```

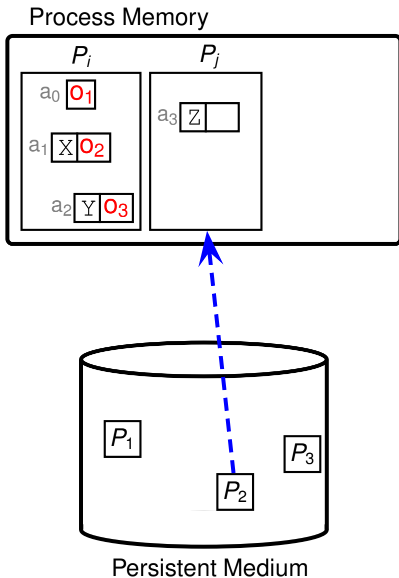
Prev	Curr
P_1	P_i



Restoring a Persistence Point

```
struct pcont {
  list *l;
} C;
a() {
▶ if (!(cid = pCRestore(&C)))
     ⋮
  }
  ⋮
}
```

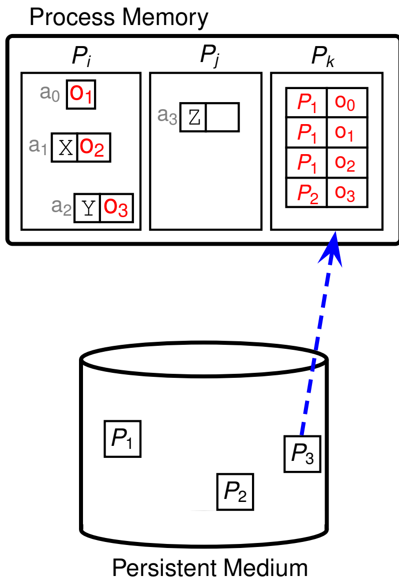
Prev	Curr
P_1	P_i
P_2	P_j



Restoring a Persistence Point

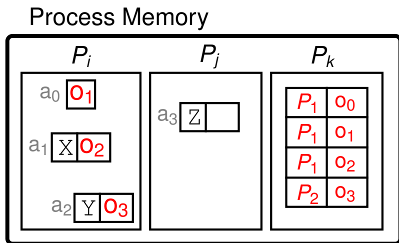
```
struct pcont {
  list *l;
} C;
a() {
►  if (!(cid = pCRestore(&C)))
      ⋮
   }
   ⋮
}
```

Prev	Curr
P_1	P_i
P_2	P_j
P_3	P_k

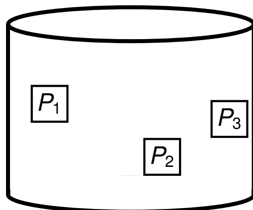


Restoring a Persistence Point

```
struct pcont {
  list *l;
} C;
a() {
▶ if (!(cid = pCRestore(&C)))
      ⋮
  }
  ⋮
}
```



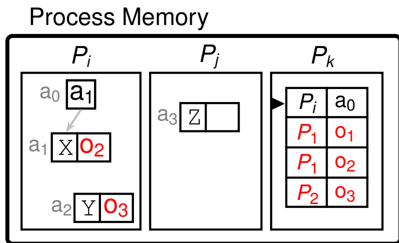
Prev	Curr
▶ P_1	P_i
P_2	P_j
P_3	P_k



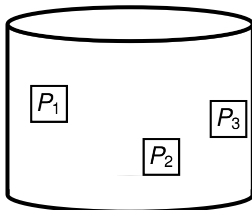
Persistent Medium

Restoring a Persistence Point

```
struct pcont {
  list *l;
} C;
a() {
  if (!(cid = pCRestore(&C)))
    ⋮
  }
  ⋮
}
```



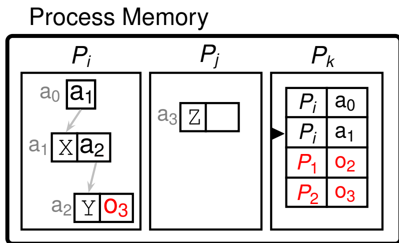
Prev	Curr
P_1	P_i
P_2	P_j
P_3	P_k



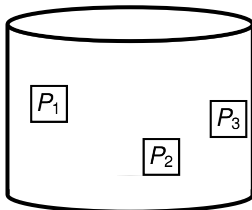
Persistent Medium

Restoring a Persistence Point

```
struct pcont {
  list *l;
} C;
a() {
►  if (!(cid = pCRestore(&C)))
      ⋮
   }
   ⋮
}
```

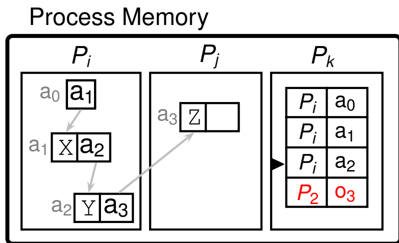


Prev	Curr
► P_1	P_i
P_2	P_j
P_3	P_k

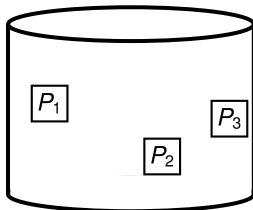


Restoring a Persistence Point

```
struct pcont {
  list *l;
} C;
a() {
▶  if (!(cid = pCRestore(&C)))
      ⋮
   }
   ⋮
}
```

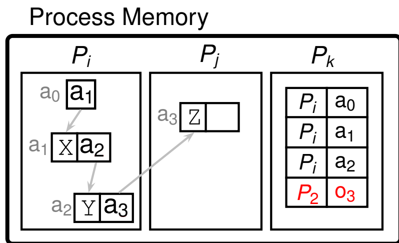


Prev	Curr
▶ P_1	P_i
P_2	P_j
P_3	P_k

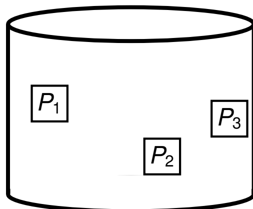


Restoring a Persistence Point

```
struct pcont {
  list *l;
} C;
a() {
►  if (!(cid = pCRestore(&C)))
      ⋮
   }
   ⋮
}
```



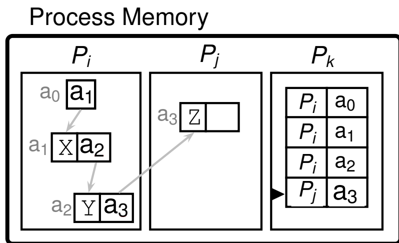
Prev	Curr
P_1	P_i
► P_2	P_j
P_3	P_k



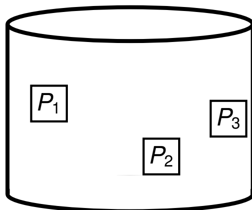
Persistent Medium

Restoring a Persistence Point

```
struct pcont {
  list *l;
} C;
a() {
▶  if (!(cid = pCRestore(&C)))
      ⋮
   }
   ⋮
}
```



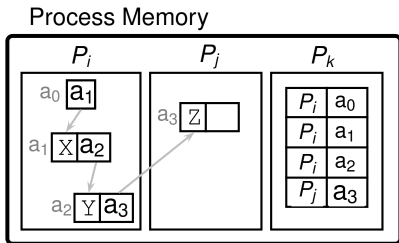
Prev	Curr
P_1	P_i
▶ P_2	P_j
P_3	P_k



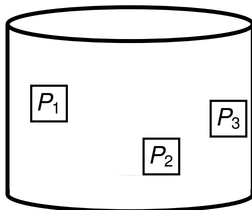
Persistent Medium

Restoring a Persistence Point

```
struct pcont {
  list *l;
} C;
a() {
►  if (!(cid = pCRestore(&C)))
      ⋮
   }
   ⋮
}
```



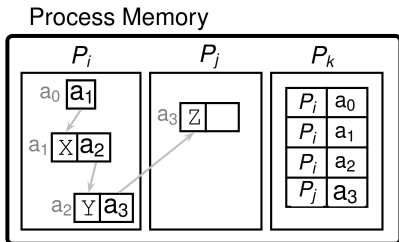
Prev	Curr
P_1	P_i
P_2	P_j
► P_3	P_k



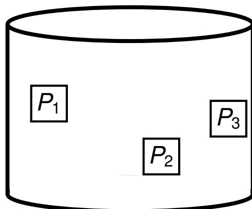
Persistent Medium

Restoring a Persistence Point

```
struct pcont {
  list *l;
} C;
a() {
  if (!(cid = pCRestore(&C)))
    ⋮
  }
  ⋮
}
```



Prev	Curr
P_1	P_i
P_2	P_j
P_3	P_k

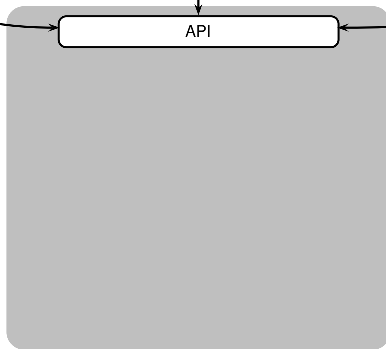


Persistent Medium

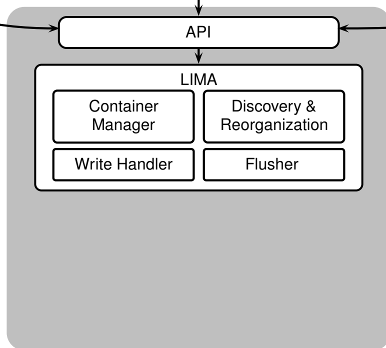
SoftPM Architecture

Recoverability Record-Replay Execution Branching

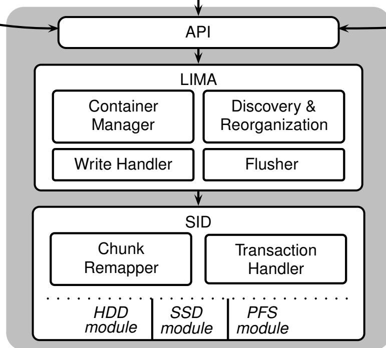
SoftPM Architecture



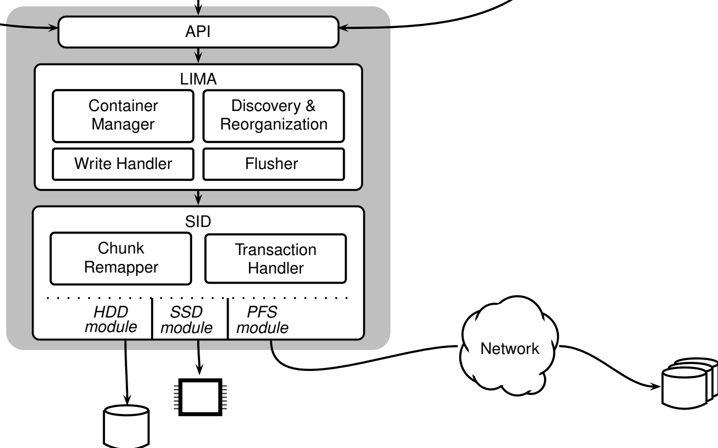
SoftPM Architecture



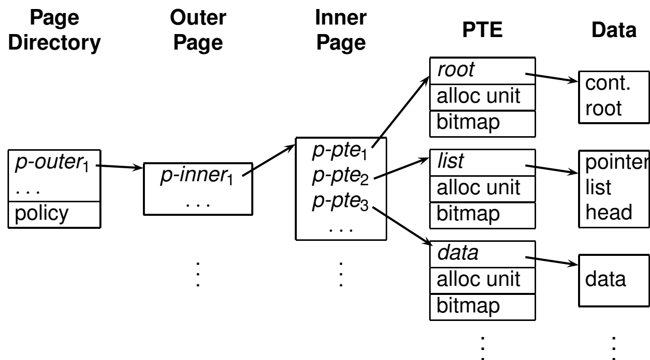
SoftPM Architecture



SoftPM Architecture



LIMA: Container Page Table



LIMA: Container Manager

Container Allocation

- Initialize container metadata
- Create container page table
- Initialize container pointer list

LIMA: Container Manager

Container Allocation

- Initialize container metadata
- Create container page table
- Initialize container pointer list

Container Restoration

- Load container data
- Fix pointers
- Return container root structure

LIMA: Discovery & Reorganization

Container Discovery

- Recursively follow pointers starting from container root
- Create persistent memory transitive closure set

LIMA: Discovery & Reorganization

Container Discovery

- Recursively follow pointers starting from container root
- Create persistent memory transitive closure set

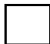
Memory Reorganization

- Memory allocation is managed by *SoftPM*
- Memory pages classified either as persistent or volatile
- Fix back-references after moving data

Automatically Discovering Data

We divide memory into 2 regions: **volatile** and **persistent.**
Hence, there can be 4 types of pointers:

Volatile



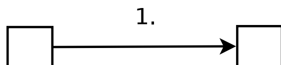
Persistent

Automatically Discovering Data

We divide memory into 2 regions: **volatile** and **persistent.**
Hence, there can be 4 types of pointers:

1. volatile $\rightarrow$ volatile

Volatile

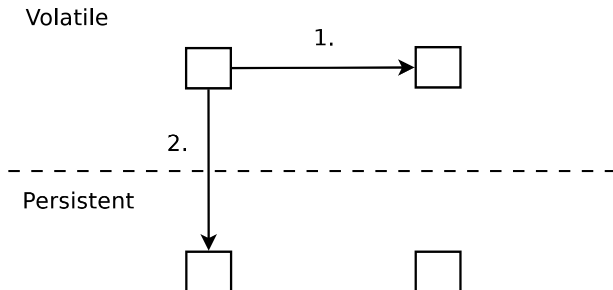


Persistent

Automatically Discovering Data

We divide memory into 2 regions: **volatile** and **persistent.**
Hence, there can be 4 types of pointers:

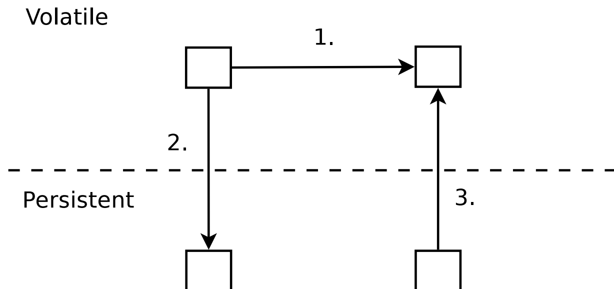
1. volatile → volatile
2. volatile → persistent



Automatically Discovering Data

We divide memory into 2 regions: **volatile** and **persistent.**
Hence, there can be 4 types of pointers:

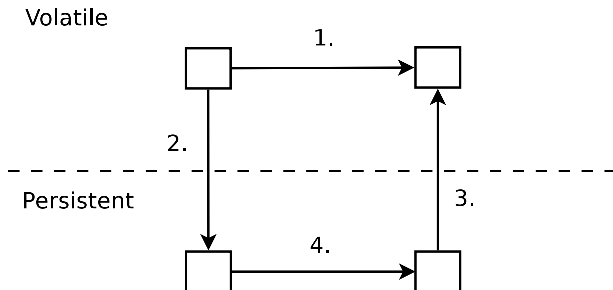
1. volatile → volatile
2. volatile → persistent
3. persistent → volatile



Automatically Discovering Data

We divide memory into 2 regions: **volatile** and **persistent.**
Hence, there can be 4 types of pointers:

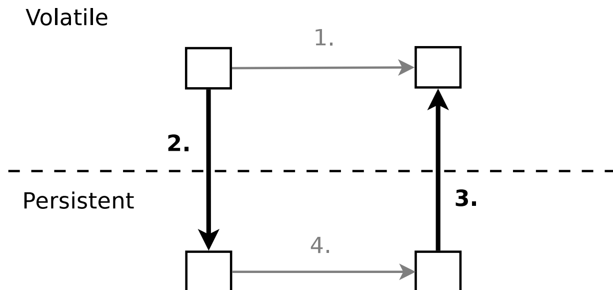
1. volatile → volatile
2. volatile → persistent
3. persistent → volatile
4. persistent → persistent



Automatically Discovering Data

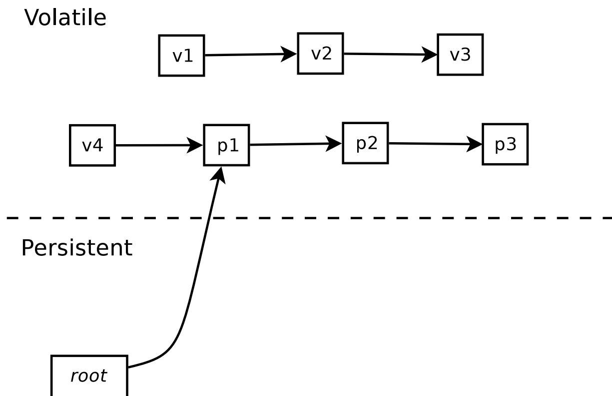
We divide memory into 2 regions: **volatile** and **persistent.**
Hence, there can be 4 types of pointers:

1. volatile → volatile
2. volatile → persistent ✓
3. persistent → volatile ✓
4. persistent → persistent



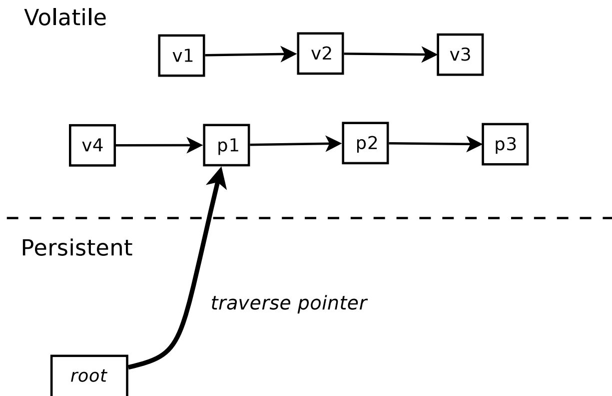
Automatically Discovering Data

Initially all memory is volatile except a *root* structure.



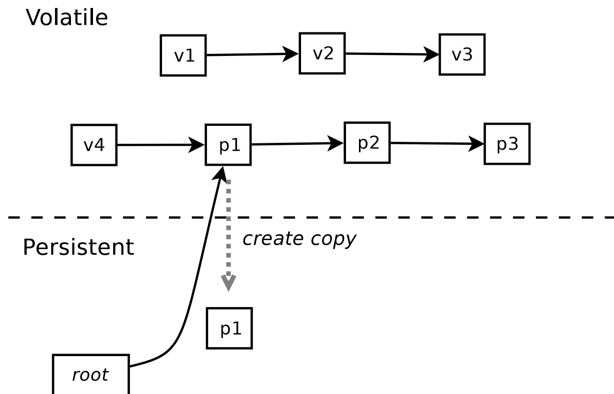
Automatically Discovering Data

Initially all memory is volatile except a *root* structure.



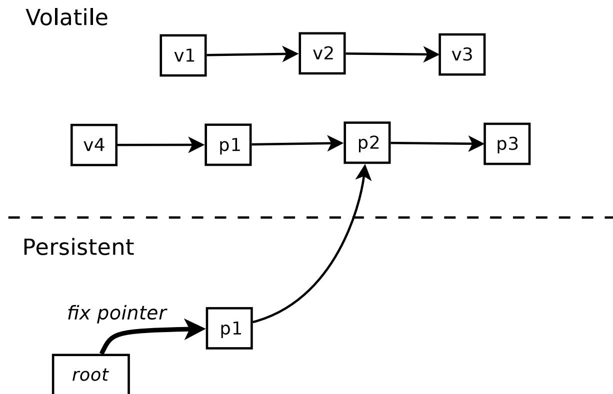
Automatically Discovering Data

Initially all memory is volatile except a *root* structure.



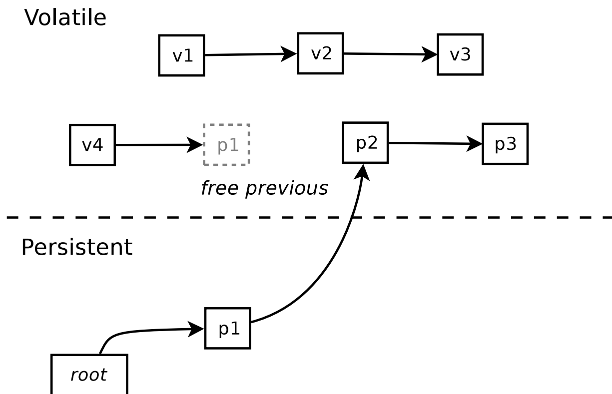
Automatically Discovering Data

Initially all memory is volatile except a *root* structure.



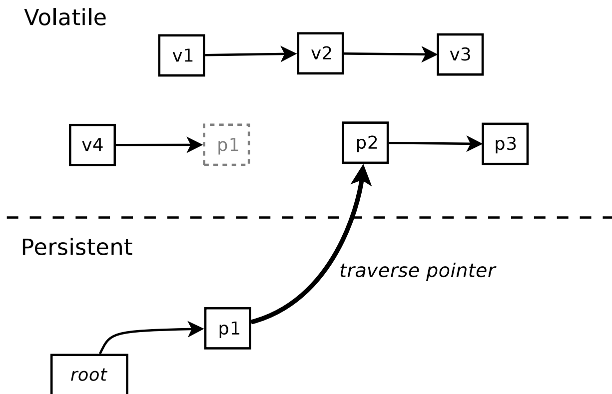
Automatically Discovering Data

Initially all memory is volatile except a *root* structure.



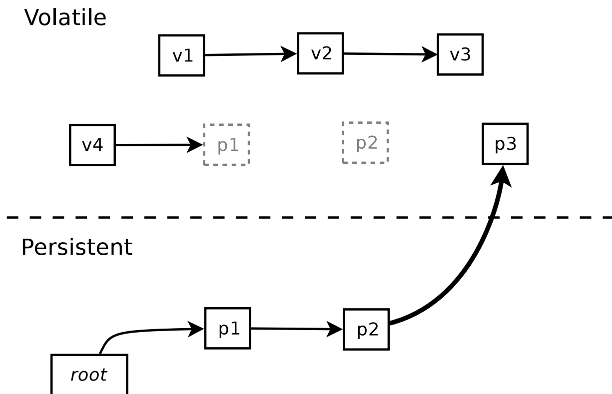
Automatically Discovering Data

Initially all memory is volatile except a *root* structure.



Automatically Discovering Data

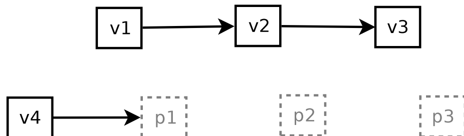
Initially all memory is volatile except a *root* structure.



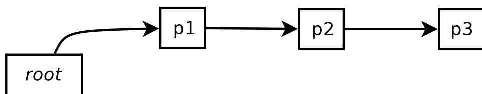
Automatically Discovering Data

Initially all memory is volatile except a *root* structure.

Volatile



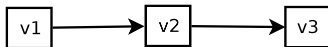
Persistent



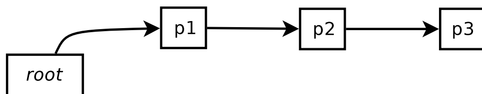
Automatically Discovering Data

Initially all memory is volatile except a *root* structure.

Volatile

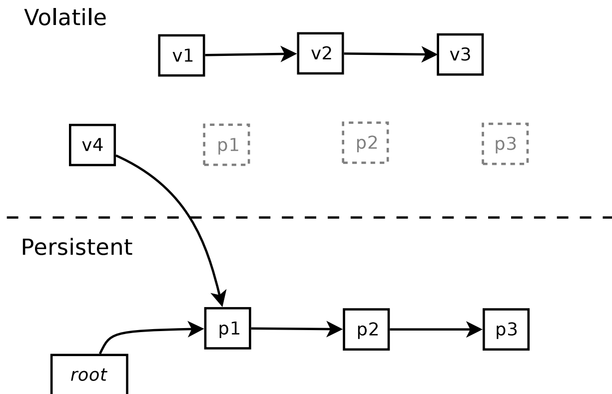


Persistent



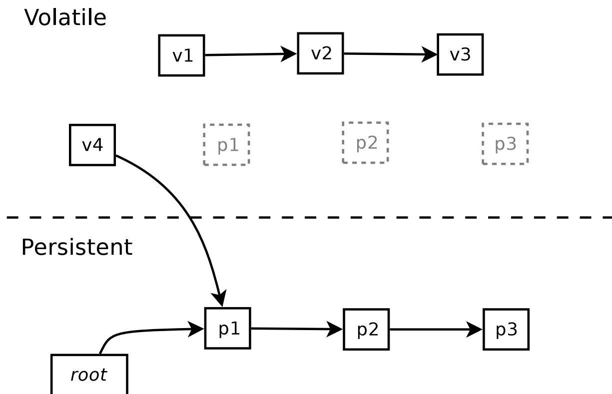
Automatically Discovering Data

Initially all memory is volatile except a *root* structure.



Automatically Discovering Data

Initially all memory is volatile except a *root* structure.



We refer to this root structure and it's closure as **container**.

What Modifications are Needed?

Developer responsibility

- Allocate the containers
- Assign root structures to containers
- Make root structures point to data
- Invoke persistence point creation operations

Pointer Detection in *SoftPM*

Life-cycle of a pointer

- **Allocation:** memory is allocated to store the pointer
- **Initialization:** value of pointer is initialized
- **Use:** pointer value is used
- **Deallocation:** memory used to store pointer is deallocated

Pointer Detection in *SoftPM*

Life-cycle of a pointer

- **Allocation:** memory is allocated to store the pointer
- **Initialization:** value of pointer is initialized
- **Use:** pointer value is used
- **Deallocation:** memory used to store pointer is deallocated

Two-phase pointer detection

- **Static analysis:** CIL-based source translation stage
- **Dynamic analysis:** Runtime tracking of **initialized** pointers

SoftPM Static Analysis

Type Definition	Usage

`struct list {`
`int val;`
`struct list *next;`
`};`
	`struct list *l = malloc(n);`

CIL-based static translation hints

SoftPM Static Analysis

Type Definition	Usage

`struct list {`
`int val;`
`struct list *next;`
`};`
	`struct list *l = malloc(n);`
	`mallocat(l, n);`

CIL-based static translation hints

- Record the location/size of memory allocation/move operations

SoftPM Static Analysis

Type Definition	Usage

`struct list {`
`    int val;`
`    struct list *next;`
`};`
	`struct list *l = malloc(n);`
	`mallocat(l, n);`
	`pointerat(&l);`

CIL-based static translation hints

- Record the location/size of memory allocation/move operations
- Register the location of all initialized pointers
  - ✓ Explicit initialization: *l-value* of assignment operations
  - ✓ Implicit initialization: *memory copying/move* operations

LIMA: Write Handler and Flusher

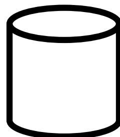
Write Handler

- Writes are monitored at a chunk (multi-page) granularity
- One page-fault per chunk between persistence points
- Asynchronous persistence points use COW pages

Persistence
Point

Persistent
Memory

Persistent
Medium



LIMA: Write Handler and Flusher

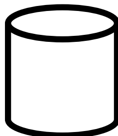
Write Handler

- Writes are monitored at a chunk (multi-page) granularity
- One page-fault per chunk between persistence points
- Asynchronous persistence points use COW pages

Persistence Point

Persistent Memory

Persistent Medium



LIMA: Write Handler and Flusher

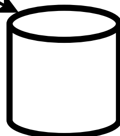
Write Handler

- Writes are monitored at a chunk (multi-page) granularity
- One page-fault per chunk between persistence points
- Asynchronous persistence points use COW pages

Persistence Point

Persistent Memory

Persistent Medium



LIMA: Write Handler and Flusher

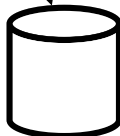
Write Handler

- Writes are monitored at a chunk (multi-page) granularity
- One page-fault per chunk between persistence points
- Asynchronous persistence points use COW pages

Persistence Point

Persistent Memory

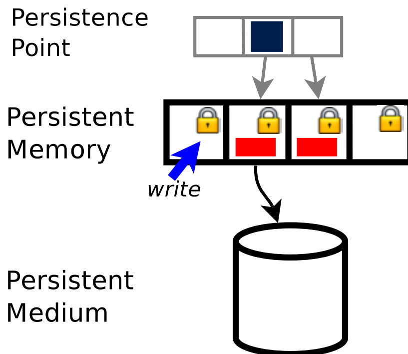
Persistent Medium



LIMA: Write Handler and Flusher

Write Handler

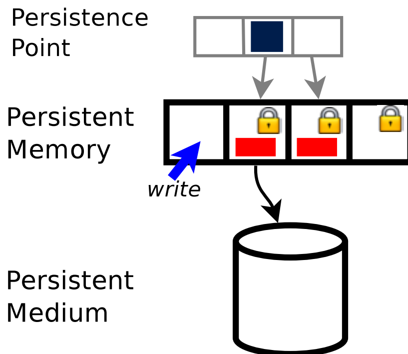
- Writes are monitored at a chunk (multi-page) granularity
- One page-fault per chunk between persistence points
- Asynchronous persistence points use COW pages



LIMA: Write Handler and Flusher

Write Handler

- Writes are monitored at a chunk (multi-page) granularity
- One page-fault per chunk between persistence points
- Asynchronous persistence points use COW pages



LIMA: Write Handler and Flusher

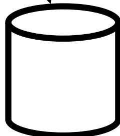
Write Handler

- Writes are monitored at a chunk (multi-page) granularity
- One page-fault per chunk between persistence points
- Asynchronous persistence points use COW pages

Persistence Point

Persistent Memory

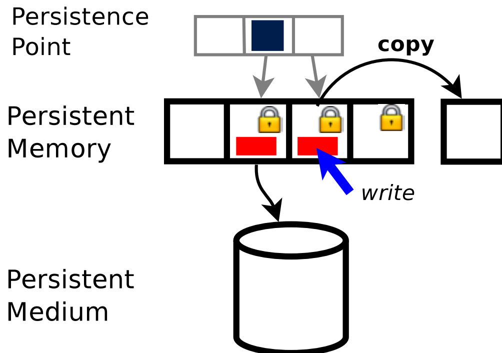
Persistent Medium



LIMA: Write Handler and Flusher

Write Handler

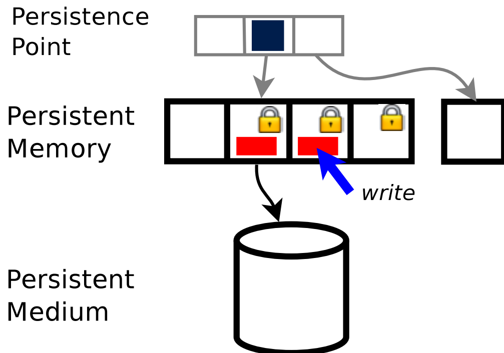
- Writes are monitored at a chunk (multi-page) granularity
- One page-fault per chunk between persistence points
- Asynchronous persistence points use COW pages



LIMA: Write Handler and Flusher

Write Handler

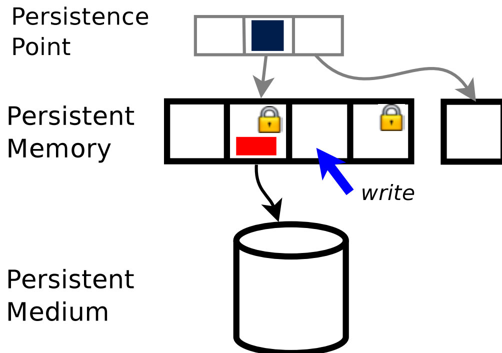
- Writes are monitored at a chunk (multi-page) granularity
- One page-fault per chunk between persistence points
- Asynchronous persistence points use COW pages



LIMA: Write Handler and Flusher

Write Handler

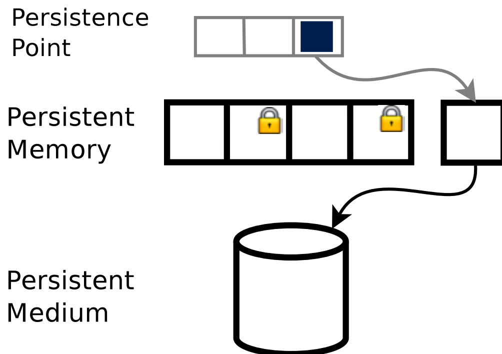
- Writes are monitored at a chunk (multi-page) granularity
- One page-fault per chunk between persistence points
- Asynchronous persistence points use COW pages



LIMA: Write Handler and Flusher

Write Handler

- Writes are monitored at a chunk (multi-page) granularity
- One page-fault per chunk between persistence points
- Asynchronous persistence points use COW pages



Storage I/O Driver (SID)

High-performance I/O

- Asynchronous I/O improves storage I/O efficiency
- Chunk remapping writes provide storage location flexibility

Storage I/O Driver (SID)

High-performance I/O

- Asynchronous I/O improves storage I/O efficiency
- Chunk remapping writes provide storage location flexibility

Atomic persistence points (transaction based)

- Chunk versioning
- Barrier chunks

Storage I/O Driver (SID)

High-performance I/O

- Asynchronous I/O improves storage I/O efficiency
- Chunk remapping writes provide storage location flexibility

Atomic persistence points (transaction based)

- Chunk versioning
- Barrier chunks

Portable I/O handling

- Three different storage I/O drivers

Storage I/O Driver (SID)

High-performance I/O

- Asynchronous I/O improves storage I/O efficiency
- Chunk remapping writes provide storage location flexibility

Atomic persistence points (transaction based)

- Chunk versioning
- Barrier chunks

Portable I/O handling

- Three different storage I/O drivers

Scalability

- Custom parallel file system metadata management
- Custom parallel file system space management

Evaluation

Configurations

- **ALP**: Application-level persistence using the file system
- ***SoftPM***: Using SSD or disk drive

Development complexity evaluation

- Evaluate using basic data structure types
- Complexity measure using lines of code (LoC)

Performance evaluation

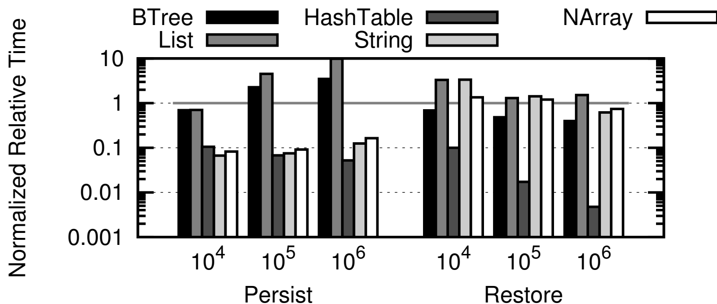
- Microbenchmarks create 1 or more containers
- Populate each container with a single data structure type
- Container size is variable
- Perform random updates when update is necessary
- Vary update locality

Evaluating Development Complexity

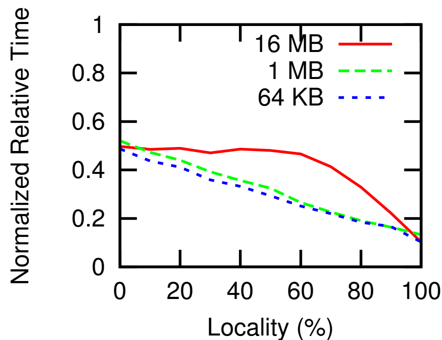
Data Structure	ALP			SoftPM	
	LoC	*P*	*R*	*P*	*R*
String	28	7	13	2	1
N-Dimensional Array	104	13	17	2	1
Linked List	60	9	11	2	1
Hash Table	229	87	18	2	1
Binary Tree	70	14	40	2	1

P: persistence LoC
R: restoration LoC

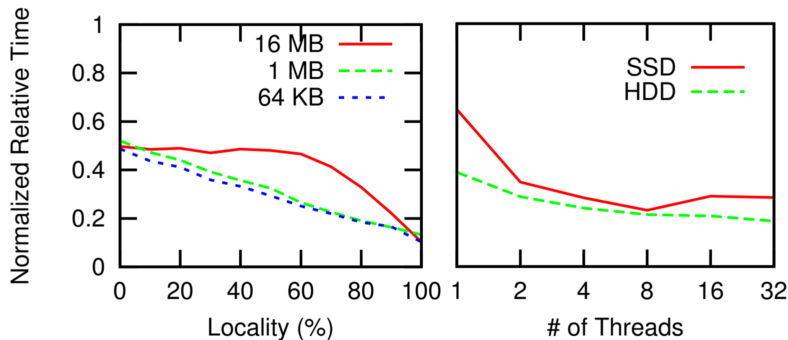
Sensitivity to the Data Structure



Update Locality, Scalability, and Portability



Update Locality, Scalability, and Portability



SQLite: A Case Study

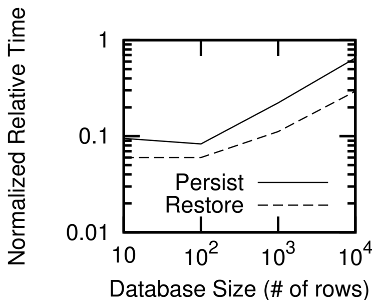
SQLite: An in-memory production database

- Claim: Most widely deployed SQL DB engine
- 39KLoC
- Uses a variety of data structures (HTs, trees, arrays, etc.)
- Does not provide Tx commit support

SQLite: A Case Study

SQLite: An in-memory production database

- Claim: Most widely deployed SQL DB engine
- 39KLoC
- Uses a variety of data structures (HTs, trees, arrays, etc.)
- Does not provide Tx commit support



Caveats

When is *SoftPM* less relevant?

- Unstructured data (e.g., strings, audio) easy to serialize
- Unstructured data do not typically get modified in place
- When subsets of data structures must be serialized

Unresolved issues

- Container size restrictions
- Container sharing semantics across processes

Summary

Simplifying reliable software development

- A memory abstraction for persistent data
- Fully automated persistence
- Transactional persistence
- High-performance persistence
- Versioning, branching, and out-of-core support (under development)

The *SoftPM* Team

Daniel C.

Daniel G.

Jorge

Leonardo

The *SoftPM* Team

Daniel C.

Daniel G.

Jorge

Leonardo

Jason

Jinpeng

Ming

Raju

Thank you!

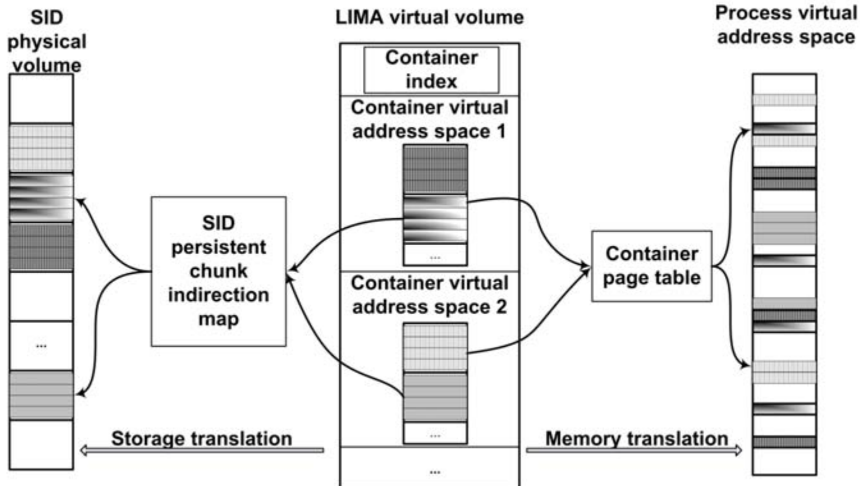
Backup Slides

SoftPM API

API v1.0

- Container administration
  `pCAlloc` / `pCFree` / `pCDelete` / `pExclude`
- Container persistence management
  `pCSetAttr` / `pCGetAttr` / `pPoint` / `pPointSync`
- Restoring and navigating through branches
  `pCRestore` / `pCSwitch` / `pCClone`

Indirection Summary



Distribution of Overheads