Making Object-Based STM Practical in Unmanaged Environments

Torvald Riegel and Diogo Becker de Brum
(Dresden University of Technology, Germany)
## Background: Word-based or Object-based?

| Location to metadata mapping | Hash into array:  
- variable, can be tuned  
- But on fast path  
| In-place metadata  
- Fixed mapping  
| Hash into array also possible |
| Cache locality | Potentially miss in data and in metadata | Data and metadata potentially share cache line vs. perhaps more misses for nontxnal code |
| False conflicts | Depends on hash but not unlikely | Probably unlikely if objects are small |
| Space overhead | Seems to need large metadata arrays | Fixed per object (might get large) |

Choice **depends on the workload** and on **tuning quality**
Object-based has advantages that we should not ignore
Object-based might have advantages: Can we actually choose?

**TM must be practical**
- Manual load/store transactification is **not** sufficient
- Annotations are error-prone and less composable
- Programmer should not need to choose

**Unmanaged environments matter**
- A lot of server code in C / C++
- Desktop apps too (e.g., KDE)!

**Need compiler support!**
- Managed Environments (Java, C#, …): environment provides object-abstraction for memory
- Unmanaged Environments (C, C++, …): no ready-to-use object abstraction 😞

“Making Object-Based STM Practical in Unmanaged Environments”
Contributions

Show how to **use pointer analysis** to detect which memory chunks / pointers can be safely used for object-based accesses

- No programmer-supplied annotations necessary
- Programming language is not extended nor restricted
- Use object-based accesses where this is safe, fall back to word-based otherwise
- Integrated into Tanger (STM Compiler for C/C++/..., [Transact 07])
- Enables TinySTM [PPoPP 08] to provide object-based accesses (in-place and external metadata)

**Performance results** for word-based vs. object-based
analysis

- We use **Data Structure Analysis (DSA [1])**:
  - **Pointer analysis** for LLVM compiler framework
  - Creates a points-to graph with Data Structure (DS) nodes
  - Context-sensitive:
    - Data structures distinguished based on call graphs
  - Field-sensitive:
    - distinguish between DS fields
  - Unification-based:
    - Pointers target a single node in the points-to graph
    - Information about pointers from different places get merged
    - If incompatible information, node is collapsed (= “nothing known”)
  - **Can safely analyze incomplete programs**:
    - Calls to external / not analyzed functions have an effect only on the data that escapes into / from these functions (get marked “External”)
    - Analyzing more code increases analysis precision

Analysis (2)

Integration into Tanger compilation process:
1. **Compile and link** program parts into LLVM intermediate representation module
2. **Analyze** module using DSA
   - Local intra-function analysis: per-function DS graph
   - Merge DS graphs bottom-up in callgraph (put callees’ information into callers)
   - Merge DS graphs top-down in callgraph (vice versa)
3. **Transactify** module
   - Use DSA information to decide between object-based / word-based
   - Requirement: If memory chunk (DS node) is object-based, then it must be safe for object-based everywhere in the program
   - DSA can give us this guarantee
4. Link in STM library and **generate native code**
Example: STAMP Vacation

Type, if known

struct has 4 fields, 2 are pointers

A Red-Black Tree instance

Collapsed: accessed in different ways (e.g., as void* and as int)

HMR: onHeap, Modified, Read

Partial, simplified DS graph for main()
How many object-based accesses?

- Memory chunks that we consider for object-based accesses:
  - All uses must have been analyzed (e.g., doesn’t escape to external function): required for safety
  - Not an array (in-place metadata would be more tricky, but could be possible in some cases)
  - Not a primitive type (probably not useful, but is possible)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Object-based loads/stores (static)</th>
<th>Object-based loads/stores at runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red-Black Tree</td>
<td></td>
<td>Only object-based accesses</td>
</tr>
<tr>
<td>Linked List</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STAMP Vacation</td>
<td>90% / 95%</td>
<td>97% / 84%</td>
</tr>
<tr>
<td>STAMP KMeans</td>
<td></td>
<td>No object-based accesses</td>
</tr>
<tr>
<td>STAMP Genome</td>
<td>40% / 48%</td>
<td>12% / 98%</td>
</tr>
</tbody>
</table>
STM Implementations + Transformations

Word-based (TinySTM):
- Time-based reads, blocking writes
- Metadata: array of versioned locks: lock = (address >> shifts) % locks:
- More details: see PPoPP 08 paper
- \%val = load %addr transformed to \%val = call STMLoad(%addr)

Object-based (where possible, otherwise fall back to word-based)

External metadata (TinySTM-ObjE):
- Metadata:
  - Reuse word-based lock array
  - Use base address of object to select lock from lock array
- \%val = call STMLoad(%addr, %baseaddr)

In-place metadata (TinySTM-Obj):
- Metadata:
  - Single per-object versioned lock
  - Located after the end of the object
- Compiler must enlarge all memory allocations accessed in an object-based way and initialize metadata
- \%val = call STMLoad(%addr, %baseaddr, %objSize)
Microbenchmarks: Tuning

External metadata (TinySTM, TinySTM-ObjE):
- different “shapes”

In-place metadata (TinySTM-Obj):
- flat “shape” (lock array not used)
- usually better performance
Lock arrays: Tuning difficulties

(Cuts with “enough locks”, performance relative to object-based, in-place metadata)

- Bad tuning is costly
- Best tuning configuration can easily change (e.g., tree gets larger)
- Tuning trade-offs: one data structure benefits, another loses
- Object-based, in-place often better or equal and not affected!
Application has word-based and object-based accesses.

Performance degrades even for TinySTM-Obj if word-based accesses suffer from false conflicts.

TinySTM-ObjE is just slightly better than TinySTM.
30% throughput increase for object-based with in-place metadata (TinySTM-Obj)

Possible reasons:
- App has different data structures:
  No tuning trade-offs
- Large working set:
  Better cache locality
Conclusion

- STM compilers for unmanaged environments can target object-based STMs without requiring anything special from the programmer.
- Pointer analysis very useful.
- Object-based accesses can have better performance:
  - In-place metadata less sensitive to tuning / false conflicts.
  - Potentially improved cache locality.

- [http://tinystm.org](http://tinystm.org)