A Concurrent Persistent Functional Language
Towards Practical Functional Databases

Lesley Wevers ¹  Ander de Keijzer ²  Marieke Huisman ¹

¹University of Twente, Netherlands
²Windesheim University of Applied Sciences, Netherlands

IFL 2012, Oxford, UK
1 September, 2012
Transactions

- There is a global state.
Transactions

- There is a global state.
- A transaction is a collection of operations on the global state.
Transactions

- There is a global state.
- A transaction is a collection of operations on the global state.

ACID Properties

Atomic: Either all or none of the operations of a transaction are executed.
Transactions

- There is a global state.
- A transaction is a collection of operations on the global state.

ACID Properties

**Atomic**: Either *all or none* of the operations of a transaction are executed.

**Consistent**: After each transaction, the system is in a consistent state.
Transactions

- There is a global state.
- A transaction is a collection of operations on the global state.

ACID Properties

Atomic: Either all or none of the operations of a transaction are executed.

Consistent: After each transaction, the system is in a consistent state.

Isolated: It seems as if the transactions are executed one by one (serializability and recoverability).
Transactions

- There is a global state.
- A transaction is a collection of operations on the global state.

ACID Properties

**Atomic:** Either all or none of the operations of a transaction are executed.

**Consistent:** After each transaction, the system is in a consistent state.

**Isolated:** It seems as if the transactions are executed one by one (serializability and recoverability).

**Durable:** The effect of a transaction is permanent.
Examples

- Banking
Transaction Processing Systems

Examples

- Banking
- Ticket reservation
Transaction Processing Systems

Examples

- Banking
- Ticket reservation
- Inventarisation systems
Transaction Processing Systems

Examples
- Banking
- Ticket reservation
- Inventarisation systems
- Websites

Challenges
Thousands of simultaneous users:
- Transactions have to be processed correctly.
- Everyone wants a quick response.
- We want to handle a lot of data.
Current Approach

Traditional Architecture

User → Application → DBMS → Database

• Optimised to handle large amounts of data.
• Interface to query and manipulate data.
• Transactions.

Most DBMS's only partially support isolation of transactions due to efficiency reasons.
Current Approach

Traditional Architecture

- User
- Application
- DBMS
- Database

Database Management System

- Optimised to handle large amounts of data.
Current Approach

Traditional Architecture

User → Application → DBMS → Database

Database Management System

- Optimised to handle large amounts of data.
- Interface to query and manipulate data.
Current Approach

### Traditional Architecture

- **User**
  - Application
  - **DBMS**
  - Database

### Database Management System

- Optimised to handle large amounts of data.
- Interface to query and manipulate data.
- Transactions.
Current Approach

Traditional Architecture

User → Application → DBMS → Database

Database Management System

- Optimised to handle large amounts of data.
- Interface to query and manipulate data.
- Transactions.

Most DBMS's only partially support isolation of transactions due to efficiency reasons.
Current Approach

Traditional Architecture

- User
- Application
- DBMS
- Database

- Interface to the outside world.
- Can enforce additional security constraints.
Current Approach

**Traditional Architecture**

- User
- Application
- DBMS
- Database

**Application**

- Interface to the outside world.
Current Approach

Traditional Architecture

- User
- Application
- DBMS
- Database

Application

- Interface to the outside world.
- Can enforce additional security constraints.
Current Approach

Traditional Architecture

- User
- Application
- DBMS
- Database

Limitations:
- Application and DBMS have different type systems.
- Serial interface between application and DBMS.
- Distributed system complicates implementation.
- DBMS's are vulnerable to command injection attacks.
- System as a whole is difficult to verify.
Current Approach

Traditional Architecture

- User
- Application
- DBMS
- Database

Limitations
- Application and DBMS have different type system.
Current Approach

Traditional Architecture

Limitations

- Application and DBMS have different type system.
- Serial interface between application and DBMS.
Current Approach

Traditional Architecture

User → Application → DBMS → Database

Limitations

- Application and DBMS have different type system.
- Serial interface between application and DBMS.
- Distributed system complicates implementation.
Current Approach

Traditional Architecture

User → Application → DBMS → Database

Limitations

- Application and DBMS have different type system.
- Serial interface between application and DBMS.
- Distributed system complicates implementation.
- DBMS’s are vulnerable to command injection attacks.
Current Approach

Traditional Architecture

User → Application → DBMS → Database

Limitations

- Application and DBMS have different type system.
- Serial interface between application and DBMS.
- Distributed system complicates implementation.
- DBMS’s are vulnerable to command injection attacks.
- System as a whole is difficult to verify.
Functional Transaction Processing

- A transaction function: \( State \rightarrow State \times Result \).
## Functional Transaction Processing

- A transaction function: \( \text{State} \rightarrow \text{State} \times \text{Result} \).
- A transaction manager: \( \text{State} \times [\text{Transaction}] \rightarrow [\text{Result}] \).

**Correctness (ACID)**

- Atomicity and isolation hold trivially for total transactions.
- A transaction must enforce consistency rules.
- Implementation can easily support durability.
Functional Transaction Processing

- A transaction function: \( \text{State} \rightarrow \text{State} \times \text{Result} \).
- A transaction manager: \( \text{State} \times [\text{Transaction}] \rightarrow [\text{Result}] \).

\[
\begin{align*}
S_0 &\rightarrow t_1 & S_1 &\rightarrow t_2 & S_2 &\rightarrow t_3 & S_3 &\rightarrow \ldots \\
&t_1 &\rightarrow r_1 &t_2 &\rightarrow r_2 &t_3 &\rightarrow r_3
\end{align*}
\]
Functional Transaction Processing

- A transaction function: $\text{State} \rightarrow \text{State} \times \text{Result}$.
- A transaction manager: $\text{State} \times \text{[Transaction]} \rightarrow \text{[Result]}$.

Correctness (ACID)

- Atomicity and isolation hold trivially for total transactions.
**Functional Transaction Processing**

- A *transaction function*: $\text{State} \rightarrow \text{State} \times \text{Result}$.
- A *transaction manager*: $\text{State} \times [\text{Transaction}] \rightarrow [\text{Result}]$.

**Correctness (ACID)**

- Atomicity and isolation hold trivially for *total* transactions.
- A transaction must enforce consistency rules.
Functional Transaction Processing

- **A transaction function**: $\text{State} \rightarrow \text{State} \times \text{Result}$.
- **A transaction manager**: $\text{State} \times [\text{Transaction}] \rightarrow [\text{Result}]$.

Correctness (ACID)

- Atomicity and isolation hold trivially for *total* transactions.
- A transaction must enforce consistency rules.
- Implementation can easily support durability.
Functional Transaction Processing

---

Diagram:

```
  s
    ↓
   B 3
  /   \
 B 1   B 4
   /     \
 L 1     L 8
```

---

- Functional Transaction Processing
- Language Implementation
- Persistence
- Conclusions
Functional Transaction Processing

\[
\begin{array}{c}
  s \\
  \downarrow \\
  B 3 \\
  \downarrow \\
  B 1 \\
  \downarrow \\
  L 1 \\
  \downarrow \\
  \end{array}
\quad
\begin{array}{c}
  s' \\
  \downarrow \\
  \text{map(-1)} \\
  \downarrow \\
  \end{array}
\]

\[
\begin{array}{c}
  B 4 \\
  \downarrow \\
  L 3 \\
  \downarrow \\
  L 4 \\
  \downarrow \\
  L 8 \\
  \end{array}
\]
Functional Transaction Processing
Functional Transaction Processing
Functional Transaction Processing

\[
\begin{array}{c}
B_3 \\
B_1 \\
L_1 \\
B_4 \\
L_3 \\
B_4 \\
L_4 \\
L_8 \\
\end{array}
\quad \begin{array}{c}
s' \quad \text{map(-1)} \quad \text{contains}(7) \\
r \\
\end{array}
\]
Functional Transaction Processing

```
s
B 3
B 1
L 1

B 4
L 3 L 4 L 8

map(-1)

s'
B 2

map(-1)

contains(7)
r

r
```
Functional Transaction Processing

\[
\begin{align*}
\text{s} & \rightarrow B_3 \\
B_1 & \rightarrow B_4 \\
L_1 & \rightarrow L_3 \rightarrow L_4 \rightarrow L_8 \\
s' & \rightarrow B_2 \\
\text{map(-1)} & \rightarrow \text{map(-1)} \\
r & \rightarrow \text{contains(7)}
\end{align*}
\]
Functional Transaction Processing

\[ s \leftarrow B_3 \rightarrow B_1 \rightarrow L_1 \]
\[ s' \leftarrow B_2 \rightarrow B_4 \rightarrow map(-1) \rightarrow L_3 \]
\[ s' \leftarrow B_2 \rightarrow B_3 \rightarrow map(-1) \rightarrow L_4 \]
\[ r \leftarrow contains(7) \rightarrow \]

- \( B_1 \)
- \( L_1 \)
- \( B_1 \)
- \( L_3 \)
- \( B_4 \)
- \( L_4 \)
- \( B_3 \)
- \( L_8 \)

map(-1) map(-1)
Functional Transaction Processing
Functional Transaction Processing
Functional Transaction Processing
A state is a set of bindings \( x = E \), where:

- \( x \) is a name.
- \( E \) is an expression.

A transaction is a set of bindings \( x = E \), where:

- \( x \) is a variable.
- \( E \) is an expression.

**Transaction Variables**

- Current state variables: \( x, y, z, \ldots \).
- Next state variables: \( x', y', z', \ldots \).
- Result variable: \( \text{result} \).
A state is a set of bindings $x = E$, where:

- $x$ is a *name*.
- $E$ is an *expression*.
A state is a set of bindings $x = E$, where:

- $x$ is a name.
- $E$ is an expression.

A transaction is a set of bindings $x = E$, where

- $x$ is a variable.
- $E$ is an expression.
Transactional Functional Language

A state is a set of bindings $x = E$, where:
- $x$ is a name.
- $E$ is an expression.

A transaction is a set of bindings $x = E$, where
- $x$ is a variable.
- $E$ is an expression.

Transaction Variables
- *Current state variables*: $x$, $y$, $z$, ....
Transactional Functional Language

A state is a set of bindings $x = E$, where:
- $x$ is a name.
- $E$ is an expression.

A transaction is a set of bindings $x = E$, where
- $x$ is a variable.
- $E$ is an expression.

Transaction Variables
- Current state variables: $x$, $y$, $z$, ....
- Next state variables: $x'$, $y'$, $z'$, ....
Transactional Functional Language

A state is a set of bindings $x \equiv E$, where:

- $x$ is a name.
- $E$ is an expression.

A transaction is a set of bindings $x \equiv E$, where

- $x$ is a variable.
- $E$ is an expression.

Transaction Variables

- **Current state variables**: $x$, $y$, $z$, ....
- **Next state variables**: $x'$, $y'$, $z'$, ....
- **Result variable**: result
Example

\[ s_0: \text{names} = \text{["alice", "bob"]} \]
Example

\[ s_0: \quad \text{names} = ["alice", "bob"] \]

\[ t_1: \quad \text{names'} = "dave" : \text{names} \]

\[ \text{result} = \text{names'} \]
Transactional Functional Language

Example

\[ s_0: \quad \text{names} \ = \ ["alice", \ "bob"] \]

\[ t_1: \quad \text{names’} \ = \ "dave" : \ \text{names} \]

```
result = names’  \rightarrow r_1: ["dave", \ "alice", \ "bob"]
```

Example

$s_0$: names = ["alice", "bob"]
$t_1$: names’ = "dave" : names
result = names’ → $r_1$: ["dave", "alice", "bob"]
$s_1$: names = ["dave", "alice", "bob"]
Example

\(s_0: \text{names} = ["alice", "bob"]\)

\(t_1: \text{names'} = "dave" : \text{names}\)

\(\text{result} = \text{names'} \rightarrow r_1: ["dave", "alice", "bob"]\)

\(s_1: \text{names} = ["dave", "alice", "bob"]\)

\(t_2: \text{length'} = \lambda \text{list} . \text{case list of}\)

\[\text{[]} \rightarrow 0\]

\[x:xs] \rightarrow 1 + \text{length'} x\)
Example

\[ S_0: \text{names} = ["alice", "bob"] \]
\[ T_1: \text{names'} = "dave" : \text{names} \]
\[ \text{result} = \text{names'} \rightarrow r_1: ["dave", "alice", "bob"] \]
\[ S_1: \text{names} = ["dave", "alice", "bob"] \]
\[ T_2: \text{length'} = \lambda \text{list} . \text{case list of} \]
\[ \quad [\text{[]} -> 0 \]
\[ \quad [x:xs] -> 1 + \text{length'} \text{ xs} \]
\[ S_2: \text{names} = ["dave", "alice", "bob"] \]
\[ \text{length} = \lambda \text{list} . \text{case list of} \]
\[ \quad [\text{[]} -> 0 \]
\[ \quad [x:xs] -> 1 + \text{ length } \text{ xs} \]
Example

\( s_0: \) names = ["alice", "bob"]
\( t_1: \) names’ = "dave" : names
\( \text{result} = \text{names’} \rightarrow r_1: ["dave", "alice", "bob"] \)
\( s_1: \) names = ["dave", "alice", "bob"]
\( t_2: \) length’ = \( \lambda \text{list} . \text{case list of} \)
\( \quad [] \rightarrow 0 \)
\( \quad [x:xs] \rightarrow 1 + \text{length’} \times xs \)
\( s_2: \) names = ["dave", "alice", "bob"]
length = \( \lambda \text{list} . \text{case list of} \)
\( \quad [] \rightarrow 0 \)
\( \quad [x:xs] \rightarrow 1 + \text{length} \times xs \)
\( t_3: \) result = length names
Example

$s_0$: names = ["alice", "bob"]

$t_1$: names’ = "dave" : names

result = names’ → $r_1$: ["dave", "alice", "bob"]

$s_1$: names = ["dave", "alice", "bob"]

$t_2$: length’ = λ list . case list of

  [] → 0

  [x:xs] → 1 + length’ xs

$s_2$: names = ["dave", "alice", "bob"]

length = λ list . case list of

  [] → 0

  [x:xs] → 1 + length xs

$t_3$: result = length names → $r_3$: 3
Goals

- Execute transactions concurrently.
Goals

- Execute transactions concurrently.
- Store states in persistent memory.
Implementation

Goals

- Execute transactions concurrently.
- Store states in persistent memory.

Overview

- Interpretation of transactions.

Prototype

We implemented a prototype in Java.
Implementation

Goals

- Execute transactions concurrently.
- Store states in persistent memory.

Overview

- Interpretation of transactions.
- Allow bindings to be created dynamically.
Goals

- Execute transactions concurrently.
- Store states in persistent memory.

Overview

- Interpretation of transactions.
- Allow bindings to be created dynamically.
- Execute transactions concurrently and in parallel.
Goals

- Execute transactions concurrently.
- Store states in persistent memory.

Overview

- Interpretation of transactions.
- Allow bindings to be created dynamically.
- Execute transactions concurrently and in parallel.
- Store state in persistent memory.

Prototype

We implemented a prototype in Java.
Graph Reduction

Based on template instantiation:

- For every binding we have a template graph.
Graph Reduction

Based on template instantiation:

- For every binding we have a template graph.
- We have a reduction graph.
Graph Reduction

Based on template instantiation:

- For every binding we have a template graph.
- We have a reduction graph.
- On application of a binding we instantiate its template.
Graph Reduction

Based on template instantiation:

- For every binding we have a template graph.
- We have a reduction graph.
- On application of a binding we instantiate its template.

Adaptations to support dynamic bindings

We resolve references to template graphs statically.

- Templates are anonymous, and can be garbage collected.
Concurrent and Parallel Graph Reduction

We use multiple reduction threads to:

• Execute transactions concurrently, such that they do not depend on each other.
• Execute transactions in parallel using multiple processors.

Result sharing and randomisation

Reduction threads:

• Share results between one another.
• Take random reduction “paths”.
We use multiple reduction threads to:

- Execute transactions *concurrently*, such that they do not depend on each other.
Concurrent and Parallel Graph Reduction

We use multiple reduction threads to:

- Execute transactions *concurrently*, such that they do not depend on each other.
- Execute transactions *in parallel* using multiple processors.

Result sharing and randomisation

Reduction threads:
Implementation - Graph Reduction

Concurrent and Parallel Graph Reduction

We use multiple reduction threads to:

- Execute transactions *concurrently*, such that they do not depend on each other.
- Execute transactions *in parallel* using multiple processors.

Result sharing and randomisation

Reduction threads:

- Share results between one another.
Implementation - Graph Reduction

Concurrent and Parallel Graph Reduction

We use multiple reduction threads to:

- Execute transactions \textit{concurrently}, such that they do not depend on each other.
- Execute transactions \textit{in parallel} using multiple processors.

Result sharing and randomisation

Reduction threads:

- Share results between one another.
- Take random reduction “paths”.
Sharing Results

Special nodes to share results:

• Inserted before every redexes.
• Reference to either redex or result, which is updated on reduction.
• Algorithm that ensures that updating maintains sharing between threads.

Randomisation
• We reduce strict-function arguments in a random order:
• We reduce data elements in a random order.
Implementation - Graph Reduction

Sharing Results

Special nodes to share results:

- Inserted before every redexes.
Implementation - Graph Reduction

Sharing Results

Special nodes to share results:

- Inserted before every redexes.
- Reference to either redex or result, which is updated on reduction.
Sharing Results

Special nodes to share results:

- Inserted before every redexes.
- Reference to either redex or result, which is updated on reduction.
- Algorithm that ensures that updating maintains sharing between threads.
Implementation - Graph Reduction

Sharing Results

Special nodes to share results:

- Inserted before every redexes.
- Reference to either redex or result, which is updated on reduction.
- Algorithm that ensures that updating maintains sharing between threads.

Randomisation

- We reduce strict-function arguments in a random order:
- We reduce data elements in a random order.
Evaluation

Reduction Threads

Relative Speedup

treesize

treesize-native

nfib

nfib-native

ideal speedup

19 / 30
## Evaluation

<table>
<thead>
<tr>
<th></th>
<th>treesize</th>
<th>treesize-native</th>
<th>nfib</th>
<th>nfib-native</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Serial</strong></td>
<td>2666 ms</td>
<td>819 ms</td>
<td>3294 ms</td>
<td>626 ms</td>
</tr>
<tr>
<td><strong>Parallel</strong></td>
<td>3243 ms</td>
<td>1291 ms</td>
<td>4162 ms</td>
<td>819 ms</td>
</tr>
<tr>
<td><strong>Overhead</strong></td>
<td>21.6%</td>
<td>57.6%</td>
<td>26.4%</td>
<td>30.1%</td>
</tr>
</tbody>
</table>
Evaluation

Number of transactions after update.

- **no updates**
- **1 update**

Δ Bytes

1,000,000
800,000
600,000
400,000
200,000
0

100
1,000
10,000
100,000
1,000,000
Journaling

Journaling ensures atomicity and durability:

- Transactions are logged to a journal before execution.
- If the system crashes, the state can be recovered by re-executing the logged transactions.
Journaling ensures *atomicity* and *durability*:

- Transactions are logged to a journal before execution.
- If the system crashes, the state can be *recovered* by re-executing the logged transactions.

Checkpointing

Journaling is not enough:

- The log can grow very large.
Persistence - Journaling

**Journaling**

Journaling ensures *atomicity* and *durability*:

- Transactions are logged to a journal before execution.
- If the system crashes, the state can be *recovered* by re-executing the logged transactions.

**Checkpointing**

Journaling is not enough:

- The log can grow very large.
- Long recovery times.
Journaling ensures *atomicity* and *durability*:

- Transactions are logged to a journal before execution.
- If the system crashes, the state can be *recovered* by re-executing the logged transactions.

Checkpointing

Journaling is not enough:

- The log can grow very large.
- Long recovery times.
- The size of the state is limited to main-memory.
Persistence - Snapshotting

Approach

Serialise the state to a snapshot file.

• Complication: Concurrent serialisation and graph reduction.

Advantages
• We can snapshot computations.
• We can maintain sharing.

Disadvantages
• The state is limited to main-memory.
• Recovery can take a while.
Persistence - Snapshotting

**Approach**

Serialise the state to a snapshot file.

- Complication: Concurrent serialisation and graph reduction.
## Persistence - Snapshotting

### Approach
Serialise the state to a snapshot file.
- Complication: Concurrent serialisation and graph reduction.

### Advantages
- We can snapshot computations.
Persistence - Snapshotting

**Approach**

Serialise the state to a snapshot file.

- Complication: Concurrent serialisation and graph reduction.

**Advantages**

- We can snapshot computations.
- We can maintain sharing.
Persistence - Snapshotting

**Approach**
Serialise the state to a snapshot file.
- Complication: Concurrent serialisation and graph reduction.

**Advantages**
- We can snapshot computations.
- We can maintain sharing.

**Disadvantages**
- The state is limited to main-memory.
## Persistence - Snapshotting

### Approach

Serialise the state to a snapshot file.
- Complication: Concurrent serialisation and graph reduction.

### Advantages

- We can snapshot computations.
- We can maintain sharing.

### Disadvantages

- The state is limited to main-memory.
- Recovery can take a while.
Persistence - Log-Structured Storage

```
root
    ▼
  7: Branch
     ▼  ▼
3: Branch  6: Branch
      ▼  ▼  ▼
1: Leaf  2: Leaf  4: Leaf  5: Leaf
```

1: Leaf
2: Leaf
3: Branch 1 2
4: Leaf
5: Leaf
6: Branch 4 5
7: Branch 3 6
8: Root 7
Persistence - Log-Structured Storage

root

7: Branch

3: Branch

1: Leaf 2: Leaf

6: Branch

4: Leaf 5: Leaf

11: Branch

1: Leaf

2: Leaf

3: Branch 1 2

4: Leaf

5: Leaf

6: Branch 4 5

7: Branch 3 6

8: Root 7

9: Leaf

10: Branch 9 5

11: Branch 3 10

12: Root 12
Persistence - Log-Structured Storage

1: Leaf
2: Leaf
3: Branch 1 2
4: Leaf
5: Leaf
6: Branch 4 5
7: Branch 3 6
8: Root 7

root
  ↓
7: Branch
  ↓  ↓
3: Branch 6: Stored
  ↓  ↓
1: Leaf 2: Leaf
Persistence - Log-Structured Storage

Advantages

- States larger than main-memory.
Advantages

- States larger than main-memory.
- Fast recovery.
# Persistence - Log-Structured Storage

## Advantages
- States larger than main-memory.
- Fast recovery.

## Disadvantages / Complications
- Checkpointing of computations is expensive.
Persistence - Log-Structured Storage

Advantages

- States larger than main-memory.
- Fast recovery.

Disadvantages / Complications

- Checkpointing of computations is expensive.
- Garbage collection is needed.
## Persistence - Log-Structured Storage

### Advantages
- States larger than main-memory.
- Fast recovery.

### Disadvantages / Complications
- Checkpointing of computations is expensive.
- Garbage collection is needed.
- Maintaining sharing is expensive.
Advantages

- States larger than main-memory.
- Fast recovery.

Disadvantages / Complications

- Checkpointing of computations is expensive.
- Garbage collection is needed.
- Maintaining sharing is expensive.
- We need to maintain locality of reference.
Persistence - Combined Approach

Approach

We split the heap into two parts:

**Unreduced:** Snapshotting

**Reduced:** Log-structured storage

Advantages

• We can checkpoint computations.

• States can be larger than main-memory.

Disadvantages

• Garbage collection is still needed.

• There is no sharing in the reduced heap.
Persistence - Combined Approach

**Approach**

We split the heap into two parts:

- **Unreduced**: Snapshotting
- **Reduced**: Log-structured storage

**Advantages**

- We can checkpoint computations.
Persistence - Combined Approach

Approach

We split the heap into two parts:

Unreduced: Snapshotting

Reduced: Log-structured storage

Advantages

- We can checkpoint computations.
- States can be larger than main-memory.
## Persistence - Combined Approach

### Approach

We split the heap into two parts:

- **Unreduced**: Snapshotting
- **Reduced**: Log-structured storage

### Advantages

- We can checkpoint computations.
- States can be larger than main-memory.
- Fast recovery.
Persistence - Combined Approach

Approach

We split the heap into two parts:

- **Unreduced**: Snapshotting
- **Reduced**: Log-structured storage

Advantages

- We can checkpoint computations.
- States can be larger than main-memory.
- Fast recovery.

Disadvantages

- Garbage collection is still needed.
## Persistence - Combined Approach

### Approach

We split the heap into two parts:

- **Unreduced**: Snapshotting
- **Reduced**: Log-structured storage

### Advantages

- We can checkpoint computations.
- States can be larger than main-memory.
- Fast recovery.

### Disadvantages

- Garbage collection is still needed.
- There is no sharing in the reduced heap.
Conclusions

We designed a functional language for transaction processing:

- Prototype implementation.
Conclusions

We designed a functional language for transaction processing:

- Prototype implementation.
- Concurrent execution of transactions.
We designed a functional language for transaction processing:

- Prototype implementation.
- Concurrent execution of transactions.
- Allow bindings to be created dynamically.
Conclusions

We designed a functional language for transaction processing:

- Prototype implementation.
- Concurrent execution of transactions.
- Allow bindings to be created dynamically.
- New approach to parallel graph reduction.
Conclusions

We designed a functional language for transaction processing:

- Prototype implementation.
- Concurrent execution of transactions.
- Allow bindings to be created dynamically.
- New approach to parallel graph reduction.
- Investigated method for storing states in persistent memory.
Future Work

(1) Resolving thunk leaks in lazily constructed states.
(2) Implementation of storage mechanisms.
(3) Practical use of system.
(4) Scheduling of reduction threads.