From Atomic Variables to Data-Centric Concurrency Control*

Hervé Paulino Daniel Parreira Nuno Delgado António Ravara
NOVA Laboratory for Computer Science and Informatics
Departamento de Informática, Faculdade de Ciências e Tecnologia
Universidade NOVA de Lisboa, Portugal

Ana Almeida Matos
SQIG – Instituto de Telecomunicações
Universidade de Lisboa

ABSTRACT

The mainstream use of concurrent programming is bound to the provision of constructs that abstract details intrinsic to concurrency, while ensuring safety and liveness properties. Several control-centric approaches meet these requirements but decentralise concurrency management, hindering reasoning. The alternative data-centric approach promotes local rather than distributed reasoning, however it is a fairly new approach embraced only by *Atomic Sets* [4,8], a rather complex model that does not guarantee progress in all scenarios. In this paper we propose a simple data-centric concurrency control model that builds only on the notion of *atomic variable*. We informally present the model and its properties, as well as a prototype implementation that we used to compare our approach against the aforementioned *Atomic Sets* and control-centred approaches in general.

CCS Concepts

•**Software and its engineering** → **Concurrency control; Concurrent programming languages;**

Keywords

Concurrency control, Data-centric, Java Language

1. INTRODUCTION

Parallel processing hardware is currently present in nearly all computational systems, from the nodes that compose cloud infrastructures to small hand-held devices. Concurrent programming thus plays a fundamental role in software development. However, concurrent code is considered difficult to write and its correctness complex to assess, being hence the source of many execution errors [11]. .

*This work was partially funded by FCT-MEC in the framework of UID/CEC/04516/2013 strategic project

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC 2016,April 04-08, 2016, Pisa, Italy

Copyright 2016 ACM 978-1-4503-3739-7/16/04…$15.00

http://dx.doi.org/10.1145/2851613.2851734

Classically, expressing concurrency constraints on accesses to shared memory objects requires delimiting explicitly the sequences of instructions that must operate atomically upon such objects, what we refer to as *control-centric concurrency control*. The main drawback of control centrality is that it is prone to the dispersal of concurrency related bugs, which hampers reasoning about correctness as the code scales: a single missing lock/unlock operation (or atomic keyword) may adulterate an application's behaviour. Relevant work has addressed correctness in this context, with focus on progress and isolation [1, 6, 7, 10, 13] and, on atomicity and protocol compliance [3,5]. Despite these efforts, a study [11] identifies that 97% of the non-deadlock errors stem from either atomicity or protocol violation.

Data-centric concurrency control is a recent approach that promotes local rather than distributed reasoning, by centralising all concurrency control management on data declaration. We are aware of only one approach to data-centric concurrency [4,8], which builds on the notion of atomic set, a set of memory objects that share consistency properties. The model is complex and does not guarantee progress and atomicity in all scenarios. We share the data-centric viewpoint of concurrency control, but envision a rather distinct approach to the problem. To that extent we introduce the Resource-Centred Concurrency Control (RC^3) model that builds *only* upon the individual annotation of the data items (the resources) that must be atomically manipulated, to provide a simpler model for data-centric concurrency control.

The contributions of this paper are: 1. an informal presentation of the RC^3 model and of its properties (in §3); 2. an overall description of how the model may be implemented in the Java language (in §4), and; 3. an evaluation of RC^3 from a productivity and performance perspective, comparing it against Java's native synchronisation mechanisms, Atomic Sets and a well-known Java STM (in §5).

2. BACKGROUND

Data-centric concurrency control is a rather recent research area that has been lead by Atomic sets [4,8]. In such model, the programmer does not have to reason about the synchronisation of execution flows, but rather about which memory locations share consistency properties. Accordingly, variables may be marked as belonging to a given set. For that purpose, the programmer must first create an atomic set (through the `atomicset` construct) and next bind the vari-

able to such set, by prefixing its declaration with `atomic(s)`, where s denotes the set identifier. Associated to groups are *units of work*, code fragments that maintain the consistency properties of the associated sets. In order to express which memory objects must be acquired by a unit of work, the `unitfor` annotation enables the union of the atomic sets of such unit with the ones derived from the method's parameters. Similarly, alias annotations (`b=this.a`) allow the union of atomic sets from distinct classes at object creation.

We argue that most of the concerns associated to these annotations should be subsumed either by design or at compile-time. The considerable number of annotations of Atomic Sets, with distinct semantics and applications, hinders reasoning and is error-prone, e.g. `unitfor` and aliasing annotations are easily forgotten. As a result, the benefits of data-centrality are overshadowed by the complexity of the specification. Moreover, Atomic Sets do not guarantee progress in all scenarios. Deadlocks may arise when in the presence of transitive circular dependencies between sets. Recent work has addressed the problem [2], but the programmer is called to intervene when the analysis cannot infer a partial order between sets – a new ordering annotation allows for the explicit definition of such order.

Given the complexity of Atomic Sets, there has been some work on their automatic inference. AJ-lite [12] is a lighter version of AJ that assumes a single atomic set per Java class. The number of annotations are reduced to three - the ones needed to express the relation between a class' atomic set and the ones of its fields. Nonetheless, the programmer must still reason about behaviours such as: "*a particular class field only stores values that share consistency properties with its hosting class*" or "*it stores a value that shares consistency properties with other classes*", or even "*I do not know if it shares consistency properties with other classes*". Although the authors claim that the number of annotations needed to express data-centric concurrency control in a collection of classes is considerably reduced, potential for error is still there. Furthermore, the current solution is only applicable to libraries and not entire programs.

In [9], the authors propose an initial step for automating the inference of atomic sets. The approach is to process execution traces and from them identify patterns in the access to class fields (currently only fields are supported). These patterns are then used to automatically form the atomic sets. The reported experimental results attest that the automatic solution inferred most of the manually placed annotations, having however also generated more annotations than necessary. Also to be noted is the fact that the final result is sensitive to the quality of the input traces. Lastly, the compilation time increased from seconds to several minutes.

3. THE RC3 MODEL

In this section we informally define the syntax and the semantics of the RC^3 model. A couple of simple, yet expressive examples illustrate the main ideas and intended properties. We assume a Java-like language and add only an extra keyword – `@Atomic` – which is a type declaration, and a mechanism for automatically generating locks (ensuring deadlock-freedom). This simple annotation says which data items must be accessed atomically in a method's execution, *i.e.*, allow to declare atomic variables. This annotation may be applied to all variable declarations (class fields, local variables, and parameters) and to the return type of methods.

3.1 Semantics

An *atomic resource* (or briefly resource) is a data item that can be attained from an atomic variable in the scope of a method. Our main goal is to ensure that concurrent accesses to atomic resources happen in *mutual exclusion*.

We assume a multi-threaded Java where methods are the "unit of execution". Therefore, if two methods may concurrently access a resource and at least one of them writes in the resource; the methods must execute in *mutual exclusion*. Assuming that no thread abruptly terminates during the execution of a method, and that all exceptions raised during the execution are adequately handled, then all resource modifications applied by that method are perceived as one instantaneous operation to the remainder of the system.

Running examples: To illustrate the expressiveness and properties of the model we present two running examples. The first is the paradigmatic scenario of concurrent money transfers between bank accounts. Naturally, these operations must be executed atomically, in order to ensure the consistency of the accounts' balances. The solution presented in Listing 1 uses the simple assignment of accounts to atomic variables (i.e. their declaration as atomic resources) to guarantee the atomicity property. Consequently, `transfer` operations that operate upon the same must accounts execute in mutual exclusion. Figure 1 showcases the execution of five `transfer` operations distributed over six accounts. As desired, synchronisation is performed at account level – operations over distinct pairs of accounts do not require coordination. For example, `transfer(acc1,acc2)` and `transfer(acc3,acc4)` may execute in parallel, while `transfer(acc1,acc2)` and `transfer(acc1,acc3)` may not, because they share (and modify) `acc1`. The same happens for `transfer(acc3, acc4)` and `transfer(acc1, acc3)` with regard to account `acc3`.

```
1  class Bank{
2   Map<Integer, @Atomic Account> accounts;
3
4   void transfer(int src, int dst, float amount) throws OverdraftException {
5    @Atomic Account destAccount = accounts.get(dst);
6    @Atomic Account srcAccount = accounts.get(src);
7    srcAccount.withdraw(amount);
8    destAccount.deposit(amount);
9   }
10  float balance(int accountNumber) {
11   @Atomic Account acc = accounts.get(accountNumber);
12   return acc.getBalance();
13  }
14 }
15
16 class Account {
17   float balance;
18   void deposit(float amount) { ... }
19   void withdraw(float amount) throws OverdraftException  { ... }
20   float getBalance() { ... }
21 }
```

Listing 1: Bank transfer.

The second example is the integer linked-list implementation presented in Listing 2, of which we highlight only the implementation of method `add`, being the other two similar. To cope with concurrent invocations, the implementation of `add` (and `remove`) must ensure that the modification of a node's configuration is atomic. Once again, this requirement is simply expressed by assigning the nodes to atomic

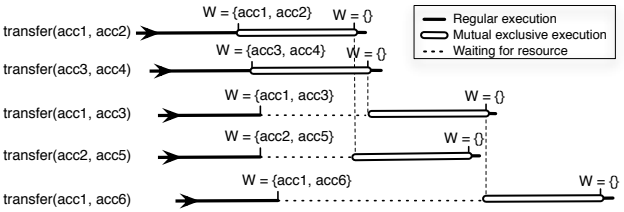

Figure 1: Six concurrent bank transfers

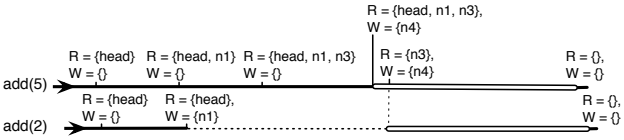

Figure 2: Two concurrent insertions in list: 1 :: 3 :: 4

variables. Both methods modify a single atomic resource, being thus mutual exclusion confined to that one operation. Figure 2 illustrates the execution of two concurrent executions of `add` upon a list containing elements: 1 :: 3 :: 4. Although operating upon distinct nodes, in the presented schedule, execution `add(2)` may only alter the node storing value 1, denoted by `n1`, after execution `add(5)` has released that same resource and its predecessor (`head`). This release operation can only be performed from the moment execution `add(2)` has acquired all the resources it needs and will no longer access `n1` and `head`. From then on, both executions may execute in parallel with mutual exclusive access to the resources they require to apply the programmed behaviour.

```
1   class LinkedList {
2    @Atomic Node head;
3
4    boolean add(int value) {
5     int v;
6     @Atomic Node prev = head;
7     @Atomic Node next = head.getNext();
8     while ((next != null) || (v = next.getValue()) < value)) {
9      prev = next;
10     next = prev.getNext();
11    }
12    boolean result = (v != value);
13    @Atomic Node newNode = new Node(value,next);
14    if (result) prev.setNext(newNode);
15    return result;
16   }
17   ...
18  }
```

Listing 2: Integer linked list.

3.2 Properties

The model guarantees safety properties, such as strong atomicity, absence of data races, and serialisability, and liveness properties, such as progress (deadlock absence). We are currently formalising and proving the properties. All these are ensured statically but at different stages of the compilation process. More particularly, strong atomicity and the absence of data races are ensured by the type system, whist serialisability and deadlock absence is ensured by the encoding of the annotated source code into pure Java (or Java bytecode) + locks. In other words, the generated code features a set of operations over locks that ensure that the concurrent schedules of method executions produce the same results as their serial execution, and guarantee that resources are never acquired in such an order that may drive to deadlocked states.

3.3 Type System

We want the `@Atomic` annotation of a variable to have an impact on the latter's type and, consequently, on the type of the values that are assigned to it. The goal is to ensure that well-typed programs do not feature data-races that fall out of the control of RC^3's runtime system. To achieve such goal, we must ensure that mutable memory objects that have been assigned to atomic variables may not be assigned to regular (non-atomic) variables, and vice-versa. To that end, we introduce the notion of *atomic type*: the atomic counterpart of a regular type, either primitive to the language or defined programmatically.

For each type declared in the program, there is an implicit atomic equivalent that abides to the following rules: 1) the class' supertype is also converted into its atomic counterpart, to ensure that atomic and non-atomic types have different roots, and 2) the methods that return instances of the class itself have their return type C replaced by the class' atomic counterpart, denoted by $C^{@}$. This approach implicitly provides an atomic duplicate of the type of the original class, removing this burden from the programmer.

Strong Atomicity: The clear separation between regular and atomic types provides the framework for the centrality of concurrency control in RC^3. The omission of an annotation on what should be an atomic variable results in a typing error. Consider the wrongful implementation of the example depicted in Listing 3. The missing `@Atomic` annotation at lines 6 and 7 results in a compile-time error, because the type of the left and right-side values of the assignments in either of those statements have different types. For instance, at line 6, the type of head is *Atomic Node*, while the type pref is simply *Node*. The placement of the `@Atomic` annotation in line 6 will trigger a compilation error at line 10. The program will only compile when all values deemed atomic are only assigned to atomic variables and values deemed not-atomic are only assigned to *regular* variables.

```
1   class LinkedList {
2    @Atomic Node head;
3
4    boolean add(int value) {
5     int v;
6     Node prev = head;  // Compilation error
7     Node next = head.getNext(); // Compilation error
8     while ((next != null) || (v = next.getValue()) < value)) { ... }
9     boolean result = (v != value);
10    Node newNode = new Node(value, next);
11    ...
12   }
13  }
```

Listing 3: Integer linked list with compilation errors.

```
1   class Bank{
2    Map<Integer, Account> accounts;
3
4    void transfer(int src, int dst, float amount) throws OverdraftException {
5     Account destAccount = accounts.get(dst);
6     Account srcAccount = accounts.get(src);
7     srcAccount.withdraw(amount);
8     destAccount.deposit(amount);  // Warning - High-Level Data Race
9    }
10  }
11
12  class Account {
13    @Atomic float balance;
14    void deposit(float amount) { ... }
15    void withdraw(float amount) throws OverdraftException  { ... }
16  }
```

Listing 4: Bank transfer with possible HLDR.

Absence of Data Races: The semantics of the model ensures the absence of data race between two concurrent method executions. However, High-Level Data Races (HLDR) may occur. Our approach is to detect them statically via the type system, which will detect that a method execution is accessing two resources in two distinct atomic scopes. Given

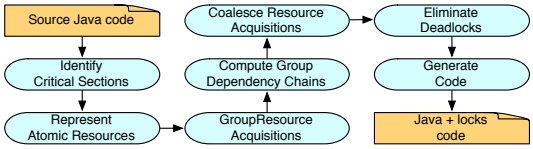

Figure 3: Main stages of the encoding process

that such kind of scenario may lead to a high-level data race, the system will automatically generate an atomic scope that encompasses all accesses and inform the programmer of the decision. The approach is conservative, in the sense that it compromises performance in favour of correctness.

Listing 4 illustrates an version of the Bank example that leads to an HLDR. The `@Atomic` annotation is placed only in the balance field of class `Account`, and thus the balances of the accounts are evaluated in distinct atomic scopes.

4. ENCODING INTO JAVA

The proposed model is a high-level abstraction that must be encoded into lower-level concurrency control primitives. This encoding must preserve the semantics of the original model, and ensure additional properties, such as progress and serializabilty. We favour a pessimistic concurrency control approach and hence will resort to a simple lock-based system. The data-centric nature of RC^3 already imprints, in the code, which data items must be subjected to concurrency constraints in the execution of a method. We use this information as a starting point, associating one lock to each atomic resource, and forcing the acquisition of this lock before the first access to the associated resource in a method execution. There is however a considerable semantic gap between our data-centric model and these low-level lock-based primitives, as the latter: a) are code-centric and hence require the delimitation of the code regions that have mutual exclusion requirements – the *critical sections* – and; b) are prone to deadlock situations, which compromises progress. Figure 3 depicts the main steps of the encoding process, which we briefly describe below.

Identify Critical Sections: This step is quite straightforward, as there is a one-to-one correspondence between critical sections and methods that operate upon atomic variables. Section boundaries are narrowed down to the list of statements that mediate between the first and the last access to an atomic variable in the correspondent method's body. Multiple exit points are allowed, from both return statements and the raising of exceptions. All methods detailed in Listings 1 and 2 embed an atomic section, as they all operate upon atomic variables.

Static Representation of Atomic Resources: Being limited by the reach of static analysis, we have to approximate the set of resources that will be accessed during the execution of a program. To that end, we extract information from the source code, namely the declaration and manipulation of atomic variables. Currently, the runtime resources that cannot be unequivocally identified at compile-time are approximated by their type. Additionally, to adequately implement the semantics of the model, resource accesses must be differentiated between read and write accesses. For example, variable `prev` in the linked-list example will hold a resource of type `Node` that is accessed in read-write mode.

Group Atomic Resource Acquisitions: In a computation, if all locks detained by all threads at any given instant have been acquired according to a pre-established total order, then, that same computation is deadlock-free. This strategy is, however, only applicable to the cases where the set of locks to grab in a critical region is known in advance, before entering the region - what we call a *static* scenario. Otherwise the ordering premise cannot be guaranteed. Despite this limitation, we want to leverage this property and, whenever possible, atomically acquire multiple resources at once. Thus, we group all resource acquisitions that do not have data dependencies between them. For that purpose we introduce the notion of *resource group* - a set of resources that can be atomically and synchronously acquired, following an all-or-nothing semantics that respects a predefined total ordering relation. Looking back to the `transfer` operation of Listing 1, both accounts are known beforehand and can thus be seized at the entry of the atomic section (line 7) and released when they are no longer needed (after line 8).

Compute Resource Group Dependency Chains: To be able to detect deadlocks, we compute the control dependencies between all resource acquisitions performed within the scope of a given atomic section, including the ones carried out in the scope of nested method calls.

Coalesce Resource Acquisitions: This step eliminates all redundant resource acquisitions by coalescing them into others. It impacts on the efficiency of the deadlock detection algorithm because it reduces the number of dependencies to be considered, and it improves the application's execution time by reducing the number of lock/unlock operations.

Deadlock Elimination: Deadlocks in our approach arise from the fact that the all-or-nothing semantics of singular group acquisition is not preserved across distinct acquisitions. Works such as [6, 7, 10] prevent deadlocks by generating course-grain locks to serialize the execution of deadlock-prone code. Our approach follows a similar philosophy. We partition the resource group space in such a way that interdependent groups, that may lead to deadlocked computations, are placed in the same partition. Then, we force the acquisition of such groups to first obtain exclusive access to the partition they are associated to. This automatically breaks the circular-wait deadlock precondition, given that interdependent group acquisitions are protected by coarser-grain locks that can be grabbed in a deadlock-free manner.

Code Generation: The encoding process produces Java code with calls to a library that supports the acquisition of resource groups. The library offers operations for the creation of a chain of group acquisitions, and for the actual acquisition and release of groups. With such API the encoding is rather simple: for each atomic section we create a chain of group acquisitions and process it sequentially, from its beginning, generating a resource group acquisition before the first access to an element of each group. Serializability of the concurrent schedules of method executions is ensured by applying a two-phase locking approach to resource acquisition. Hence, there are two non-overlapping phases for resource group acquisition and release. To boost concurrency, in the latter phase, resources may be released individually.

	DeuceSTM	RC^3		
	Total	Primary	Derived	Total
TSP	**15**	9	0	**9**
Elevator	**9**	4	0	**4**
Bank	**3**	1	5	**6**
IntHashSet	**3**	5	0	**5**
LinkedList	**3**	1	13	**14**
RBTree	**3**	3	15	**18**

Table 1: Number of annotations in DeuceSTM and RC^3.

	AJ						RC^3
	atomicset	atomic	owned	unitfor	alias (/*this.L=L*/)	Total	@Atomic
TSP	2	9	0	0	0	**11**	**9**
Elevator	1	4	0	8	8	**21**	**4**
Weblech	2	4	0	0	0	**6**	**4**
Jcursez1	5	15	0	16	29	**65**	**25**
Cewolf	4	5	0	0	1	**10**	**5**
Collections	5	53	0	40	330	**428**	**146**

Table 2: Number of annotations required in AJ and RC^3.

5. EVALUATION

In this section we evaluate the RC^{3}model and our prototype implementation. The model is compared against both AJ and atomic blocks according to their productivity and their properties. In turn, the prototype is compared from a performance perspective against a known Java Software Transactional Memory system, DeuceSTM, and Java synchronized blocks. This evaluation will give us an assessment of RC^3's competitiveness against leading technologies.

5.1 Productivity *vs* control-centric approaches

The advantages of data-centric concurrency control over its code-centric counterpart have been widely discussed in [4]. Nonetheless, we highlight two properties of RC^3: the centrality of the concurrency management and the strong atomicity. In RC^{3}the programmer only has to annotate the variables that should be atomically evaluated throughout the program. Any concurrency anomaly is bound to be linked to the data declaration. Moreover, in RC^3, the annotation of a single variable with `@Atomic` is sufficient to disseminate its concurrency requirements throughout the program. The compiler will refuse to compile until all these are fulfilled.

Table 1 details the number of constructs required by RC^3 and atomic sections for 6 case-studies taken from the AJ and DeuceSTM documentation. As can be observed, RC^3 requires less annotations in the TSP and Elevator case-studies, but requires more in the remaining. This difference is, in great part, due to the scale of these benchmarks. The evaluated data-structures comprise only three atomic operations, regardless of their internal complexity. In RC^3, these structures require a greater number of annotations, in great part, due to the passing of resources by argument and the use of local atomic variables that have to be annotated. However, lets consider two distinct types of annotations. The *primary* annotations are the ones that, when applied, will force the programmer to apply the remaining annotations in order for the program to compile. The remaining *derived* annotations can be easily inferred from compiler errors and, as such, can be delegated to an IDE. In this scenario, if we only consider the *primary* annotations, RC^3 requires a lower number of annotations. The exception here is the IntHashSet benchmark, where we need to annotate 5 class fields, despite the fact that only 3 methods use them.

5.2 Productivity *vs* Atomic Sets

The advantages of RC^3 against Atomic Sets are twofold: programming simplicity and the guaranteed absence of deadlocks. The Atomic Sets model burdens the programmer with several constructs with distinct semantic meanings. The correct and efficient use of these constructs may not be trivial and is prone to both mutual exclusion violations if the `unitfor` is not carefully used, and to deadlock situations [4].

The authors recognize that the use of `unitfor` may require some code refactoring. Conversely, in RC^3, the simple annotation of which variables hold values that must be evaluated atomically is enough to express concurrency restrictions in a safe manner. The simplicity regarding the number of constructs and respective semantic meanings provided by RC^3; one annotation, against the six used by AJ marks a clear advantage for RC^3.

AJ is more generic than RC^3 in a single aspect: the programmer may subdivide groups so that these may be partially acquired. The feature may, eventually, lead to a more fine grained management of group level concurrency. However, none of the case-studies studied by us and made available by the authors of AJ benefit from this feature.

Table 2 depicts the number of constructs required by RC^3 and AJ to express concurrency control for 6 case-studies taken from the AJ literature. As can be observed, RC^3 consistently requires a lower number of annotations - the total sums both primary and derived annotations. In fact, a source code analysis evidenced that almost all uses of our `@Atomic` annotations coincide with uses of AJ's `atomic` (perceptible in the table by comparing the columns associated to both). Therefore, our static analysis is able to subsume most of the remainder annotations. We also bring attention to the fact that AJ's `atomic` annotation may be applied to classes, as a mean to express that all of its fields are atomic. This decreases the number of annotations required in some case studies and can also be trivially implemented in RC^3.

5.3 Performance

Setup: node with four 16-core AMD Opteron™ processors and 64 GBytes of RAM. Benchmarks compiled and ran with OpenJDK 8 Build b94 x86_64.

Benchmarks: AJ's compiler is not publicly available. Nonetheless, the code generated for some examples is available from site https://www.cs.purdue.edu/sss/projects/aj/. Of those, only TSP is a benchmark. The performance results delivered by RC^3 for that one example are close to the Java synchronized block-based implementation, surpassing AJ's performance in more than 200x for 64 cores. To have a more rigorous assessment of our prototype's performance, we adapted 4 micro-benchmarks of the DeuceSTM framework to RC^3, and to a naïve implementation using Java synchronized blocks. We have then run these benchmarks in scenarios of low (10% writes), medium (25% writes), and high contention (50% writes).

Bank (Figure 4a): RC^3 yields better performance than DeuceSTM. This is in great part due to the static nature of the transactions between two accounts, where an ordered locking of both accounts is sufficient to assure atomicity. The naive approach of the synchronized block-based implementation achieved worse performance than both systems.

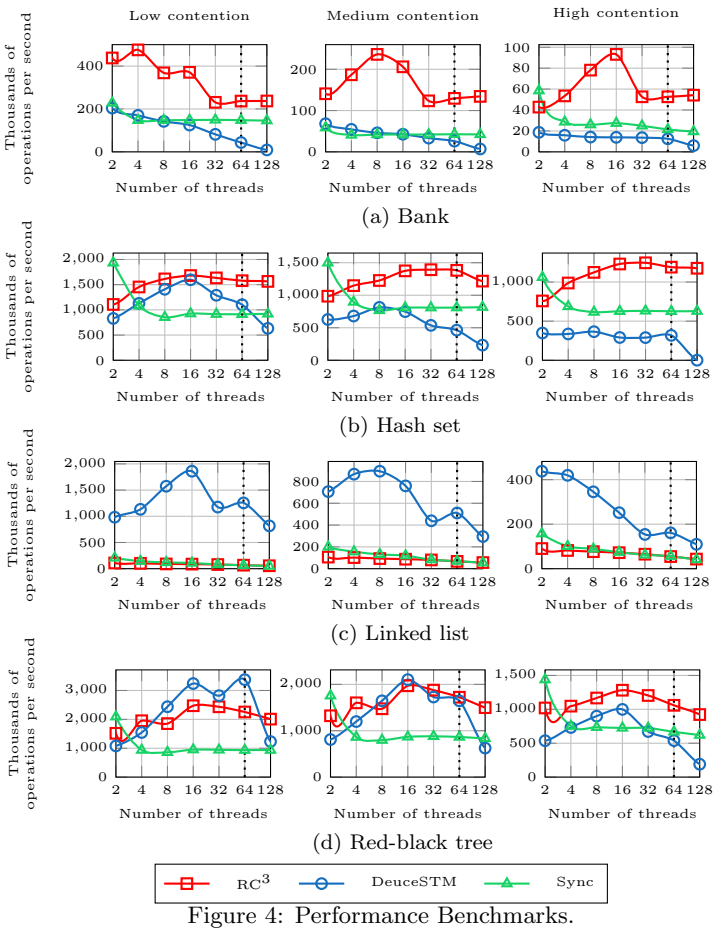

Figure 4: Performance Benchmarks.

Naturally, a more in-depth study of the code may lead to more performable solutions.

Hash set (Figure 4b): RC^{3}outperforms DeuceSTM, specially at medium and high contention. The hash table relies on a single array, and most operations require access to a single position of the array. Due to disjoint data accesses patterns, DeuceSTM thrives under low contention, as expected. The increase in the conflict ratio is exacerbated as contention increases.

Linked list (Figure 4c): RC^3, along with synchronized blocks, performed worse then DeuceSTM. This data structure is very suitable for optimistic approaches, hence the STM's high throughput. Furthermore, in RC^3, a potential deadlock scenario is identified. This a false positive[1] that results from the inability to statically differentiate the multiples nodes of the list. As a result, a coarse-grain lock is generated and all operations that alter a list's configuration are serialized against each other.

Red-black tree (Figure 4d): In this benchmark, similarly to the linked list, coarse-grain locks are generated to avoid potential deadlocks in node acquisitions. DeuceSTM provides better performance at lower contentions, due to the sparsity of data access, but looses to RC^3 when contention rises. The synchronized blocks implementation performed worse then RC^3, which, at a glance, is peculiar, since both systems basically serialize all operations. In fact, this worse performance is attributed to the forced re-acquisition of locks in the synchronized blocks, in nested method invocations. In our prototype, the lock coalescing step removes these locks, boosting performance.

[1]From the multiple case studies we implemented, this is the only one for which the static analysis outputs a false positive.

6. CONCLUSIONS AND FUTURE WORK

We have presented RC^3, a concurrency control mechanism that conjoins the advantages of data-centric concurrency management with the ones of a simple programming framework, with a simple semantics, upon which is easy to reason about. RC^3 requires less annotations that Atomic Sets in all the examples that we have implemented. Although its simplicity, RC^3 grants important safety and progress properties, while still providing a competitive performance against coarse-grained synchronised blocks and a known Java STM.

Herein we present informally the model and its intended properties. Ongoing work is addressing the formal definition of the model, of a type system, and of encoding the annotations into locks, so that the model's properties can be formally proven. Future work intends to leverage recent shape analysis techniques to improve the static approximations of the resources to be acquired at runtime, and, with that, further minimise the gap between our system and the manual, fined-tuned, application of low-level synchronisation primitives.

7. REFERENCES

[1] B. M. et al. Autolocker: Synchronization inference for atomic sections. In *POPL*, pages 346–358. ACM, 2006.

[2] D. M. et al. Detecting deadlock in programs with data-centric synchronization. In *ICSE*, pages 322–331. IEEE, 2013.

[3] F. T. S. et al. Dynamic optimization for efficient strong atomicity. In *OOPSLA'08*, pages 181–194. ACM, 2008.

[4] J. D. et al. A data-centric approach to synchronization. *TOPLAS*, 34(1):4:1–4:48, 2012.

[5] M. A. et al. Semantics of transactional memory and automatic mutual exclusion. *TOPLAS*, 33(1):2, 2011.

[6] M. E. et al. Lock allocation. In *POPL*, pages 291–296. ACM, 2007.

[7] M. H. et al. Lock inference for atomic sections. In *TRANSACT*. ACM, 2006.

[8] M. V. et al. Associating synchronization constraints with data in an object-oriented language. In *POPL*, pages 334–345. ACM, 2006.

[9] P. D. et al. Automated inference of atomic sets for safe concurrent execution. In *ASTE*, pages 1–8, 2013.

[10] S. C. et al. Inferring locks for atomic sections. In *PLDI*, pages 304–315. ACM, 2008.

[11] S. L. et al. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, pages 329–339. ACM, 2008.

[12] W. Huang and A. Milanova. Inferring aj types for concurrent libraries. In *FOOL*, 2012.

[13] N. Shavit and D. Touitou. Software transactional memory. In *APODC*, pages 204–213. ACM, 1995.