Concurrent programming made simple

The (r)evolution of transactional memory

Torvald Riegel
Red Hat

Nuno Diegues
INESC-ID, Lisbon, Portugal

FOSDEM 2014

Concurrent programming

- Concurrent = at the same time and not independent
  - Concurrent actions need to *synchronize* with each other

Shared memory (synchronization)

\+ *Transactions*

= **Transactional memory (TM)**

- *Atomicity* enables synchronization
  - Example: atomic HW instructions such as x86 cmpxchg
  - Database folks: think atomicity + isolation

TM is a programming abstraction

- Underlying vision: Allow programmers...
  ... to *declare which code sequences are atomic*
  ... instead of requiring them to implement how to make those atomic.

- Generic implementation ensures atomicity
  - Not specific to a particular program
  - Purely SW, purely HW, or mixed SW/HW

- Our focus: TM for high-level programming languages

Agenda

- 1st part: TM for shared memory on a single machine
  - C/C++ language constructs
  - A peek into GCC's implementation
  - Some notes on performance
- 2nd part: TM for distributed shared memory (multiple machines)
  - Importance of strong transactions
  - A framework for distributed applications
- Q & A

TM is still rather new

- Proposed 20 years ago
- Substantive research started 10 years ago, and ongoing
- Standardization for C/C++ started 5 years ago
  - ISO C++ Study Group 5 on TM since mid 2012
- GCC support for C/C++ TM constructs since 4.7
- HW TM implementations: Azul, BlueGene/Q, ***Intel Haswell***

C/C++ language constructs

- Declare that compound statements must execute atomically
  - `__transaction_atomic { if (x < 10) y++; }`
  - No data annotations or special data types required
  - Existing (sequential) code can be used in transactions: function calls, nested transactions, ...
- Code in atomic transactions must be *transaction-safe*
  - Compiler checks whether code is safe
  - Unsafe: use of locks or atomics, asm, volatile, functions not known to be safe
  - For cross-CU calls / function pointers, annotate functions:
    ```
    void foo() __attribute__((transaction_safe)) { x++; }
    ```
- Further information: ISO C++ paper N3718

Synchronization semantics

- Transactions extend the C11/C++11 memory model
  - All transactions totally ordered
  - Order contributes to memory model's *happens-before*
  - TM ensures some valid order consistent with happens-before
  - Does not imply sequential execution at runtime!

- Data-race freedom still required (as with locks,...)

```
init(data); __transaction_atomic { data_public = true; }
```

Correct:

```
__transaction_atomic {
    if (data_public) use(data); }
```

Incorrect:

```
__transaction_atomic { temp = data;  // Data race
    if (data_public) use(temp); }
```


TM supports modular programming

- Programmers don’t need to manage association between shared data and synchronization metadata (e.g., locks)
  - TM implementation takes care of that :-)
- Functions containing only txnal synchronization compose without deadlock
  - Nesting order of transactions does not matter
  - But can’t expect another thread to make progress in an atomic transaction!
- Example: Synchronize moving an element between lists

```
void move(list& l1, list& l2, element e)
{ if (l1.remove(e)) l2.insert(e); }
```

  - TM: `__transaction_atomic { move(A, B, 23); }`
  - Locks: ?

GCC's implementation: Compiler

- Ensure atomicity guarantee (at compile time!)
  - Find all transaction-safe code (implicitly or by annotation)
  - Check that transaction-safe code is indeed safe
- Create an instrumented clone of all transactional code
  - Transaction-safe functions, code in transactions
  - Memory loads/stores rewritten to calls to *TM runtime library*
  - Function calls redirected to instrumented clones
  - Result: both an instrumented and uninstrumented code path
- Generate begin/commit code for each transaction
  - Runtime library decides whether to execute instrumented or uninstrumented code path
- Delegation to runtime library = implementation flexibility

GCC's implementation: TM runtime library (libitm)

- Enforces atomicity of transactions at runtime
- libitm contains different SW-only implementations (STM)
  - Do not need special hardware
  - Default:
    - Write-through with undo logging
    - Multiple locks (automatic memory-to-lock mapping)
    - Uses instrumented code path
- Using HW TM implementations (HTM)
  - Current HTMs are all best-effort
    - Not able to execute all txns, thus need a fallback (e.g., STM)
  - libitm uses HTM with a global lock as fallback
    - HW transactions use uninstrumented code path
  - No hybrid STM/HTM yet

Performance: It's a tool, not magic

- Performance goal:
  A useful balance between ease-of-use and performance

- Not meaningful to try to draw conclusions about TM performance *today*
  - Implementations are work-in-progress (e.g., libitm, HTMs, ...)
  - Performance heavily influenced by many factors
    - HW, compiler, TM algorithm, HTM implementation, allocator, LTO or not, ...
    - Txn conflict probability, txn length, load/store ratio in txns, memory access patterns, data layout, allocation patterns, other code executed in txns, ...
  - Tuning for real-world workloads: chicken-and-egg situation

Performance: Rough estimates that are probably still true in the future

- Single-thread performance
  - STM slower than sequential
  - STM slower (or equal) to coarse locking
  - HTM about as fast as uncontended critical section
    - If HTM can run the transaction
- Multiple-thread performance
  - STM scales well
    - But less likely if low single-thread overhead
  - HTM scales well
    - Unless slower fallback needs to run frequently
  - Hybrid STM/HTM: hopefully HTM performance with a fallback that scales
- TM runtime libraries can adapt at runtime!

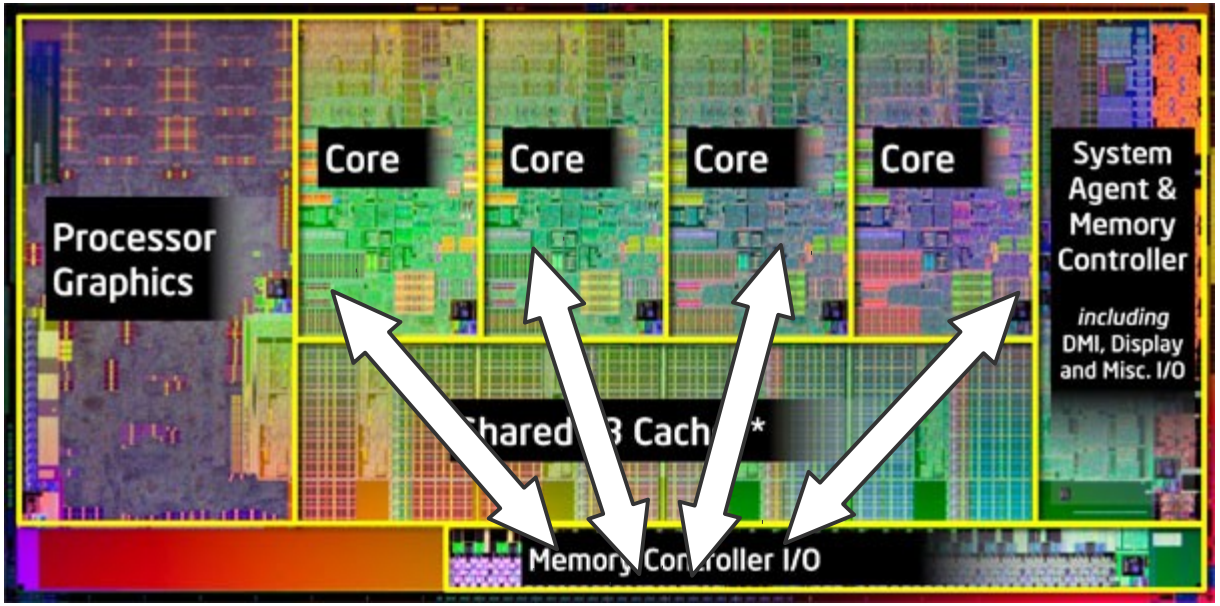
Ways to get involved

- Use it
  - Try it out (gcc -fgnu-tm), measure performance for your code, read the C++ specification (N3718 / N3859), ...
- Report about your findings and experience
  - Blog about it and let us know, report bugs in the GCC implementation, ...
- Get involved in ISO C++ TM standardization (SG5)
  - http://isocpp.org/forums
- Dive into libitm / GCC
  - Extensive comments in the libitm code
  - Many interesting things to work on (e.g., improving the (auto-)tuning)

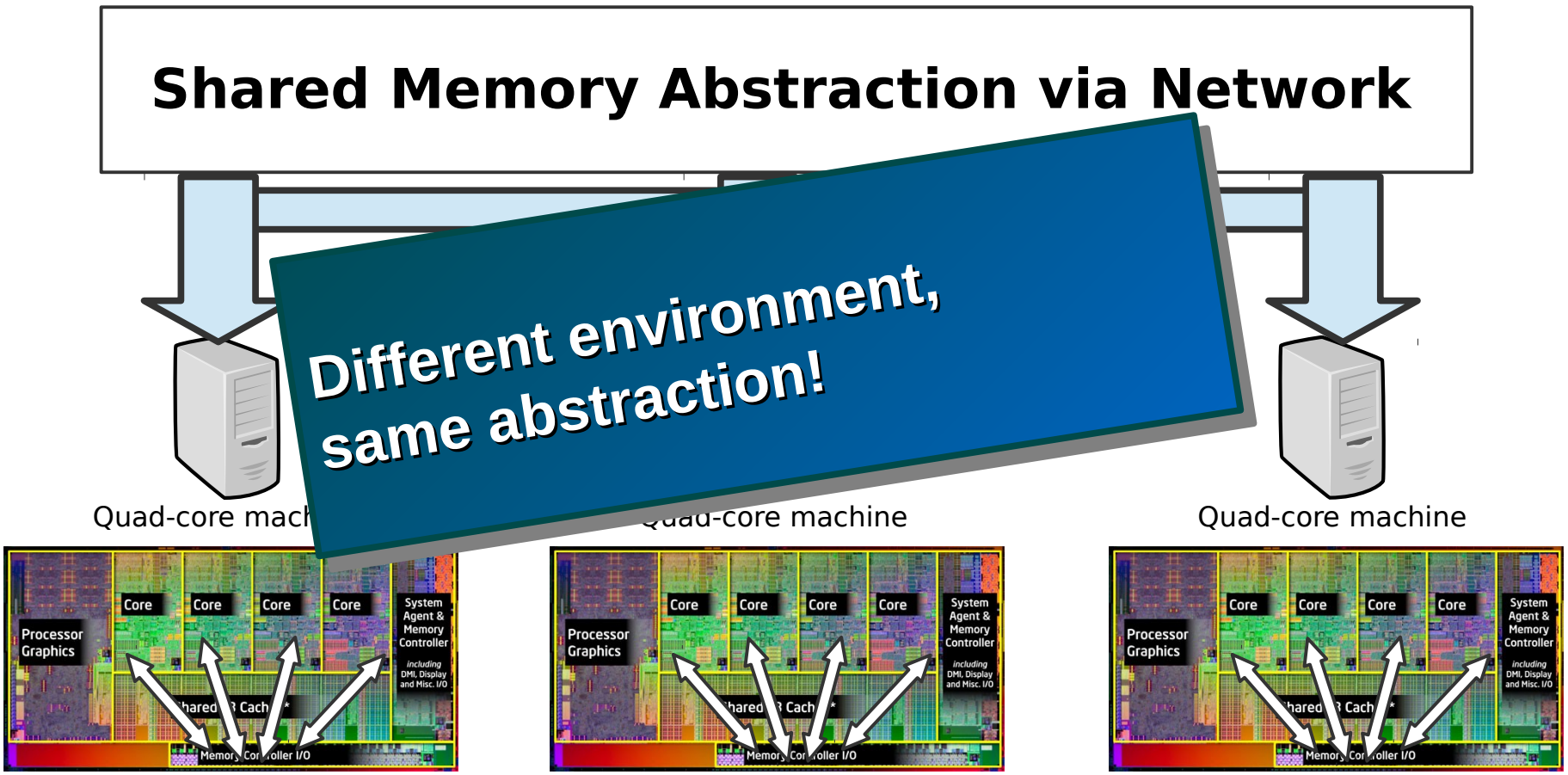
The Cloud-TM Approach

Moving to a distributed world

Quad-core machine



Moving to a distributed world

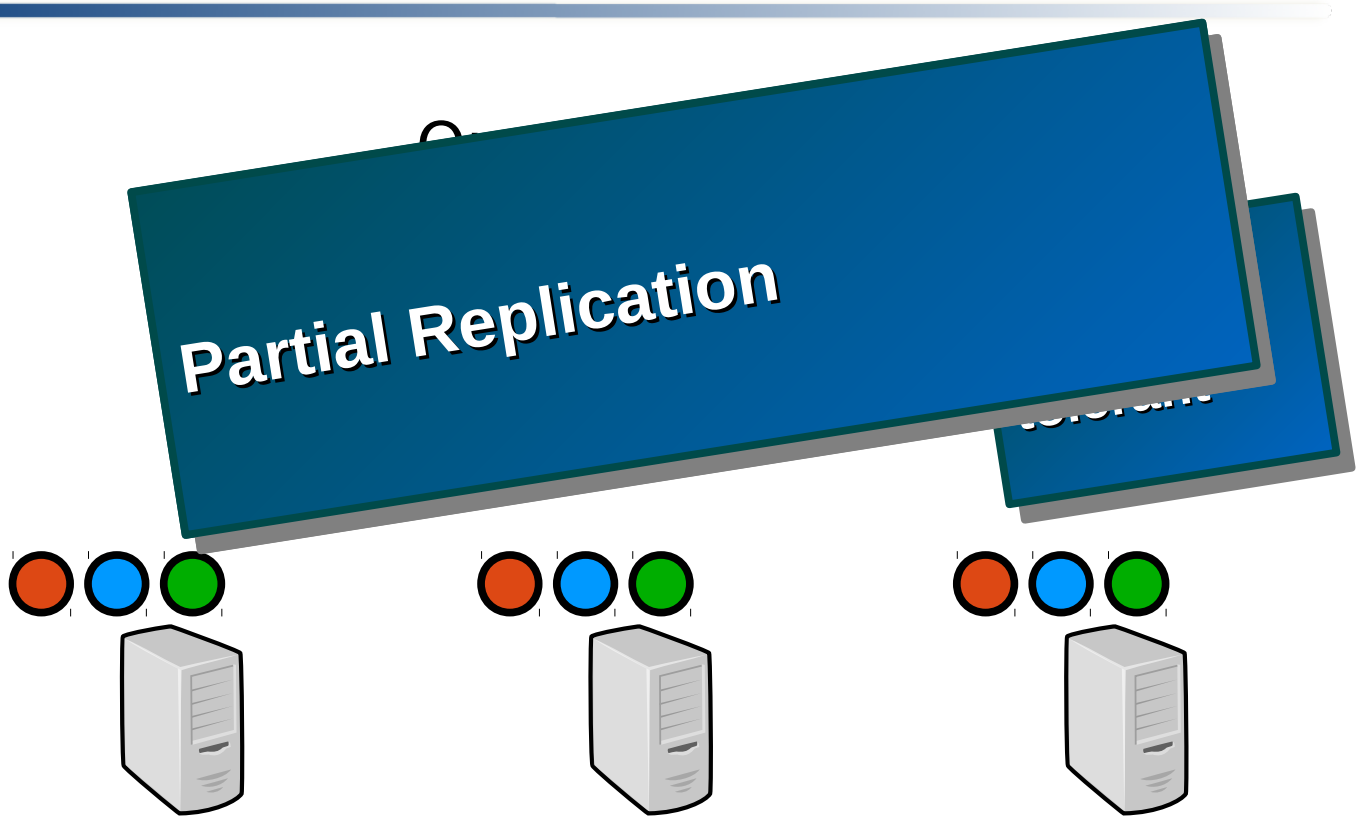


Distributed Transactional Memory

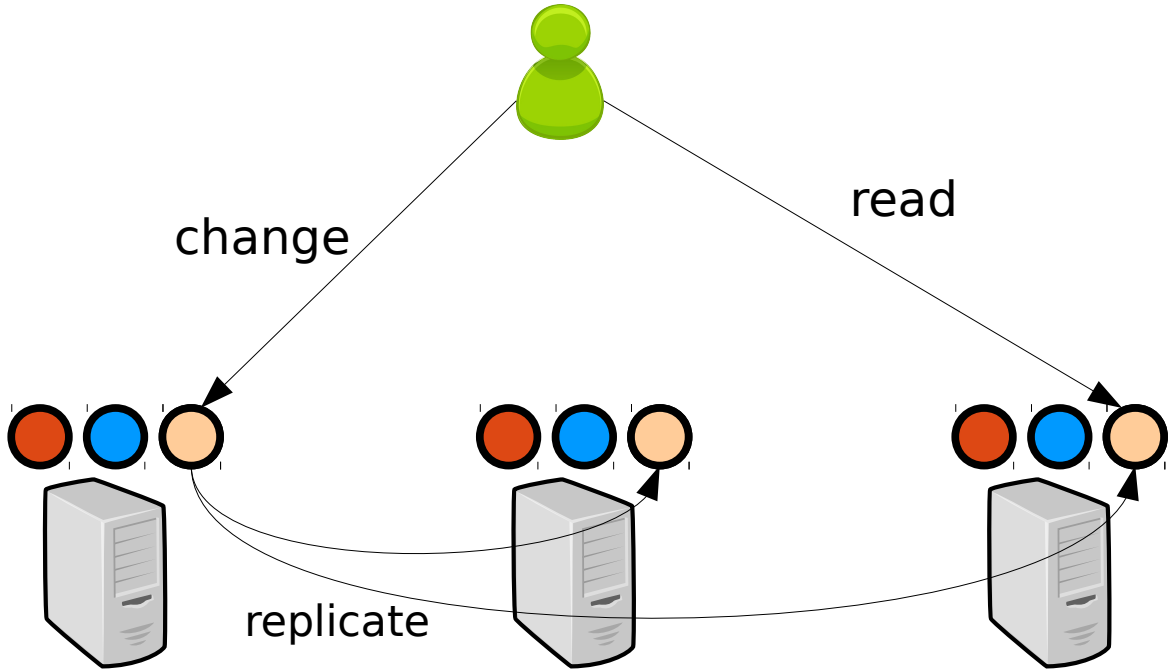
- Similarly to TM:
  - Bring transactions to the top of the stack
  - Dynamic transactions
  - Straight in the app logic
  - Long-lived transactions

- Different from TM:
  - Persistence
  - Distribution
  - Fault-tolerance

Distributing Data



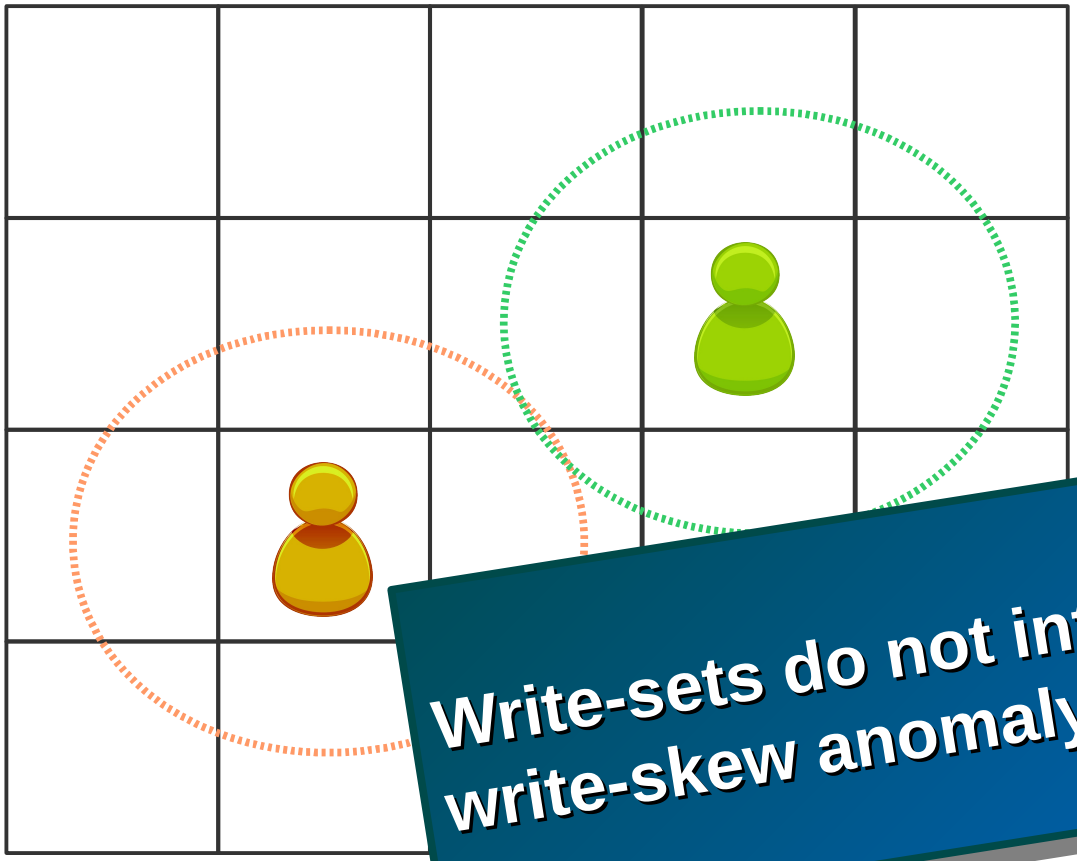
Why strong consistency?



Eventual Consistency → no consistency

Why serializable transactions?

Snapshot Isolation



Write-sets do not intersect: write-skew anomaly

The Cloud-TM Approach

Embraces distribution

- Serializable transactions
- Partial replication
- Scalable solution

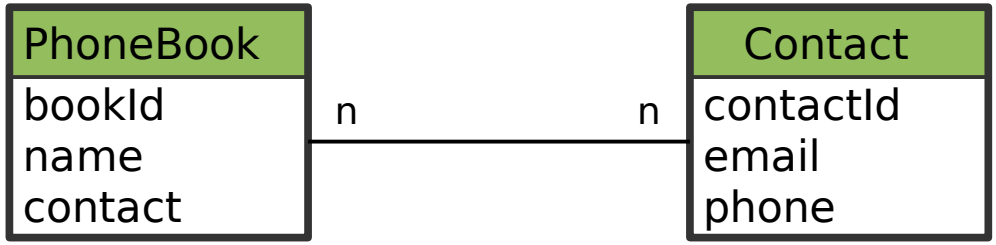
Targets many common use cases

- Simple bootstrap
- Details hidden from programmer
- Easy management
- Fast/scalable enough

The Cloud-TM Approach

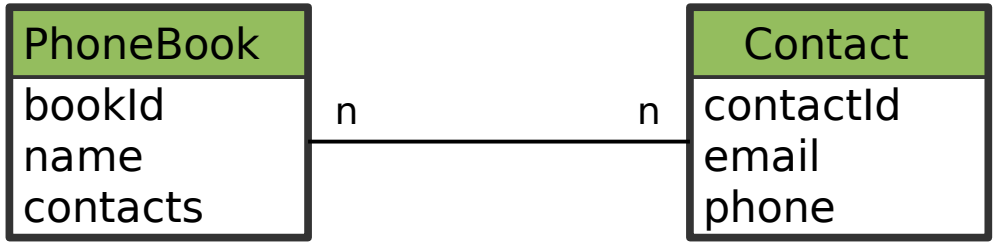
- DSL to specify Object-Oriented domain model
- Hides:
  - Concurrency control
  - Persistence
  - Data Placement
- OO view of:
  - Distributed execution
  - Data locality
- API for expert programmers

From design to code



- Entities → (Java) Classes
- Relationships → Collections/References
  - Bidirectional updates
  - Type of collection used
  - ...

From design to code




```
@Entity
class PhoneBook {
    @Id @Gen
    public St

    public St

    @ManyToMan
    public Set
}
```

```
@Entity
                                    lue
                                    tactId;
                                    il;
                                    ne;
@ManyToMany(mappedBy="contacts")
public Set<PhoneBook> books;
}
```

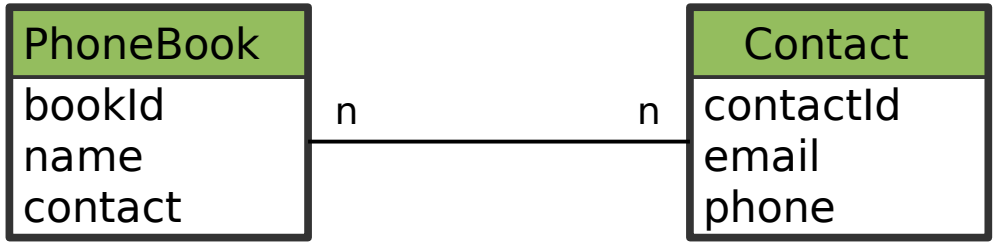
Can we make it simpler?

Domain Modeling Language

- Programmer describes:
  - Entities
  - Relationships

- DML is a DSL

From design to code




```
class PhoneBook {
    String name;
}
```

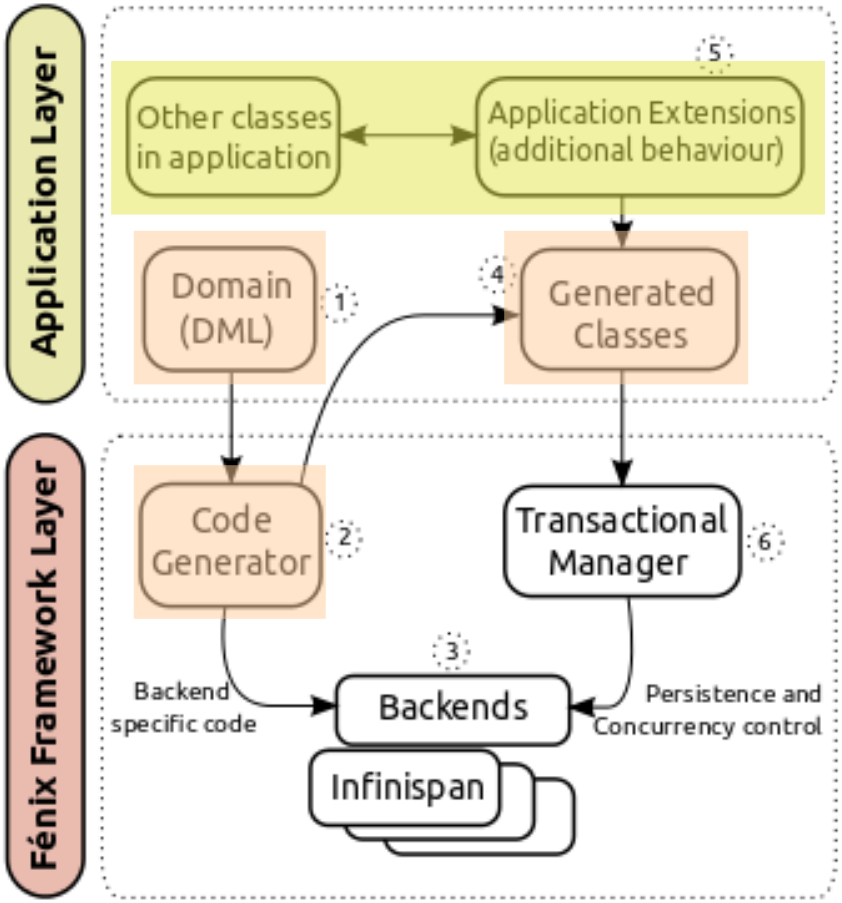
```
class Contact {
    String email;
    String phone;
}
```

```
relation PhoneBooksHaveContacts {
    PhoneBook playsRole book { multiplicity *; }
    Contact playsRole contact { multiplicity *; }
}
```

Domain Modeling Language

- Programmer describes:
  - Entities
  - Relationships
- DML is a DSL
- Framework generates code with:
  - Well known interface
  - Entities → Classes with getters / setters
  - Relations → Management methods on both sides

Generating the code



Generating the code

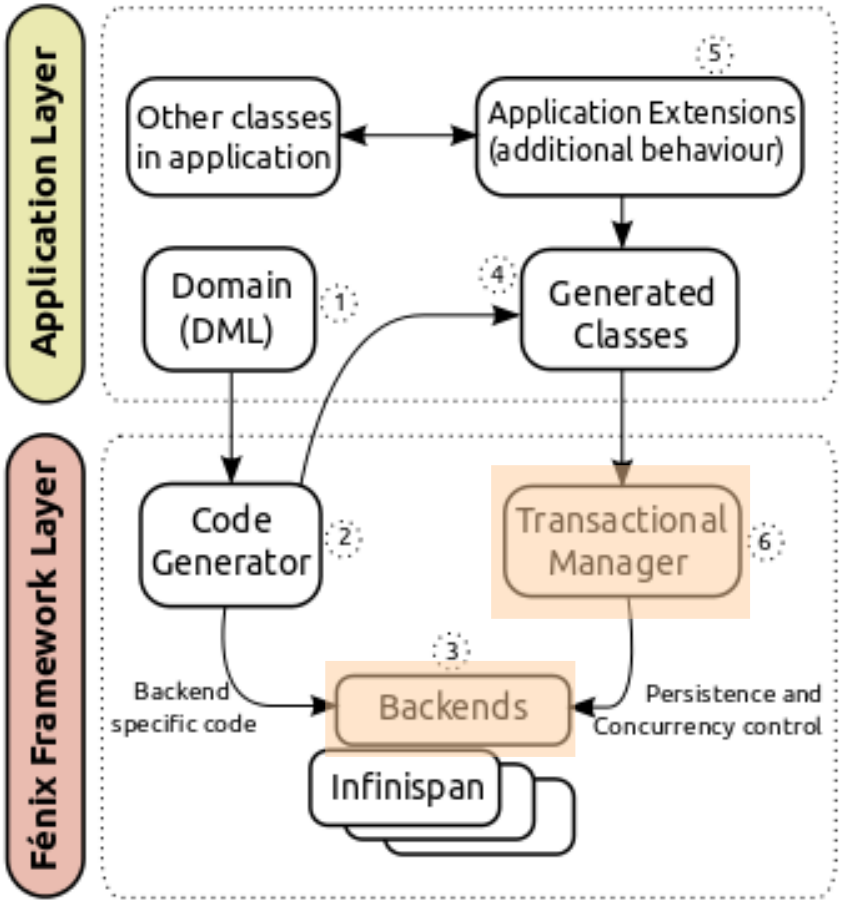
```
class PhoneBook {
    String getName();
    void setName(String name);
    Set<Contact> getContactSet();
    void addContact(Contact contact);
    void removeContact(Contact contact);
    ...
}

class Contact {
    String getEmail();
    void setEmail(String email);
    Set<PhoneBook> getBookSet();
    void addBook(PhoneBook book);
    void removeBook(PhoneBook book);
}
```

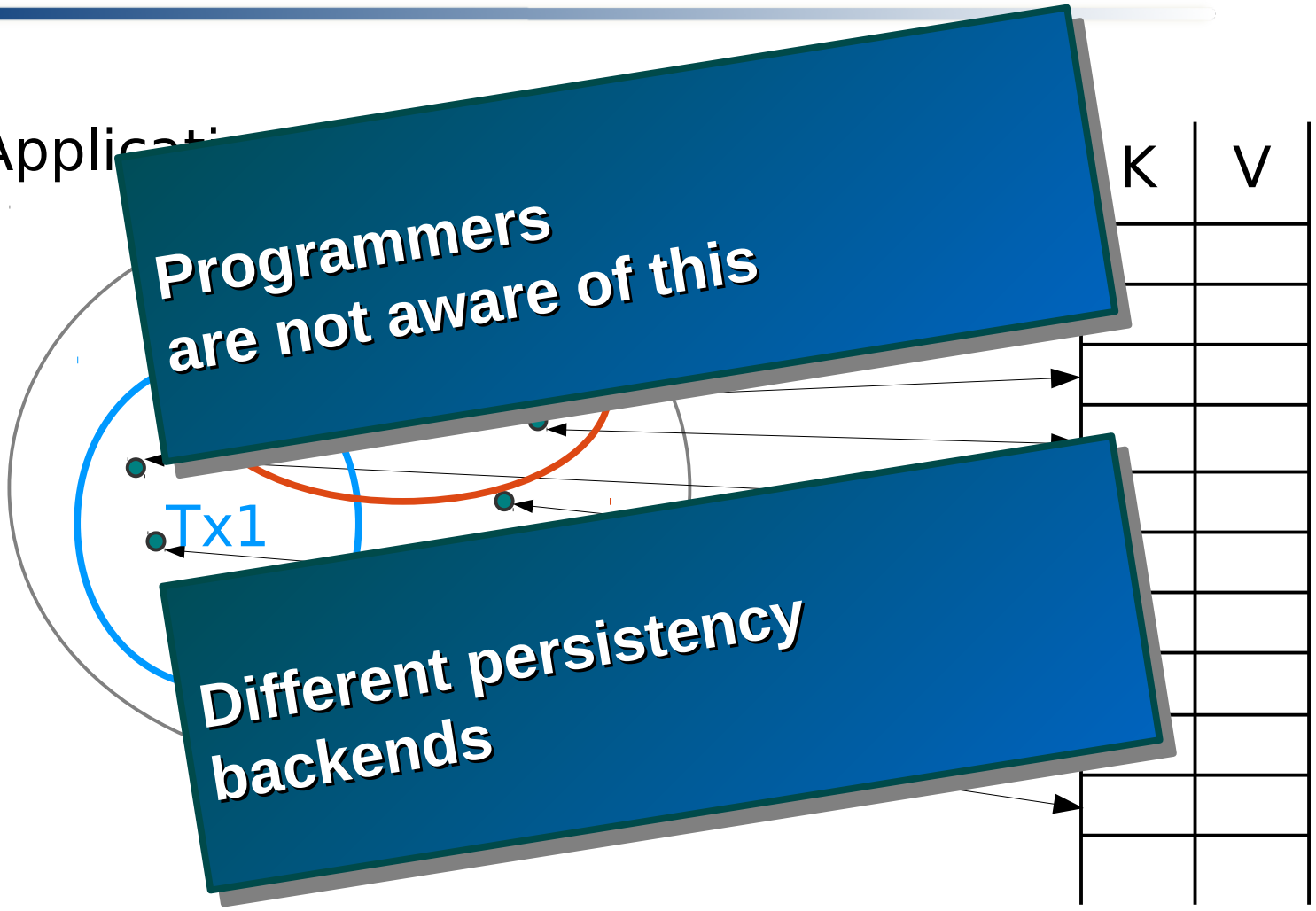
Interface remains the same

Actual code varies

Generating the code



Mapping to Persistence



Using the code

```
@Scoped(REQUEST)
public class BookService {
    @Inject EntityManager em;

    @Transactional
    public PhoneBook createBook(String name) {
        PhoneBook book = new PhoneBook();
        book.name = name;
        em.persist(book);
        return book; }

    @Transactional
    public addContact(String bookId, String email) {
        PhoneBook book = em.find(PhoneBook.class, bookId);
        Contact contact = new Contact();
        contact.email = email;
        contact.addBook(book);
        book.addContact(contact);
        em.persist(contact);
    }
}
```

Using the code

```
public class BookService {


    @Atomic
    public PhoneBook createBook(String name) {
        PhoneBook book = new PhoneBook();
        book.setName(name);

        return book; }

    @Atomic
    public addContact(String bookId, String email) {
        PhoneBook book = App.getBookById(bookId);
        Contact contact = new Contact();
        contact.setEmail(email);

        book.addContact(contact);

    }
}
```

Configuring the App

fenix-framework.properties:

```
appName = phonebook

# configurations for backend-infinispan
ispnConfigFile = infinispan.xml
expectedInitialNodes = 2
jgroupsConfigFile = jgroups.xml
```

Configuring the App

jgroups.xml:

```
<config>
    <TCP
        ...
        />
    <MERGE2
        ...
        />
    <UNICAST2
        ...
        />
    ...
</config>
```

Pre-built configurations

Configuring the App

ispn.xml:

```
<infinispan>
    <default>
        <locking isolationLevel="SERIALIZABLE"/>
        <transaction lockingMode="OPTIMISTIC"/>
        <clustering mode="distributed">
            <l1-cache enabled="true"/>
        </clustering>
        <versioning
                enabled="true"
                versioningScheme="GMU" />
    </default>
    <namedCache name="phoneCache" />
</infinispan>
```

Configuring the App

ispn.xml:

```
<infinispan>
    <default>
        <locking
            isolationLevel="REPEATABLE_READ"
           writeSkewCheck="true"/>
        <transaction lockingMode="PESSIMISTIC"/>
        <clustering mode="replicated"/>
        <versioning
                enabled="true"
                versioningScheme="SIMPLE" />
    </default>
    <namedCache name="phoneCache" />
</infinispan>
```

Comp

Advanced Features

- Indexed relations
- Data (co-)location
- Data-Oriented execution
- Hibernate Search
- Ghost reads
- Concurrency-friendly collections
- L1 and L2 object caches

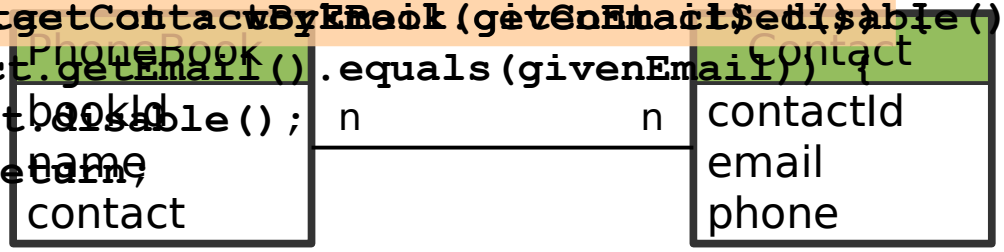
Indexed Relations

```
class PhoneBook {
    String name;
}

class Contact {
    String email;
    String phone;
}
```

```
relation PhoneBooksHaveContacts {
    PhoneBook playsRole book { multiplicity *; }
    Contact playsRole contact {
        multiplicity *;
        indexed by email;
    }
}
```

```
PhoneBook workBook = (…) // get the book
Contact ct = workBook.getContactByEmail(givenEmail);
if(ct.getEmail().equals(givenEmail)) {
    ct.disable();
    return;
}
```



Data Location

```
int countryCode = extractCode(phone);
LocalityHints hints = new LocalityHints("code", countryCode);

Contact ct = new Contact(email, phone);  // where does it go?
```

Data-Oriented Execution

```
public class DistributedTask implements Callable, Serializable {
    Contact ct;

    public DistributedTask(Contact ct) { this.ct = ct; }

    @Atomic
    public void call() {
        // execution is migrated to machine replicating "ct"
    }
}

(…)

    Callable task = new DistributedTask(ct);
    executor.submit(task, FF.getLocality(ct));
```

Data (Co-)Location

```
int countryCode = extractCode(phone);
LocalityHints hints = new LocalityHints("code", countryCode);

// use hints to place it
Contact ct = new Contact(hints, email, phone);

// ct and otherCt are co-located
Contact otherCt = new Contact(hints, emailOther, phoneOther);
```

By default objects of a given relation are co-located

In summary

- Distributed programs are hard to develop
- Low-level mechanisms make it harder
- API should hide complexity, whenever possible
- Do not limit expressiveness, whenever needed

References

Cloud-TM stack:

- www.cloudtm.eu

Fénix Framework:

- http://fenix-framework.github.io

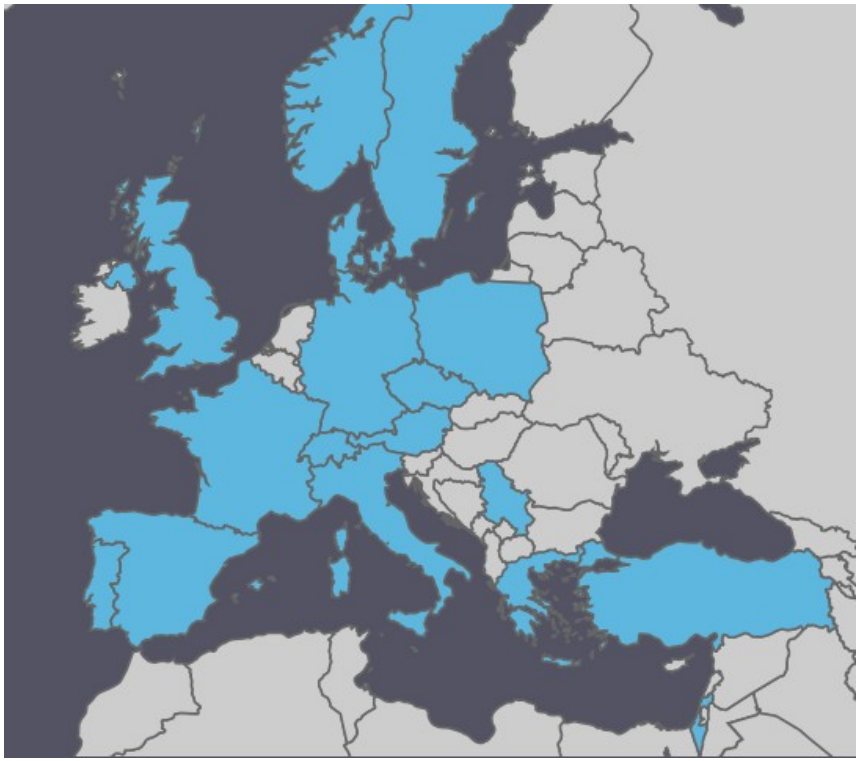
FénixEdu:

- http://fenixedu.org

Get involved..!

EuroTV

- Research network: Industry + Academia



If you are interested in TM:
Euro-TM can fund your participation

- Workshops: Amsterdam 13 April'14
- Training Schools: La Plagne 16-21 March'14
- Collaborations / Joint works
  - 1-3months to visit another EU institution

www.eurotm.org