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

TeReSe, Vrije Universiteit, Amsterdam
14 November 2012
Transaction Processing Systems

- Banking
Transaction Processing Systems

- Banking
- Ticket reservation
### Transaction Processing Systems

- Banking
- Ticket reservation
- Inventarisation systems
Transaction Processing Systems

- Banking
- Ticket reservation
- Inventarisation systems
- Websites

Challenges:
- Thousands of simultaneous users:
  - Transactions have to be processed correctly.
  - Everyone wants a quick response.
  - Transactions can be very large.
  - We want to handle a lot of data.
Transaction Processing Systems

- Banking
- Ticket reservation
- Inventarisation systems
- Websites

Challenges

Thousands of simultaneous users:
Transaction Processing Systems

Transaction Processing Systems

- Banking
- Ticket reservation
- Inventarisation systems
- Websites

Challenges

Thousands of simultaneous users:

- Transactions have to be processed correctly.
Transaction Processing Systems

Transaction Processing Systems

- Banking
- Ticket reservation
- Inventarisation systems
- Websites

Challenges

Thousands of simultaneous users:

- Transactions have to be processed correctly.
- Everyone wants a quick response.
Transaction Processing Systems

Transaction Processing Systems

- Banking
- Ticket reservation
- Inventarisation systems
- Websites

Challenges

Thousands of simultaneous users:
- Transactions have to be processed correctly.
- Everyone wants a quick response.
- Transactions can be very large.
# Transaction Processing Systems

## Transaction Processing Systems

- Banking
- Ticket reservation
- Inventarisation systems
- Websites

## Challenges

Thousands of simultaneous users:

- Transactions have to be processed correctly.
- Everyone wants a quick response.
- Transactions can be very large.
- We want to handle a lots of data.
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.
Current Approach

Traditional Architecture

User \rightarrow Application \rightarrow DBMS \rightarrow 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.

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.
- 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

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
- 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

User ➔ Application ➔ DBMS ➔ Database

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: $\text{State} \rightarrow \text{State} \times \text{Result}$. 
Functional Transaction Processing

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

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

Diagram: 

- $s_0 \rightarrow t_1 \rightarrow s_1 \rightarrow t_2 \rightarrow s_2 \rightarrow t_3 \rightarrow s_3 \rightarrow \ldots$

- $r_1 \downarrow t_1$
- $r_2 \downarrow t_2$
- $r_3 \downarrow t_3$
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 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 \textit{total} transactions.
- A transaction must enforce consistency rules.
Functional Transaction Processing

A transaction function: State $\rightarrow$ State $\times$ Result.

A transaction manager: State $\times$ [Transaction] $\rightarrow$ [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

\[
\begin{array}{c}
S \\
B 3 \\
B 1 \quad B 4 \\
L 1 \quad L 3 L 4 \quad L 8 \\
\end{array}
\]
Functional Transaction Processing

```
s
B 3

B 1

L 1

B 4

L 3

L 4

L 8

s'

map(-1)
```

B 3

B 4

L 3

L 4

L 8
Functional Transaction Processing

\[
s \quad \downarrow \quad s' \quad \downarrow \quad \text{map}(-1) \quad \downarrow \quad \text{contains}(7) \\
\begin{array}{c}
B 3 \\
\text{map}(-1) \\
\end{array} \\
\begin{array}{c}
B 1 \\
\text{contains}(7) \\
\end{array} \\
\begin{array}{c}
B 4 \\
\text{contains}(7) \\
\end{array} \\
\begin{array}{c}
L 1 \\
\text{contains}(7) \\
\end{array} \\
\begin{array}{c}
L 3 \\
\text{contains}(7) \\
\end{array} \\
\begin{array}{c}
L 4 \\
\text{contains}(7) \\
\end{array} \\
\begin{array}{c}
L 8 \\
\text{contains}(7) \\
\end{array} \\
\]
Functional Transaction Processing

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

\[
\begin{align*}
\text{s} & \quad \downarrow \\
\text{B 3} & \quad \text{map(-1)} \quad \downarrow \\
\text{B 1} & \quad \text{contains}(7) \\
\text{L 1} & \\
\text{L 3} & \\
\text{L 4} & \\
\text{L 8} & \\
\end{align*}
\]
Functional Transaction Processing
Functional Transaction Processing
Functional Transaction Processing
Functional Transaction Processing

\[
s \quad \downarrow \quad B_3
\]

\[
B_1 \quad \downarrow \quad B_2 \quad \downarrow \quad B_4
\]

\[
L_1 \quad \downarrow \quad L_3 \quad \downarrow \quad L_4 \quad \downarrow \quad L_8
\]

\[
s' \quad \downarrow \quad map(-1) \quad \downarrow \quad B_3
\]

\[
r \quad \downarrow \quad contains(7)
\]

map(-1) map(-1)
Functional Transaction Processing
Functional Transaction Processing
A state is a set of bindings $x = E$, where:
- $x$ is a variable.
- $E$ 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: $x', y', z', ...$
- Result variable: result
Transactional Functional Language

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

- $x$ is a name.
- $E$ 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*. 

<table>
<thead>
<tr>
<th>Transaction Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current state variables: $x, y, z, ...$</td>
</tr>
<tr>
<td>Next state variables: $x', y', z', ...$</td>
</tr>
<tr>
<td>Result variable: $\text{result}$</td>
</tr>
</tbody>
</table>
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*: $x, y, z, \ldots$
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: $x, y, z, \ldots$
- Next state variables: $x', y', z', \ldots$
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**: \( x, y, z, \ldots \)
- **Next state variables**: \( x', y', z', \ldots \)
- **Result variable**: result
Example

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

\[ s_0: \quad \text{name} = ["alice", "bob"] \]
\[ t_1: \quad \text{name}' = "dave" : \text{name} \]
\[ \text{result} = \text{name}' \]
### Example

\[ s_0: \quad \text{names} = ["alice", "bob"] \]
\[ t_1: \quad \text{names'} = "dave" : \text{names} \]
\[ \text{result} = \text{names'} \quad \rightarrow \quad r_1: ["dave", "alice", "bob"] \]
Example

\[ s_0: \text{names} = ["alice", "bob"] \]
\[ t_1: \text{names'} = "dave" : \text{names} \]
\[ \text{result} = \text{names'} \rightarrow r_1: ["dave", "alice", "bob"] \]
\[ s_1: \text{names} = ["dave", "alice", "bob"] \]
Example

\[ s_0: \text{names} = ["alice", "bob"] \]
\[ t_1: \text{names'} = \text{"dave" : names} \]
\[ \text{result} = \text{names'} \rightarrow r_1: ["dave", "alice", "bob"] \]
\[ s_1: \text{names} = ["dave", "alice", "bob"] \]
\[ t_2: \text{length'} = \lambda \text{list} . \text{case list of} \]
\[ \text{[]} \rightarrow 0 \]
\[ [x:xs] \rightarrow 1 + \text{length'} xs \]
**Example**

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

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

result = names' \quad \rightarrow r_1: ["dave", "alice", "bob"]

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

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

\[
\begin{align*}
[] & \rightarrow 0 \\
[x:xs] & \rightarrow 1 + \text{length'} \hspace{1mm} xs
\end{align*}
\]

\(s_2: \quad \text{names} = ["dave", "alice", "bob"]\)

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

\[
\begin{align*}
[] & \rightarrow 0 \\
[x:xs] & \rightarrow 1 + \text{length} \hspace{1mm} xs
\end{align*}
\]
Example

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

$t_1$: names’ = "dave" : names

result = names’ \rightarrow 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
Transactional Functional Language

Example

\[ s_0: \text{names} = ["alice", "bob"] \]
\[ t_1: \text{names'} = "dave" : \text{names} \]
\[ \text{result} = \text{names'} \rightarrow r_1: ["dave", "alice", "bob"] \]

\[ s_1: \text{names} = ["dave", "alice", "bob"] \]
\[ t_2: \text{length'} = \lambda \text{list} . \text{case list of} \]
\[ \quad [\text{}] \rightarrow 0 \]
\[ \quad [\text{x:xs}] \rightarrow 1 + \text{length'} \text{xs} \]

\[ s_2: \text{names} = ["dave", "alice", "bob"] \]
\[ \text{length} = \lambda \text{list} . \text{case list of} \]
\[ \quad [\text{}] \rightarrow 0 \]
\[ \quad [\text{x:xs}] \rightarrow 1 + \text{length} \text{xs} \]

\[ t_3: \text{result} = \text{length names} \rightarrow r_3: 3 \]
Goals

- Transactional functional language.
Goals

- Transactional functional language.
  - Allow ad-hoc transactions.
Goals

- Transactional functional language.
  - Allow ad-hoc transactions.
  - Allow bindings to be created dynamically.
Goals

- Transactional functional language.
  - Allow ad-hoc transactions.
  - Allow bindings to be created dynamically.
- Execute transactions concurrently and in parallel.
Goals

- Transactional functional language.
  - Allow ad-hoc transactions.
  - Allow bindings to be created dynamically.
- Execute transactions concurrently and in parallel.
  - Low response latency.
Goals

- Transactional functional language.
  - Allow ad-hoc transactions.
  - Allow bindings to be created dynamically.

- Execute transactions concurrently and in parallel.
  - Low response latency.
  - Exploit parallel hardware.
Goals

☐ Transactional functional language.
  - Allow ad-hoc transactions.
  - Allow bindings to be created dynamically.

☐ Execute transactions concurrently and in parallel.
  - Low response latency.
  - Exploit parallel hardware.

☐ Store states in persistent memory.
Goals

☐ Transactional functional language.
  ● Allow ad-hoc transactions.
  ● Allow bindings to be created dynamically.

☐ Execute transactions concurrently and in parallel.
  ● Low response latency.
  ● Exploit parallel hardware.

☐ Store states in persistent memory.
  ● Durability
Goals

□ Transactional functional language.
  • Allow ad-hoc transactions.
  • Allow bindings to be created dynamically.
□ Execute transactions concurrently and in parallel.
  • Low response latency.
  • Exploit parallel hardware.
□ Store states in persistent memory.
  • Durability
  • States larger than main memory.

Current Implementation

□ Implementation in Java.
Goals

- Transactional functional language.
  - Allow ad-hoc transactions.
  - Allow bindings to be created dynamically.

- Execute transactions concurrently and in parallel.
  - Low response latency.
  - Exploit parallel hardware.

- Store states in persistent memory.
  - Durability
  - States larger than main memory.

Current Implementation

- Implementation in Java.
- Only main memory states.
Architecture

User

Interface

Parser

Transaction Manager

Recovery

Journaling

Binding Manager

Graph Reducer

Graph

Snapshot

Snapshottting
Template Instantiation Approach

Each function is a supercombinator $\lambda x_1. \lambda x_2. \ldots \lambda x_n. E$: 
Graph Reduction

Template Instantiation Approach

- Each function is a supercombinator $\lambda x_1.\lambda x_2.\ldots\lambda x_n.E$:
  - $E$ is not a supercombinator.
Graph Reduction

Template Instantiation Approach

- Each function is a supercombinator $\lambda x_1. \lambda x_2. \ldots \lambda x_n. E$:
  - $E$ is not a supercombinator.
  - Any $\lambda$-abstraction in $E$ is a supercombinator.
Graph Reduction

Template Instantiation Approach

☐ Each function is a supercombinator \( \lambda x_1.\lambda x_2. \ldots \lambda x_n. E \):
  - \( E \) is not a supercombinator.
  - Any \( \lambda \)-abstraction in \( E \) is a supercombinator.

☐ For every supercombinator we construct a template graph.
Graph Reduction

Template Instantiation Approach

- Each function is a supercombinator $\lambda x_1. \lambda x_2. \ldots \lambda x_n.E$:
  - $E$ is not a supercombinator.
  - Any $\lambda$-abstraction in $E$ is a supercombinator.

- For every supercombinator we construct a template graph.
  - Each template graph has a name.
Template Instantiation Approach

□ Each function is a *supercombinator* $\lambda x_1.\lambda x_2.\ldots\lambda x_n.E$:
  - $E$ is not a supercombinator.
  - Any $\lambda$-abstraction in $E$ is a supercombinator.

□ For every supercombinator we construct a *template graph*.
  - Each template graph has a name.
  - Template graphs may refer to other templates through free variables.
Graph Reduction

Template Instantiation Approach

☐ Each function is a supercombinator $\lambda x_1.\lambda x_2.\ldots\lambda x_n.E$:
  - $E$ is not a supercombinator.
  - Any $\lambda$-abstraction in $E$ is a supercombinator.

☐ For every supercombinator we construct a template graph.
  - Each template graph has a name.
  - Template graphs may refer to other templates through free variables.

☐ Reduction is done in a reduction graph.
Template Instantiation Approach

- Each function is a *supercombinator* $\lambda x_1.\lambda x_2.\ldots \lambda x_n.E$:
  - $E$ is not a supercombinator.
  - Any $\lambda$-abstraction in $E$ is a supercombinator.

- For every supercombinator we construct a *template graph*.
  - Each template graph has a name.
  - Template graphs may refer to other templates through free variables.

- Reduction is done in a *reduction graph*.
  - initialised with graph of expression to be reduced.
Template Instantiation Approach

- Each function is a supercombinator $\lambda x_1.\lambda x_2.\ldots\lambda x_n. E$:
  - $E$ is not a supercombinator.
  - Any $\lambda$-abstraction in $E$ is a supercombinator.

- For every supercombinator we construct a template graph.
  - Each template graph has a name.
  - Template graphs may refer to other templates through free variables.

- Reduction is done in a reduction graph.
  - Initialised with graph of expression to be reduced.
  - Beta-reduction: $(f \ E) \rightarrow T_f[f := E]$, where $T_f$ is the template graph associated with $x$. 
Graph Reduction

Dynamic Bindings

Transactions may:

- Create and remove bindings in an ad-hoc manner.
- Define temporary 'local' bindings.

Problems
Graph Reduction

Dynamic Bindings

Transactions may:

- Create and remove bindings in an ad-hoc manner.
- Define temporary 'local' bindings.

Problems

- How do we know when a template is not in use anymore?
### Graph Reduction

#### Dynamic Bindings

Transactions may:
- Create and remove bindings in an ad-hoc manner.
- Define temporary 'local' bindings.

#### Problems

- How do we know when a template is not in use anymore?
- How do we name local bindings?
Graph Reduction

Dynamic Bindings
Transactions may:

☑ Create and remove bindings in an ad-hoc manner.
☑ Define temporary 'local' bindings.

Problems
☑ How do we know when a template is not in use anymore?
☑ How do we name local bindings?

Solution: Anonymous Templates
Graph Reduction

Dynamic Bindings

Transactions may:

☐ Create and remove bindings in an ad-hoc manner.
☐ Define temporary ’local’ bindings.

Problems

☐ How do we know when a template is not in use anymore?
☐ How do we name local bindings?

Solution: Anonymous Templates

☐ We do not maintain a mapping of names to templates.
### Dynamic Bindings

Transactions may:
- Create and remove bindings in an ad-hoc manner.
- Define temporary 'local' bindings.

### Problems

- How do we know when a template is not in use anymore?
- How do we name local bindings?

### Solution: Anonymous Templates

- We do not maintain a mapping of names to templates.
- We resolve references to template graphs statically.
### Dynamic Bindings

Transactions may:
- Create and remove bindings in an ad-hoc manner.
- Define temporary 'local' bindings.

### Problems

- How do we know when a template is not in use anymore?
- How do we name local bindings?

### Solution: Anonymous Templates

- We do not maintain a mapping of names to templates.
- We resolve references to template graphs statically.
- Garbage collection cleans up unused templates.
Graph Reduction

Scheduling

- We have multiple roots (transaction results) to be reduced.
- We have a fixed number of worker threads (processors).

Goals
- Concurrency: Minimize latency of individual transactions.
- Parallelism: Maximize overall system throughput.

Distributing Work
- Keep workers busy.
- Avoid contention between workers.
- Minimize latency of transactions.
Graph Reduction

Scheduling

- We have multiple roots (transaction results) to be reduced.
Graph Reduction

Scheduling

- We have multiple roots (transaction results) to be reduced.
- We have a fixed number of worker threads (processors).
Graph Reduction

**Scheduling**

- We have multiple roots (transaction results) to be reduced.
- We have a fixed number of worker threads (processors).

**Goals**

Concurrency
- Minimize latency of individual transactions.

Parallelism
- Maximize overall system throughput.

Distributing Work
- Keep workers busy.
- Avoid contention between workers.
- Minimize latency of transactions.
Graph Reduction

Scheduling
- We have multiple roots (transaction results) to be reduced.
- We have a fixed number of worker threads (processors).

Goals
- **Concurrency**  Minimize latency of individual transactions.
Graph Reduction

**Scheduling**
- We have multiple roots (transaction results) to be reduced.
- We have a fixed number of worker threads (processors).

**Goals**
- **Concurrency**: Minimize latency of individual transactions.
- **Parallelism**: Maximize overall system throughput.
Graph Reduction

Scheduling

- We have multiple roots (transaction results) to be reduced.
- We have a fixed number of worker threads (processors).

Goals

Concurrency
- Minimize latency of individual transactions.

Parallelism
- Maximize overall system throughput.

Distributing Work
# Graph Reduction

## Scheduling
- We have multiple roots (transaction results) to be reduced.
- We have a fixed number of worker threads (processors).

## Goals
- **Concurrency**: Minimize latency of individual transactions.
- **Parallelism**: Maximize overall system throughput.

## Distributing Work
- Keep workers busy.
# Graph Reduction

## Scheduling
- We have multiple roots (transaction results) to be reduced.
- We have a fixed number of worker threads (processors).

## Goals
**Concurrency** Minimize latency of individual transactions.

**Parallelism** Maximize overall system throughput.

## Distributing Work
- Keep workers busy.
- Avoid contention between workers.
Graph Reduction

Scheduling

- We have multiple roots (transaction results) to be reduced.
- We have a fixed number of worker threads (processors).

Goals

Concurrency  Minimize latency of individual transactions.
Parallelism  Maximize overall system throughput.

Distributing Work

- Keep workers busy.
- Avoid contention between workers.
- Minimise latency of transactions.
Graph Reduction - Distributing Work

Randomisation
Graph Reduction - Distributing Work

Randomisation

Workers reduce strict function arguments in a different order with respect to each other.
Graph Reduction - Distributing Work

Randomisation

Workers reduce strict function arguments in a different order with respect to each other.

Decisions about the order
Randomisation
Workers reduce strict function arguments in a different order with respect to each other.

Decisions about the order
- Fast random number generator
Graph Reduction - Distributing Work

Randomisation
Workers reduce strict function arguments in a different order with respect to each other.

Decisions about the order
- Fast random number generator
- Coordination between workers
Graph Reduction - Distributing Work

Randomisation

Workers reduce strict function arguments in a different order with respect to each other.

Decisions about the order

- Fast random number generator
- Coordination between workers ← we implemented this
Graph Reduction - Distributing Work

Randomisation
Workers reduce strict function arguments in a different order with respect to each other.

Decisions about the order
- Fast random number generator
- Coordination between workers ← we implemented this

Orders
Graph Reduction - Distributing Work

Randomisation
Workers reduce strict function arguments in a different order with respect to each other.

Decisions about the order
- Fast random number generator
- Coordination between workers ← we implemented this

Orders
- Permutation
Graph Reduction - Distributing Work

Randomisation
Workers reduce strict function arguments in a different order with respect to each other.

Decisions about the order
- Fast random number generator
- Coordination between workers ← we implemented this

Orders
- Permutation
- Left to right / right to left
Graph Reduction - Distributing Work

Randomisation
Workers reduce strict function arguments in a different order with respect to each other.

Decisions about the order
- Fast random number generator
- Coordination between workers ← we implemented this

Orders
- Permutation
- Left to right / right to left ← we implemented this
whnf(Add(left, right, left_to_right) : Node) : Node {
    left_to_right ← !left_to_right;
    if(left_to_right) {
        l ← whnf(left); r ← whnf(right);
    } else {
        r ← whnf(right); l ← whnf(left);
    }
    return Int(l.value + r.value);
}
Graph Reduction - Sharing Results

Problem

Multiple workers can work on the same task:
Graph Reduction - Sharing Results

Problem

Multiple workers can work on the same task:

- Duplicate (small) computations.
Graph Reduction - Sharing Results

Problem

Multiple workers can work on the same task:
- Duplicate (small) computations.
- Duplicate results
Graph Reduction - Sharing Results

Problem

Multiple workers can work on the same task:

- Duplicate (small) computations.
- Duplicate results $\rightarrow$ Duplicate (large) computations.
Graph Reduction - Sharing Results

**Problem**

Multiple workers can work on the same task:
- Duplicate (small) computations.
- Duplicate results → Duplicate (large) computations.

**Solution Strategies**
Graph Reduction - Sharing Results

Problem

Multiple workers can work on the same task:

- Duplicate (small) computations.
- Duplicate results $\rightarrow$ Duplicate (large) computations.

Solution Strategies

- Avoid duplicate computations.
Graph Reduction - Sharing Results

**Problem**

Multiple workers can work on the same task:
- Duplicate (small) computations.
- Duplicate results → Duplicate (large) computations.

**Solution Strategies**

- Avoid duplicate computations.
- Ensure sharing of duplicate results.
whnf(Sharing(shared) : Node) : Node {
    local ← shared;
    reduced ← reduce(local);
    while(local ≠ reduced) {
        if(compareAndSet(shared, local, reduced)):
            local ← reduced;
        } else {
            local ← shared;
        }
    }
    reduced ← reduce(local);
    return local;
}
Graph Reducting - Ensure sharing of duplicate results

```plaintext
whnf(Sharing(shared) : Node) : Node {
    local ← shared;
    reduced ← reduce(local);
    while(local ≠ reduced) {
        if(compareAndSet(shared, local, reduced)):
            local ← reduced;
        } else {
            local ← shared;
        }
    reduced ← reduce(local);
}
return local;
}```
whnf(Sharing(shared) : Node) : Node {
    local ← shared;
    reduced ← reduce(local);
    while(local \neq reduced) {
        if(compareAndSet(shared, local, reduced)):
            local ← reduced;
        } else {
            local ← shared;
        }
    reduced ← reduce(local);
}
return local;
Graph Reducting - Ensure sharing of duplicate results

\[
\text{whnf(Sharing(shared) : Node) : Node} \{ \\
\hspace{1em} \text{local } \leftarrow \text{shared}; \\
\hspace{1em} \text{reduced } \leftarrow \text{reduce(local)}; \\
\hspace{1em} \text{while(local } \neq \text{ reduced) } \{ \\
\hspace{2em} \text{if(compareAndSet(shared, local, reduced))}{ \\
\hspace{3em} \text{local } \leftarrow \text{reduced}; \\
\hspace{2em} }\text{ else } \{ \\
\hspace{3em} \text{local } \leftarrow \text{shared}; \\
\hspace{2em} }\} \\
\hspace{1em} \text{reduced } \leftarrow \text{reduce(local)}; \\
\} \\
\text{return local}; \\
\} 
\]
whnf(Sharing(shared) : Node) : Node {
    local ← shared;
    reduced ← reduce(local);
    while(local \neq reduced) {
        if(compareAndSet(shared, local, reduced)):
            local ← reduced;
        } else {
            local ← shared;
        }
    }
    reduced ← reduce(local);
    return local;
}
Graph Reducting - Ensure sharing of duplicate results

\[
\text{whnf}(\text{Sharing}(\text{shared}) : \text{Node}) : \text{Node} \{ \\
\text{local} \leftarrow \text{shared}; \\
\text{reduced} \leftarrow \text{reduce}(\text{local}); \\
\text{while}(\text{local} \neq \text{reduced}) \{ \\
\quad \text{if}(\text{compareAndSet}(\text{shared}, \text{local}, \text{reduced})): \\
\quad \quad \text{local} \leftarrow \text{reduced}; \\
\quad \} \ \text{else} \ { \\
\quad \text{local} \leftarrow \text{shared}; \\
\quad \} \\
\text{reduced} \leftarrow \text{reduce}(\text{local}); \\
\} \\
\text{return} \ \text{local}; \\
\} 
\]
Graph Reducting - Ensure sharing of duplicate results

\[
\text{whnf}(\text{Sharing}(\text{shared}) : \text{Node}) : \text{Node} \{
\text{local} \leftarrow \text{shared}; \n\text{reduced} \leftarrow \text{reduce}((\text{local}); \n\text{while}(\text{local} \neq \text{reduced}) \{ \n\quad \text{if}(\text{compareAndSet}(\text{shared}, \text{local}, \text{reduced})): \n\quad\quad \text{local} \leftarrow \text{reduced}; \n\quad\} \text{ else } \{ \n\quad \text{local} \leftarrow \text{shared}; \n\quad\}\n\text{reduced} \leftarrow \text{reduce}((\text{local}); \n\}
\text{return local;}
\}
Graph Reducting - Ensure sharing of duplicate results

```
whnf(Sharing(shared) : Node) : Node {
    local ← shared;
    reduced ← reduce(local);
    while(local ≠ reduced) {
        if(compareAndSet(shared, local, reduced)):
            local ← reduced;
        } else {
            local ← shared;
        }
    }
    reduced ← reduce(local);
}
return local;
```
Graph Reducting - Ensure sharing of duplicate results

\[
\text{whnf}(\text{Sharing}(\text{shared}) : \text{Node}) : \text{Node} \{ \\
\quad \text{local} \leftarrow \text{shared}; \\
\quad \text{reduced} \leftarrow \text{reduce}(\text{local}); \\
\quad \text{while}(\text{local} \neq \text{reduced}) \{ \\
\qquad \text{if}(\text{compareAndSet}(\text{shared}, \text{local}, \text{reduced})): \\
\qquad \qquad \text{local} \leftarrow \text{reduced}; \\
\qquad \} \text{ else } \{ \\
\qquad \quad \text{local} \leftarrow \text{shared}; \\
\qquad \} \\
\quad \text{reduced} \leftarrow \text{reduce}(\text{local}); \\
\} \\
\text{return local}; \\
\}
whnf(Sharing(shared) : Node) : Node {  
  local ← shared;  
  reduced ← reduce(local);  
  while(local ≠ reduced) {  
    if(compareAndSet(shared, local, reduced)):  
      local ← reduced;  
    } else {  
      local ← shared;  
    }  
  reduced ← reduce(local);  
  }  
return local;  
}
whnf(Sharing(shared) : Node) : Node {
    local ← shared;
    reduced ← reduce(local);
    while(local ≠ reduced) {
        if(compareAndSet(shared, local, reduced)):
            local ← reduced;
        } else {
            local ← shared;
        }
    }
    reduced ← reduce(local);
}

return local;
Graph Reduction - Evaluation

TODO: Worker always makes progress on given task. Bad for sequential computations.
Evaluation

![Graph showing relative speedup vs. reduction threads]

- **treesize**
- **treesize-native**
- **nfib**
- **nfib-native**
- **ideal speedup**

Relative Speedup vs. Reduction Threads
## 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.
Results

We designed a functional language for transaction processing:

- Prototype implementation.
Conclusions

Results

We designed a functional language for transaction processing:

- Prototype implementation.
- Concurrent execution of transactions.
Conclusions

Results

We designed a functional language for transaction processing:

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

Results

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

### Results

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.

### Conclusions

- Persistent functional languages are feasible.
- There are still open problems.
Future Work

(1) Lazy evaluation leads to space leaks.
(2) Practical use of system.
(3) Handling run-time errors.
(4) Concurrent data structures.
(5) Optimistic concurrency control.