Data Representation Synthesis

Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, Mooly Sagiv

presented by Andreas Schröcker
Concurrency and Memory Management Seminar
Prof. Christoph Kirsch, University of Salzburg
June 14, 2012

Datastructures

- Non-trivial program represents its data internally using dynamically-allocated data structures.
- Commit to a particular choice of heap data structures that represent the system's state.
- Must meet several requirements.
  - The representation must support all of the operations required by the code.
  - The data structures must be efficient for the workload.
  - The implementation must be correct.

Datastructures

- Choice of data structures has a pervasive influence on the subsequent code.
- As requirements evolve it is difficult and tedious to change the data structures.
- For a data representation to be correct, data structure invariants must be enforced by every piece of code that manipulates the heap.

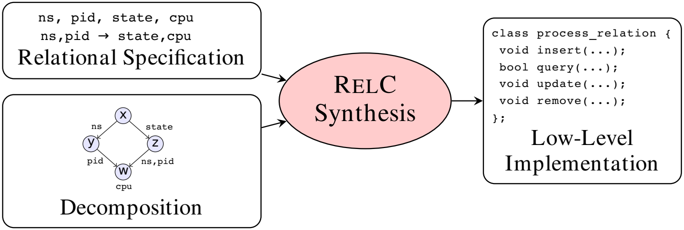
Operating System Scheduler

- Each process has:
  - *pid* - id of the process
  - *state* - the state of the process (sleeping, running)
  - *cpu* - the cpu time consumed by the process
- Adding support for virtualization:
  - *ns* - processes with the same *pid* may exist in different namespaces

Approach - Data Representation Synthesis

- A data structure client describes and manipulates data at a high level as relations.
- A data structure designer then provides decompositions which describe how those relations should be represented in memory as a combination of primitive data structures.
- Our compiler RelC takes a relation and its decomposition and emits correct and efficient low-level code that implements the relational interface.

Data Representation Synthesis



Advantages

- Synthesis allows programmers to describe and manipulate data at a high level as relations, while giving control of how relations are represented physically in memory.
- By abstracting data from its representation, programmers no longer prematurely commit to a particular representation of data.
- Synthesized representations are correct by construction; so long as the programmer conforms to the relational specification, invariants on the synthesized data structures are automatically maintained.

Relational Specification

- A relational specification is a set of column names C and functional dependencies Δ
- Scheduler Example
  - $\{ns, pid, state, cpu\}$
  - $\{ns, pid\} \to \{state, cpu\}$
- We use relations to abstract a program's data from its representation.
- Describing particular representations is the task of the decomposition language

Values, Tuples, Relations

- *tuple* $t = \langle c_1 : v_1, c_2 : v_2, \ldots\rangle$ maps a set of colums to values.
- dom $t = C$.
- A relation a is a set of tuples $\{t_1, t_2, t_3 \ldots\}$.
- $t(c)$ is the value of columns c in tuple t.
- We write $t \supseteq s$ if the tuple t extends s, that is $t(c) = t(s)$ for all c in dom s.
- We say tuple t maches tuple s, written $t \sim s$, if they are equal on all common columns.
- We say tuple t maches relation r, written $t \sim r$, if t matches every tuple in r.
- We write $s \lhd t$ for a merge of tuples. Taking values of t wherever the two disagree on a columns value,

Functional Dependencies

- Relation r has functional dependencies (FD) $C_1 \to C_2$ if any pair of tuples that are equal on columns C_1 are equal on columns C_2.
- We write $r \models_{fd} \Delta$ if a set of FDs Δ hold on relation r.
- We write $\Delta \vdash_{fd} C_1 \to C_2$ if FD $C_1 \to C_2$ is a consequence of FDs Δ.

Relation for Scheduler Example

$$
\begin{aligned}
r_s \quad = \quad \{ \quad & \langle ns:1, pid:1, state:S, cpu:7\rangle, \\
& \langle ns:1, pid:2, state:R, cpu:4\rangle, \\
& \langle ns:2, pid:1, state:S, cpu:5\rangle \quad \}
\end{aligned}
$$

Relational Algebra

We use the standard notation of relational algebra.

- Union $\cup$
- Set Intersection $\cap$
- Set Difference $\setminus$
- Symetric Difference $\ominus$
- Projection π_C
- Natural Join $r_1 \bowtie r_2$

Relational Operations

$$\begin{aligned}
\text{empty } () &= \text{ref } \emptyset \\
\text{insert } r\ t &= r \leftarrow !r \cup \{t\} \\
\text{remove } r\ s &= r \leftarrow !r \setminus \{t \in !r | t \supseteq s\} \\
\text{update } r\ s\ u &= r \leftarrow \{ \text{ if } t \supseteq s \text{ then } t \lhd u \text{ else } t | t \in !r\} \\
\text{query } r\ s\ C &= \pi_C\{t \in !r | t \supseteq s\}
\end{aligned}$$

Relational Operations

- insert r $\langle ns : 7, pid : 42, state : R, cpu : 0\rangle$
- query r $\langle ns : 7, pid : 42\rangle$ $\{state, cpu\}$
- update r $\langle ns : 7, pid : 42\rangle$ $\langle state : S\rangle$
- remove r $\langle ns : 7, pid : 42\rangle$

Relational Operations

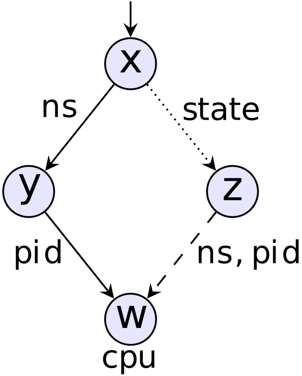
```
class scheduler_relation
{
  void insert(tuple_cpu_ns_pid_state const &r);

  void remove(tuple_ns_pid const &pattern);

  void update(tuple_ns_pid const &pattern,
              tuple_cpu_state const &changes);

  void query(tuple_state const &input,
             iterator_state__ns_pid &output);
  ...
};
```

Decomposition

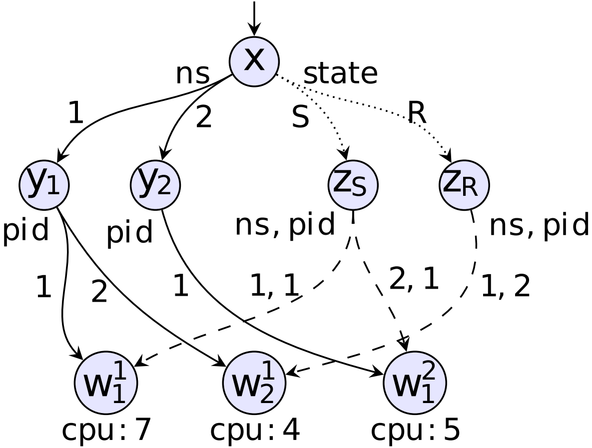
- Decompositions describe how to represent relations as a combination of primitive data structures.
- A decomposition is a static description of the structure of data, akin to a type.
- Its run-time (dynamic) counterpart is the decomposition instance, which describes the representation of a particular relation using the decomposition.

Decomposition

solid line - hash map
dotted line - vector
dashed line - dl. list



Decomposition Instance



Decomposition Language

$$\hat{p} ::= C \mid C \overset{\psi}{\mapsto} v \mid \hat{p}_1 \bowtie \hat{p}_2 \qquad \text{decomposition primitives}$$

$$\hat{d} ::= \text{let } v \colon C_1 \triangleright C_2 = \hat{p} \text{ in } \hat{d} \mid v \qquad \text{decomposition}$$

$$\psi ::= \text{dlist} \mid \text{htable} \mid \text{vector} \ldots \qquad \text{data structures}$$

Decomposition Language

$p ::= t \mid \{t \mapsto v_{t'}, \ldots\} \mid p_1 \bowtie p_2$	instance primitives
$d ::= \text{let } \{v_t = p, \ldots\} \text{ in } d \mid v_{\langle\rangle}$	data structures

Decomposition Language

$$\text{let } w : \{ns, pid, state\} \triangleright \{cpu\} = \{cpu\} \text{ in}$$

$$\text{let } y : \{ns\} \triangleright \{pid, cpu\} = \{pid\} \overset{htable}{\longmapsto} w \text{ in}$$

$$\text{let } z : \{state\} \triangleright \{ns, pid, cpu\} = \{ns, pid\} \overset{dlist}{\longmapsto} w \text{ in}$$

$$\text{let } x : \emptyset \triangleright \{ns, pid, cpu, state\} = (\{ns\} \overset{htable}{\longmapsto} y) \bowtie (\{state\} \overset{vector}{\longmapsto} z) \text{ in } x$$

Decomposition Language

$$\text{let } \{ \; w_{\langle ns:1, pid:1, state:S\rangle} = \langle cpu:7\rangle, \; w_{\langle ns:1, pid:2, state:R\rangle} = \langle cpu:4\rangle, \; w_{\langle ns:2, pid:1, state:S\rangle} = \langle cpu:5\rangle, \; \} \text{ in}$$

$$\text{let } \{ \; y_{\langle ns:1\rangle} = \{ \langle pid:1\rangle \mapsto w_{\langle ns:1, pid:1, state:S\rangle}, \; \langle pid:2\rangle \mapsto w_{\langle ns:1, pid:2, state:R\rangle} \}, \; y_{\langle ns:2\rangle} = \{ \langle pid:1\rangle \mapsto w_{\langle ns:2, pid:1, state:S\rangle} \}\} \text{ in}$$

$$\text{let } \{ \; z_{\langle state:S\rangle} = \{ \langle ns:1, pid:1\rangle \mapsto w_{\langle ns:1, pid:1, state:S\rangle}, \; \langle ns:1, pid:2\rangle \mapsto w_{\langle ns:1, pid:2, state:R\rangle} \}, \; z_{\langle state:R\rangle} = \{ \langle ns:2, pid:1\rangle \mapsto w_{\langle ns:2, pid:1, state:S\rangle} \}\} \text{ in}$$

$$\text{let } \{ \; x_{\langle\rangle} = \{ \langle ns:1\rangle \mapsto y_{\langle ns:1\rangle}, \; \langle ns:2\rangle \mapsto y_{\langle ns:2\rangle} \} \bowtie \{ \langle state:S\rangle \mapsto z_{\langle state:S\rangle}, \; \langle state:R\rangle \mapsto z_{\langle state:R\rangle} \}\} \text{ in } x_{\langle\rangle}$$

- Abstraction function
  - Computes the relation represented by a given decomposition instance.
- Well-formedness criteria
  - Check that a decomposition instance is a well-formed instance of a particular decomposition.
- Adequacy conditions
  - Which are sufficient conditions for a decomposition to faithfully represent a relation.

Querying and Updateing Decomposed Relations

Two basic kinds of relational operations:

- Queries
- Mutations

Queries

Queries are implemented in two stages:

- Query planning
  - Attempt to find most efficient execution plan for a query.
- Query execution
  - Evaluates a particular query plan over a decomposition instance.

This approach is well known in the database literature.

Query Planer implementatin in RelC

- Each strategy has a different computational complexity
- The query planner enumerates the alternatives and chooses the "best" strategy.

Query Plan

Query plan is a tree of query plan operators.

$q ::= qunit \mid qscan(q) \mid qlookup(q) \mid qlr(q, lr) \mid qjoin(q_1, q_2 lr)$

$lr ::= left \mid right$

Scheduler Example

- The query
  - query $r\langle ns : 7, pid : 42\rangle\{cpu\}$
- Possible query plan
  - $q_{cpu} = qlr(qlookup(qlookup(qunit)), left)$
- Perform query q_{cpu} on an instance d
  - $dexec\ q_{cpu}\ d\ \langle ns : 7, pid : 42\rangle$

Scheduler Example

- The query
  - query $r\langle ns : 7, state : R\rangle\{cpu\}$
- Possible query plan
  - $q_1 = qjoin(qlookup(qscann(qunit)), qlookup(qlookup(qunit)), left)$
  - $q_2 = qlr(qlookup(qscan(qunit)), rights)$

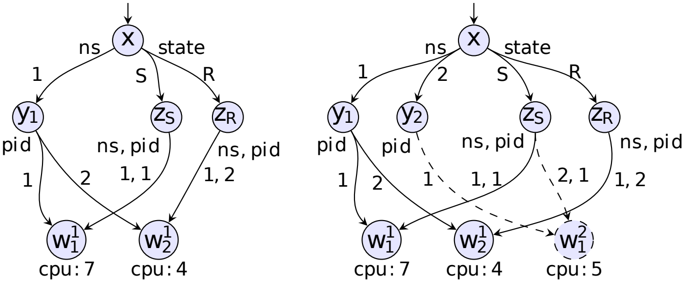
Query Validity

Not every query plan is a correct strategy for evaluating a query. We must check three properties:

- Query produce all of the columns requested as output.
- When performing a lookup all necessary key columns are available.
- Enough columns are computed on each side of a join.

Insert

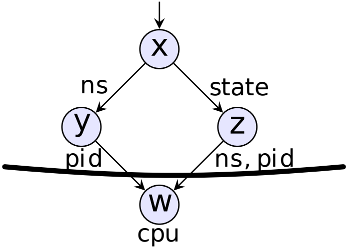
Perform insertion over the nodes of a decomposition in topological order.



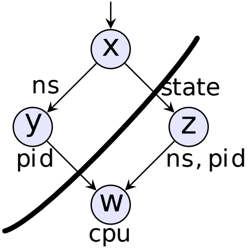
$t = \langle ns : 2, pid : 1, state : S, cpu : 5 \rangle$

Remove

Create a cut and remove nodes matching the tuple.



cut for columns $\{ns, pid\}$



cut for columns $\{state\}$

Update

- Semantically, updates are removal followed by an insertion
- Updates are performed inplace.
- Only common case is supported - no key columns.

Autotuner

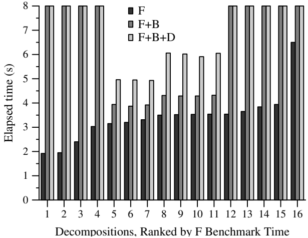
- Attemps to find the best decomposition of a relation.
- Takes as input a benchmark program, that produces as output a cost value, together with the relation to optimize.
- Constructs all possible decompositions up to a number of edges.

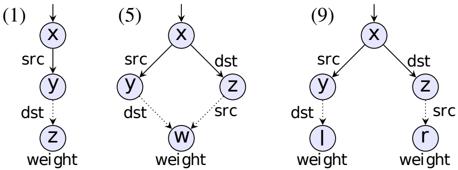
Experiments

- Micro-benchmarks
- Real World Systems.

Graph Benchmark

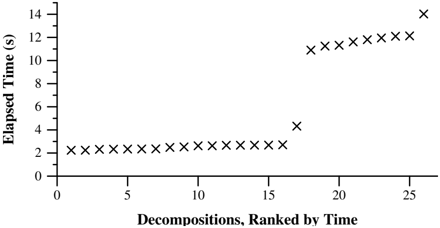
Relation $\{src, dst, weight\}$, $src, dst \rightarrow weight$.
Decompositions up to size 4.





Real World Benchmarks

	Original		Synthesis	
System	Everything	Module	Decomposition	Module
thttpd	7050	402	42	239
Ipcap	2138	899	55	794
ZTopo	5113	1083	39	1048



Experiments - Summary

Experiments show that:

- Different choices of decomposition lead to significant changes in performance
- The best performance is comparable to existing hand-written implementations
- The resulting code is concise and the soundness of the compiler guarantees that the resulting data structures are correct by construction.