FlexSync: An aspect-oriented approach to Java synchronization[*]

Charles Zhang
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
charlesz@cse.ust.hk

Abstract

Designers of concurrent programs are faced with many choices of synchronization mechanisms, among which clear functional trade-offs exist. Making synchronization customizable is highly desirable as different deployment scenarios of the same program often place different precedences on synchronization choices. Unfortunately, such customizations cannot be accomplished in the conventional non-modular implementation of synchronization. To enable customizability, we present FlexSync, an aspect oriented synchronization library, to enable the modular reasoning of synchronization and to resolve the coupling between synchronization intentions and mechanisms in Java systems. With FlexSync, programming synchronization is largely declarative. Complex Java systems can simultaneously work with multiple synchronization mechanisms without any code changes. The FlexSync load-time weaver performs deployment time optimizations and ensures these synchronization mechanisms interact with each other and with the core system consistently. We evaluated FlexSync on commercially used complex Java systems and observed significant speedups as a result of the deployment-specific customization.

1 Introduction

In Java programs, synchronization is commonly referred to as the coordination of multiple threads in accessing shared program states. As concurrency becomes a common programming practice in the multi-core era, the designers of concurrent programs are faced with many choices of synchronization mechanisms such as the use of locks, atomic blocks [7, 8], and, more recently, software transactional memory [10, 21]. For their distinctive operational differences, clear functional trade-offs exist among these synchronization mechanisms. This is problematic for building general-purpose and reusable Java systems as, in conventional approaches, synchronization mechanisms are "hard-wired" to the application logic through the use of library APIs or specialized language constructs. At the same time, choosing the most appropriate mechanism is increasingly a decision made through reasoning about how reusable systems are being integrated in diversified composition contexts. Let us further elucidate this issue through a simple example.

Our example looks at a general-purpose data structure, `Buffer`, shared by multiple threads in a concurrent program. Each thread makes accessor calls to store and to retrieve data from the buffer. We have a consistency rule such that these accessor calls (`get` or `set`) can only proceed if the state of the `Buffer` is valid (`full` or `empty`). Inconsistency can happen if, for example, thread A empties the buffer after thread B verifies that the buffer has data and before it retrieves the data. There are three popular synchronization options: the use of the Java `synchronized` keyword (`lock`), the block-level atomicity support (BA) using two-phase-locking (2PL) as in [1], and the use of software transactional memory library(`stm`) exemplified by dstm2 [10] (Please refer to Section 2 for more introduction on 2PL-based BA and dstm2). In Figure 1[1], we plot the time each version takes to complete a fixed number of work units as the number of concurrent threads increases. Each work unit consists of a fixed number of `set/get` combinations. For the lock version, we count successful operations and perform a retry if any inconsistency is detected. The BA and STM versions produce no inconsistencies. One can easily observe that the lock-based approach has the fastest response time, whereas the BA implementation is slightly slower. The performance of the dstm2 version experiences significant fluctuations. It can be as fast as the BA approach or 5-6 times slower.

The measurements show that the choice of synchronization mechanism for `Buffer` is really dependent on what

[*]Draft. In ICSE 2009, Vancouver, Canada

[1]All measurements are collected on a dual-core Linux workstation with 4GB of physical memory. The number of total threads ranges from 10 to 1000. Each point is taken as the shortest time of five runs.

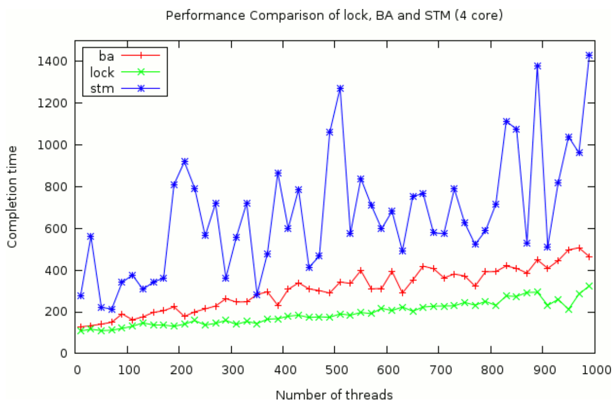

Figure 1. A comparison of response time among locks, BA and STM

matters the most to the domain of its application. For instance, the use of locks is preferred if high performance is to be pursued and inconsistencies can be tolerated. The STM approaches are appealing to the ones requiring transactional semantics on the shared states and not sensitive to the fluctuations of the processing time. If atomicity is the only required property, applications would prefer the lock-based atomicity support, which acquires more locks to achieve the consistency of states and, as the result, sacrifices a certain degree of concurrency. Therefore, the buffer code, if it were to be used as a general-purpose building lock of other concurrent application, cannot be hardwired with any particular synchronization mechanism afore-listed. The practice of client-side locking, such as the `synchronizedMap` method of the `Collection` class, is effective in treating this problem for types representing data structures. However, we only use buffer as a problem illustration. Our work considers complex reusable Java programs, many are concurrent themselves.

Our buffer example poses a paradoxical design challenge: synchronization must be designed and implemented before the program can be used; however, the best synchronization mechanism is not known until we know about how the program is used, i.e. its composition context. This design paradox is conventionally resolved if, first, the program feature is modular, and, second, its interactions with the rest of the program can be abstracted for the use of late binding techniques. None of the premises holds in the conventional treatments of synchronization, the same reason why it is considered a classic crosscutting concern [12]. The majority of the proposed solutions, including both library-based and language-based approaches [10, 22, 8, 16], require system-wide code-level commitment to particular synchronization mechanisms. The induced inflexibility is detrimental to reusable Java systems. Our case studies of real commercial middleware systems show that the performance overhead caused by this structural rigidity can be as much as 40%. This problem will exacerbate drastically because the degree of reuse and integration will increase dramatically [17].

To tackle these challenges, we present FlexSync, including both an aspect-oriented library and a load-time weaver, to enable the modular reasoning of synchronization and the code-level separation of its mechanisms from reusable Java programs. This separation is possible based on the observation that, conventionally, the marking of synchronization intentions, declaring regions of the program logic that require special synchronization attention, is an implicit outcome of the direct use of specific synchronization mechanisms, i.e., library APIs or language keywords. If these intentions have explicit and well-defined code structures, they can be reasoned and manipulated by metaprogramming such as AOP [12] as to *externalize* reusable *feature interactions* between synchronization and the core system. The design of FlexSync library APIs emphasizes on the ease of "picking out" the intentions where the interaction logic, encapsulated in the library, can be automatically applied. The FlexSync aspect weaver, an extension to the AspectJ aspect weaver, use static analysis, such as control-flow analysis and escape analysis, to automatically reason about the global composition and interaction of synchronization mechanisms. With FlexSync, the synchronization code is modular and lives separately from the operational code. We show that, through FlexSync, sophisticated Java systems can simultaneously work with multiple synchronization mechanisms of very different genres. The flexibility and the deployment time optimizations, made possible by using FlexSync, can significantly improve the performance for large complex systems.

We make the following contributions in this paper:

1. We first present the concept of the separation of intention and mechanism in the context of synchronization design. We empirically show that such separation can be achieved for large-scale and complex Java systems.

2. We present the FlexSync aspect synchronization library, which encapsulates patterns of interactions between Java code and the synchronization mechanisms and expose these patterns through the process of "tagging". We explain how the tagging process can attach different synchronization mechanisms onto the same code structure.

3. We present the FlexSync load-time synchronization weaver which supports the global reasoning of synchronization mechanisms in the scenarios of unanticipated composition of reusable systems.

4. We present a thorough evaluation of FlexSync-based synchronization implementations, covering the programming effort and its functional characteristics. We contribute[2] the source of the FlexSync library and the systems

[2]The FlexSync Project. URL:`http://www.cse.ust.hk/`

we experiment with for the interested readers to inspect and to perform further evaluations.

The rest of the paper is organized as follows: Section 2 introduces atomicity and the dstm2 implementation of software transactional memory; Section 3 presents the design methodology embodied in FlexSync. Section 4 evaluates FlexSync through standard benchmarks and case studies.

2 Background

Block-level atomicity
In the presence of multiple threads, the block-level atomicity means a group of program executions, scoped lexically within a block, is to be carried out serially without the interference from the inter-leavings of threads. In Java systems where objects are dynamically allocated, we use a two-phase-lock mechanism to acquire the associated locks of all dynamic objects in the control flow of the atomic block. These locks are released after the atomic block exists. Our implementation is based on the `cflow` constructs of AspectJ. Please refer to our source release for the details of the implementation.

Software transactional memory
Also referred to as the optimistic synchronization, software transactional memory (STM) provides runtime infrastructures to keep track of the reads and writes to the shared program states. It thrives on the optimistic assumptions that real data races occur infrequently for many concurrent programs. In STM, a *collision* can happen when thread B performs writes on the shared data after they are read by thread A. Then, the shared states are to be *rolled back* and the operation is re-executed. This can cause the large performance fluctuations because the chance of collision is subjective to the number of threads and the thread scheduling that can be non-deterministic. The particular STM library used in this research, dstm2, makes copies of the shared states to support rollbacks. This technique is reported to have the fastest runtime performance [10]. STM offers programmers a simpler concurrency control mechanism compared to the direct use of locks. In our application of dstm2, we replaced the automatic state copy capability in the original implementation with a callback method requiring the manual implementations. This is because many reads or writes in the systems that we have experimented with are not performed by "setters" and "getters" as required by the original scheme.

~`charlesz/flexsync`

3 FlexSync: the modular and the global reasoning of synchronization

Following the definitions in [13], the general design goal of Flexsync is to first enable the modular reasoning of synchronization through using the FlexSync-API to explicitly express how synchronization mechanisms interact with the operational logic. At the same time, we address the global reasoning of the unanticipated program compositions using the Flexsync loadtime weave. The rest of the section first present how we achieve these design goals in detail.

3.1 The separation of intention and mechanism

Synchronization is typically coded as lexical scopes over a group of programming statements, demarcated by either language keywords, such as `synchronized` or `atomic`, or by library calls, such as the `lock/unlock` pairs of some `Lock` object. These lexical *scopes* represent the synchronization *intention*s of the developers, identifying code regions requiring special synchronization treatments. We refer to language keywords or API calls used in the demarcations as the *mechanism*s, concerning the specific decisions of what kind of treatments to apply. These two concepts are usually undistinguished in conventional approaches. We advocate their explicit separations as a fundamental step towards the modular reasoning of synchronization.

In our current design, the separation is achieved by assuming that synchronization regions are method-like: the data flow in these regions can be re-expressed following the input/output model of a function. This is certainly true for methods prefixed with the `synchronized` keyword. For the synchronization blocks inside method bodies, we performed a study of whether these blocks can be automatically transformed into methods using the *Extract method* facility of the Eclipse JDT refactoring library. We use an AST walker to retrieve a synchronization block and ask the JDT API to return the refactoring status. We studied four open source programs covering four types of servers in which concurrency is extensively used: RPC middleware (ORBacus[3]), JMS broker (OpenJMS[4]), web server (Jigsaw[5]), and database server (Derby[6]). In our study of these servers, we encountered three common causes of automatic refactoring failures: *early return* (ER), where a `return` statement is nested inside the block, *multiple variable assignment* (MV), where multiple local variables are written, and *branch selection* (BR), where the block resides in a

[3]ORBacus: URL:`http://www.iona.com/orbacus`
[4]OpenJMS. URL:`http://openjms.sourceforge.net`
[5]Jigsaw. URL:`http://www.w3.org/Jigsaw`
[6]Derby URL:`http://db.apache.org/derby/`

branching block of either `if` or `switch`. Among these failures, the ER case can be generically treated with by setting a condition variable to true inside the refactored method and checking this condition after the method returns. It requires trivial source rewriting and, thus, can be automatically treated.

In Table 1, we report the sizes of each program, the usage of synchronization, as well as the results of invoking the Eclipse refactoring APIs. Our observation is that ER accounts for the majority of the refactoring failures, and over 97% of synchronization blocks in all of the four programs can be automatically re-expressed using functions. The non-automatic blocks require manual inspections and code restructuring. They are small in number and, from our experience, only require more sophisticated refactoring steps. Our study validates our design assumption that synchronization intentions can be characterized by the method boundaries.

Program	Derby	Jigsaw	OpenJMS	ORBacus
Size	915320	160740	112968	189852
Synchronization usages				
Method	294	637	204	466
Block	604	149	400	262
Total	898	786	604	728
Method Extraction Failures				
ER	90	11	30	29
MV	18	2	9	3
BR	8	0	6	2
Degree	87.1%	99.7%	92.5%	95.3%
Degree with ER	97.1%	98.3%	97.5%	99.3%

Table 1. Summary of Server Synchronization Usage

3.2 The FlexSync synchronization library

Synchronization perspectives

The design of the FlexSync APIs is based on our observation that different synchronization mechanisms can be treated as different interpretations of the same set of call graph elements. Suppose that Figure 2(A) represents the call graph of a fictitious Java program. For the lock-based approaches, including the atomicity support, we need to identify class types encapsulating shared program states as well as the method interfaces that could lead to data races. This interpretation is illustrated in Figure 2(B). For STM, the interpretation is totally different as, by its optimistic nature, it is not concerned with the state sharing and data races. Instead, we need to first identify the class types encapsulating transactional executions and, second, the methods that cause reads or writes to the shared states. The corresponding representation of the original call graph as depicted in Figure 2(C).

Synchronization specification

These different interpretations of the calling relationships are supported in FlexSync through a design process which we characterize as a "tagging" process. There are two types of conceptual tags: the *role tag* operates on the class level for the declaration of the required synchronization facilities; the *action tag* operates on method level for the provisioning of these facilities. We use the callgraph to explain the tagging process. However, the use of FlexSync does not require the knowledge of callgraphs. Role tags and action tags require only local reasoning about a particular type. The tags for specific synchronization mechanisms are as follows:

Lock The tag `AutoLckTarget` identifies class types the methods of which are always synchronized, i.e., having `synchronized` on the method definitions, if Java monitor is to be used. The tag `LckTarget` identifies types that are caller-synchronized. The methods cause data races are tagged using `Guarded`. For the call graph given in Figure 2(A), class type `D` is tagged as an `AutoLckTarget` as calls to its methods are unconditionally synchronized. `E,` `F` and `G`, are LckTargets, as they are selectively synchronized in the caller's lexical context. The tagged version of the call graph is presented in Figure 3(A).

Block atomicity `BAOwner` identifies the class types in which the executions of one or more of its methods are atomic. We identify these methods with the `Atomic-Execution` tag. Classes having shared states, in this case, are identified with the `BATarget` tag. We use the `AGM` (atomic group member) tag on methods defined in `BATargets` if, first, they cause reads or writes to shared states, second, they are in the control flow of `Atomic-Executions`. In our example (Figure 3(B)), class D has an atomic method d_1. The control flow of d_1 includes calls to methods e_1, f_2 and g_1. The control flow information is maintained behind the scenes by the FlexSync runtime. Therefore, these methods are tagged with `AGM` and the corresponding class types `E,` `F` and `G` with `AtomTarget`.

STM execution `Transactional` identifies class types the states of which require transactional support. The accessor methods are identified with `AccessorCall`[7]. `TXExecution` identifies methods to be executed transactionally. In our example (Figure 3(C)), `AccessorCall` identifies accessor methods c_1, e_1 and f_2. We thus identify types C, E, F as `Transactional` and the transactional method d_1 as `TXExecution`.

[7]The actual FlexSync APIs distinguish between *read* tags and *write* tags. We use a general name, `AccessorCall`, for the conciseness of the presentation.

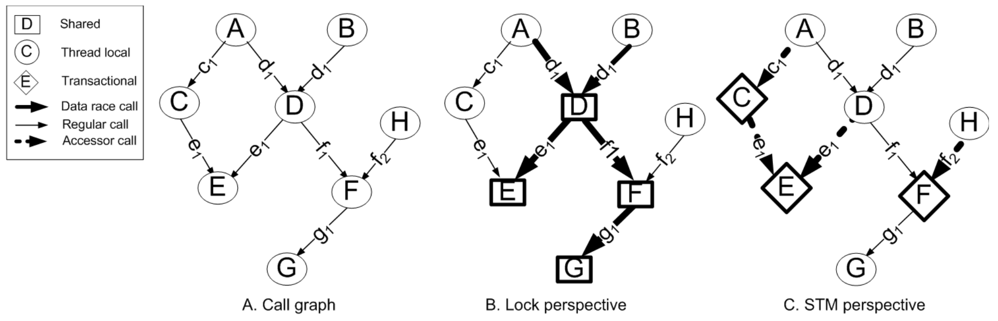

Figure 2. Synchronization perspectives of the call graph

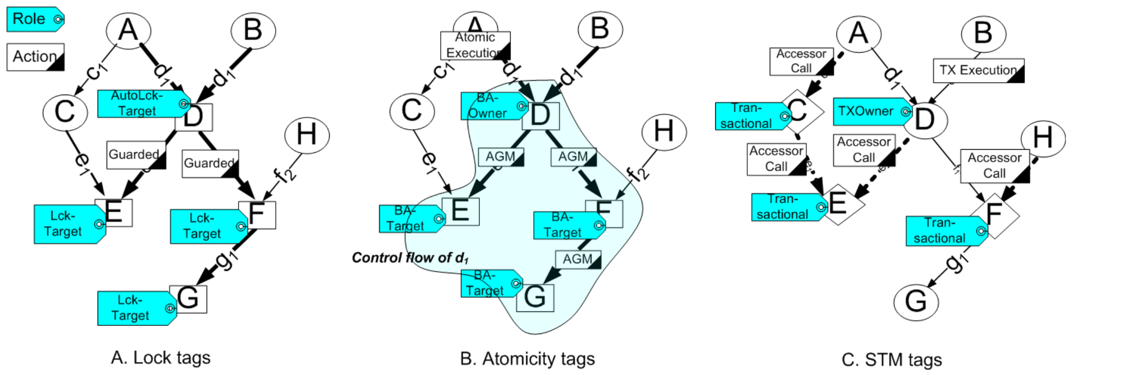

Figure 3. Call graph tagging with flex

Implementation of tags

In FlexSync, we use Java interfaces and the AspectJ abstract `pointcuts` to support the concept of tagging. The tagging process can be translated into the programmatic mappings of these interfaces and pointcuts in aspect modules to the corresponding elements of Java programs. In the AspectJ nomenclature, the mapping of role tags is accomplished declaratively through inter-type declarations(ITD) and the actions tags through "concretizing" abstract `pointcuts`. Both techniques are commonly used in aspect library implementations [11, 9, 23]. In the following, we present the reusable logic implemented in FlexSync for each mechanism, paraphrased using the tag vocabulary:

Lock 1. The executions of the `Guarded` methods of the `AutoLckTargets` are protected by the monitor of the corresponding `AutoLckTarget` instances. 2. The call sites of the `Guarded` methods of the `LckOwner` are protected by the monitor of the callee instances of `LckOwner`.

Atomicity: 1. If a `AGM` is within the control flow of the `AtomicExecution` of the `BAOwner`, acquire the monitor lock of its corresponding `BATarget` and register the lock with the `BAOwner`. 2. When a `AtomicExecution` completes, release the monitor locks of all `BATargets` registered with the `BAOwner`.

STM: 1. As `Transactional` instances initialize, set up their per-instance duplicates, as required by dstm2, to allow state rollbacks. 2. At the call sites of "accessors"[8], signal the dstm2 runtime to verify if the on-going transaction can proceed or must be aborted. 3. Repeat the `TX execution` of the `TXOwner` until the transaction successfully commits.

We emphasize the fact that the interaction logic can be well hidden behind our "tag" abstractions and implemented through AspectJ. This is a salient property of modular reasoning as pointed out in [13]. We will not bore the readers here with the implementation details and encourage the interested readers to download[9] a copy of the library for further references.

We now come back to the *Buffer* example presented in Section 1. The `Buffer` class contains four methods: `setData, getData, isFull`, and `isEmpty`. These methods are invoked by the `doWork` method of the class `BufferUser`. Remind that, before invoking the acces-

[8]Quotation here to entail that, in practice, not all methods that read or write the state of the object would lexically start with "set" or "get".

[9]FlexSync. URL: `http://www.cse.ust.hk/~charlesz/sync`

sor methods, the method `isFull` or `isEmpty` is called to check the state of the buffer. The state is validated again inside the accessor methods. Figure 4 presents the AspectJ implementation of the three supported synchronization mechanisms through the FlexSync APIs. The use of the FlexSync tags are underlined with bold italic letters. Despite its simplicity, we want to demonstrate the high degree of declarativeness in the implementation of synchronization enabled by FlexSync. As our evaluation in Section 4 shows, this property still holds for complex Java server systems. The implementation of the `copyTo` method is, however, non-declarative and enforced by the AspectJ compiler in the case of the STM support.

Limitations

To use FlexSync, refactoring is still needed to transform blocks into methods. We are currently working on an automated solution to make the process transparent. The `wait/notify` semantics are also to be treated case by case, as they often intertwine with the application logic. We provide a replacement of `wait` by releasing the object lock in the case of BA and using an "abort→re-execution" sequence in the case of STM. The use of `wait`, however, will break the atomicity guarantees of BA in general as its Java semantic mandates the release of the monitor lock. In addition, the FlexSync-adaption of the dstm2 library requires FlexSync users to manually specify how program states are duplicated. From our experience, this manual process can be tedious. Our on-going work is trying to provide simplification solutions. Finally, our lock implementation does not handle the use of library-based locks such as the `ReentrantLock` in the Java 5 library, that do not necessarily conform to the same lock/unlock interface, hence, require new library code to be created. However, the dominating majority of lock uses in the Java programs that we have studied do not use library locks.

3.3 Global reasoning of tags

The design of synchronization using FlexSync allows a program to work with multiple synchronization mechanisms through configuration. However, when we integrate these programs to build complex systems, we must ensure the consistent and optimized interactions of locally specified synchronization mechanisms from the global perspective. In FlexSync, the global reasoning is carried out at the start-up time of Java programs by the FlexSync aspect weaver. Before the first class is loaded for execution, the weaver, as an extended AspectJ bytecode weaver, carries out static analysis over the bytecode of the entire system to check for inconsistencies and optimization opportunities. It also carries out load-time weaving to specifically treat reflective loading, a common way of Java composition. We now present these capabilities in detail.

```
public aspect LckBuffer extends Lock {
    declare parents: Buffer implements AutoLockOwner;
}
```

A. Thread-safe

```
public aspect ATBuffer extends Atomicity {
    declare parents: Buffer implements AtomTarget;

    public pointcut agm():
        call(* Buffer.setData(..))||
        call(* Buffer.getData(..))||
        call(* Buffer.isFull(..))||
        call(* Buffer.isEmpty(..));

    public pointcut atomicexecution():
        execution(* BufferUser.doWork());
}
```

B. Atomic

```
public aspect STMBuffer extends ASTM {
    declare parents: Buffer implements Transactional;
    declare parents: BufferUser implements TXOwner;

    public pointcut txexecution():
        execution(* BufferUser.doWork());

    public pointcut reads():
        execution(* Buffer.getData(..))||
        execution(* Buffer.is*(..));

    public pointcut writes():
        execution(* Buffer.setData(..));

    public void Buffer.copyTo(Buffer copy){
        //copy the states. Code omitted
    }
}
```

C. STM

Figure 4. Implementing synchronization with FlexSync

Lock optimization

A synchronization mechanism is *redundant* if a type uses lock-based tags is never shared among threads in a particular compositional scenario. Such scenarios are often difficult to anticipate from the perspectives of individual programs. The FlexSync load-time weaver first leverage the techniques of escape analysis [19, 4] to detect, on the per-composition basis, the sharing status for every type associated with tags. Conventional escape analysis techniques reason about object instances. Our approach is more conservative as we define that a type escapes if any of its instances escape. Our implementation is based on the Indus project[10], which is an enhanced version of the equivalence-class-based analysis [19, 18]. The escape analysis goes through all classes seen on the Java class path, and the results are stored in a hash table maintained by the FlexSync weaver to decided whether the weaving is necessary, if a lock-based synchronization mechanism is to be used.

We claim no significant extensions to the Indus escape analysis algorithm other than the treatment of reflective class loading. The reflective class loading is referred to as instantiating a class by its lexical name through the `Class.forName` Java API combined with a type cast. In this case, we simply look up all subtypes of the type used in the type cast and store the results in the mapping table of the weaver. The weaver will have the concrete information after the actual subtype is loaded.

Consistency checking

Two synchronization mechanisms can cause *inconsistency* if, for instance, a type tagged with `BATarget` is not in the control flow of any atomic blocks, which causes incorrect holding of locks, or, a `TXTarget` is also tagged separately with `LckTarget`, which violates the lock-free assumption of STM. To prohibit erroneous usage of tags, FlexSync uses consistency rules presented in Table 2 as set predicates[11]. These rules enforces the following usages of tags: 1. each synchronized type must declare a default synchronization mechanism (from rule 1); 2. tags `TXTarget` and `BATarget` must not used without the matching "owner" tags (from rule 2); 3. $owners$ or $targets$ tags cannot be repeatedly used on the same type (from rule 3 and 4); 4. the order in the following tag pairs, (`TXOwner, TXTarget`) and (`BAOwner, BATarget`), must be maintained along the program control flow with no other "owner" tags used in between (from rule 5 and 6); 5. lock-based mechanisms cannot be in the control flow of STM-based ones (rule 7).

To perform consistency checking, we build a simple control-flow graph consisting of only the tagged types from the analysis information collected by the escape analysis of the Indus framework. Since the graph is generally small (in the order of hundreds), a simple depth-first graph traversal can accomplish the checking of the rules very quickly.

4 Evaluation

The assessment of FlexSync consists of two studies, one related to the programming effort of using FlexSync in modularizing synchronization, and the other to the functional characteristics of FlexSync. The target systems of study is specjbb2005[12], a popular benchmark for transactional enterprise Java applications, OpenJMS[13], an open source implementation of the JMS 1.1 specification, and ORBacus[14], an open source commercial implementation of the CORBA 2.4 specification. To experiment with software compositions, we switched the RPC engine of OpenJMS from Java RMI to ORBacus. This is a fully functional replacement as verified by the Sonic JMS benchmark[15].

4.1 Programming with FlexSync

In this evaluation, we want to first find out if FlexSync is capable of supporting synchronization in commercially used complex systems. We also want to study the programming characteristics of implementing synchronization in FlexSync APIs. We first remove the monitor-based synchronization from the original implementations. We then perform necessary refactorings to enclose synchronization blocks within method definitions. We use FlexSync to support the locking, block-level atomicity, and the dstm2-based STM. We cannot use STM on ORBacus because ORBacus involves network I/O operations which cannot have *rollback* semantics. This is a typical limitation of STM in general. The OpenJMS server poses the same limitations, however, since we use ORBacus as its remote procedure call (RPC) engine, the non-RPC part of the OpenJMS can have STM-compatible behaviors. This is part of our composition case study which will be presented shortly.

The total size of the FlexSync library is less than 60KB of Java bytecode. The sizes of specjbb2005, ORBacus and OpenJMS are listed tn Table 1. In Table 3, we quantify four aspects of the FlexSync-based implementation for each of the studied system: number of modules where synchronization is implemented in the original application (*orig*), the number of modules for the FlexSync-based implementation (*flex*), the total size, in lines of code (LOC), of declarative(*dec*) and non-declarative(*ND*) portion in the FlexSync user code, and the non-declarative portion of STM implementation(ND'). We define the declarative degree, α, as the ratio between the declarative code and the total size of synchronization implementation. α' is computed without the STM code. Here we treat the STM as a special case because its non-declarative code almost exclusively involves the state duplications, i.e., copying class variables. We are working on automating this process.

[10]Indus project. URL:`http://indus.projects.cis.ksu.edu/`

[11]Note that the control flow definition($cflow$) is the conservative control flow computed statically from the bytecode where all possible branches of the call flow are explored

[12]Specjbb. URL: `http://www.spec.org/jbb2005/`

[13]OpenJMS URL: `http://openjms.sourceforge.net`

[14]http://www.iona.com/orbacus

[15]Sonic JMS Benchmark. URL:`http://www.sonicsoftware.com/products/sonicmq/performance_benchmarking/index.ssp`

Definitions	*Consistency rules*
1. τ : the type variable. T: the set of all types.	1. $pref(\tau) := \emptyset$
2. $tag(\tau)$: the set of tags on τ	2. $cflow(\tau) \cap owners := \emptyset, pref(\tau) \in \{TXTarget, BATarget\}$
3. $pref(\tau)$: tag indicated by design as the preferred tag of τ	3. $\|tag(\tau)\| >= 2, tag(\tau) \subseteq owners$
4. $cflow(\tau)$: the set of tags in control flows above τ	4. $\|tag(\tau)\| >= 2, tag(\tau) \subseteq targets$
5. $cover(\tau_i) := \{tag(\tau_k)\|\tau_k \neq \tau_i, cflow(\tau_k) \cup tag(\tau_k) \equiv cflow(\tau_i)\}$	5. $cover(\tau) := \{BAOwner\}, pref(\tau) \cap \{BATarget\} = \emptyset$
	6. $cover(\tau) := \{TXOwner\}, pref(\tau) \cap \{TXTarget\} = \emptyset$
7.$targets := \{BATarget, TXTarget, LockTarget, AutoLockTarget\}$	7. $cflow(\tau) \cap \{TXOwner, TXTarget\} \neq \emptyset, tag(\tau) \cap \{LckTarget, BATarget, AutoLckTarget\} \neq \emptyset$
8.$owners := \{BAOwner, TXOwner\}$	

Table 2. Consistency checking rules

The FlexSync-based approach fully exhibits the benefit of the aspect oriented approach as, the synchronization implementations is not only the much modular (9 modules in FlexSync vs. 37 modules in the original implementations), the task of programming is also simpler for two reasons: 1. the coding effort is largely declarative in nature, meaning interactions patterns are widely used; 2. the "owner/target" relationship, which is latent and spread-out in conventional implementations, is explicit and local in FlexSync-based approaches, allowing easier reasoning and modification. The declarative degree of ORBacus is low due to the treatments of `wait/notify` semantics, a limitation we discussed previously.

App	*Orig*	*flex*	*dec*	*ND*	*ND'*	α	α'
specjbb	9	3	130	178	168	42%	93%
ORBacus	5	2	50	69	36	42%	60%
OpenJMS	23	3	313	159	109	66%	86%
Total	37	8	493	406	312	55%	84%

Table 3. Static assessment of FlexSync implementations

4.2 Functional characteristics of FlexSync

Performance of FlexSync

For assessing the performance of FlexSync, we use the specjbb2005 benchmark as an example of stand-alone Java application. For supporting transactional behaviors in specjbb, we label the group of class types representing business transactions as `TXOwner` or `BAOwner`. We label all types declaring `synchronized` methods with `TXTarget` or `BATarget`, respectively. In Figure 5, we plot the benchmark scores[16], bops (business operations per second), against the number of threads.

[16]All experiments are conducted on a 4-core Intel CPU running 2.6 Linux kernels using the JRockit R27 64-bit JVM. Each data point is an average of 3 identical runs.

Our first observation from Figure 5 is that, for the lock version, the FlexSync approach does not incur significant runtime overhead (about 5%). This is generally true in all our experiments as will be shown shortly. The general performance profile of specjbb2005 is the same as our motivating buffer example that the lock-based approach gives the best performance (highest score), followed by BA, then by STM. The difference is that the BA score is at about 30% of that of the lock-based, and the STM version is about 20%-25% of the BA version. The reason for this dramatic slowdown is that each specjbb2005 test involves a large amount of data for which the accesses need to be synchronized. As verified by our runtime profiling, in the BA version, over 20% of the CPU time is spent on lock contention which seriously limit the concurrency of the system. For the STM version, over 50% of the time is spent on backing up the data by the dstm2 runtime. The memory requirement for executing STM on specjbb ranges from 1 to 40 times larger as compared to the original version.

This experiment supports our motivation that each synchronization mechanism has its unique strengths and weaknesses. Clear trade-offs exist among properties such as correctness, safety, and performance. The FlexSync approach gives customization options to the users of Java programs and let them decide which properties should take the precedence in their application domain.

Case studies of compositions

In this study, we use OpenJMS as an example of complex program composed from reusable systems and capable of supporting multiple architectural configurations. OpenJMS uses the remote procedure call (RPC) as its transport level mechanism, which is supported by ORBacus, a general purpose RPC middleware. To support the transactional and atomic operations, we mark types contains `synchronized` methods as `TXTarget` or `BATarget`, and we make the starting point of RPC invocations in ORBacus and of the message dispatching in OpenJMS as where the transactional or the atomic executions start.

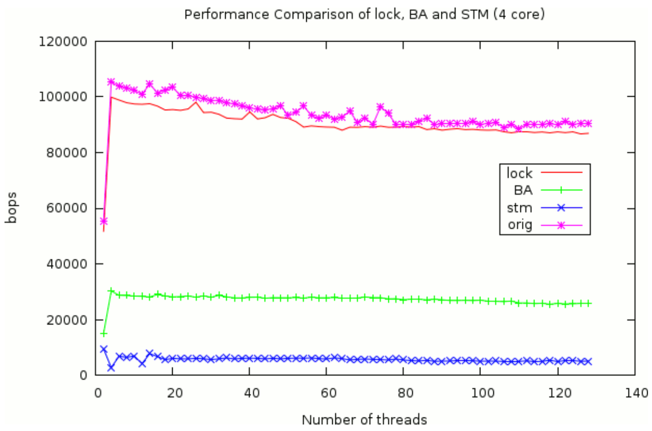

Figure 5. specjbb2005 FlexSync **performance**

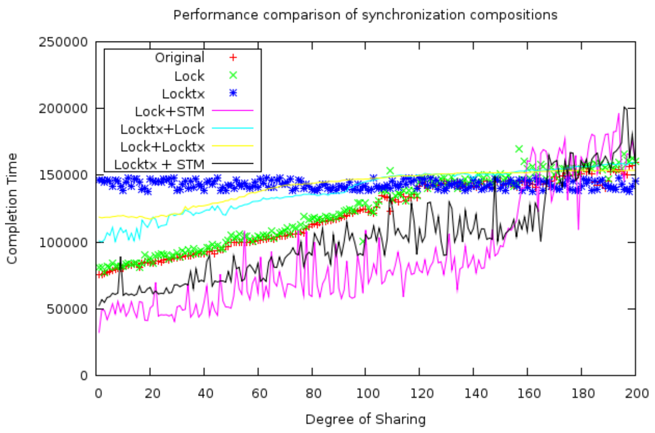

Figure 6. Combinatorial synchronization

In the first case study, we first gain a general perspective of the combinatorial complexity by quantifying all possible choices of synchronization mechanisms in the case of OpenJMS/ORBacus system. To obtain these points, we measure the response time for the JMS server in receiving a fixed number of messages into a set of message queues. We define *degree of sharing* as the average number of clients sharing each message queue. We achieve this by generating the client/queue association before each test and hardwiring the client/queue relationships during the run. We tested 7 possible configurations, reported in Figure 6. We make the following observations: 1. the lock version of FlexSync approach does not incur overhead as compared to the original version; 2. multiple synchronization mechanisms can coexist in providing JMS services; 3. the responses involving STM, although oscillating significantly, are faster than the lock-versions, which is contradictory to our previous studies; 4. the versions involving BA have the worst performance compared to other versions. The surprising results about STM is due to the fact that, as each transactional operation results from a round of client-server communication, the state duplication in OpenJMS is far less frequent compared to that of specjbb2005 and our buffer example. In the case of BA, the use of 2PL significantly limits the concurrent degree of the system, which proves that the degree of liveness is vital to the performance of server-type systems

Our second case study illustrates the scenarios of unanticipated compositions and how FlexSync-based synchronization understands these cases and achieves performance gains. The canonical concurrency policy used by the OpenJMS/ORBacus system is that each connected client is assigned a dedicated thread on the server side. Data structures storing the RPC targets and the message queues are shared among these threads.

Scenario one: Queue dedication If the physical capacity allows, the server side can dedicate a separate messaging stack for each OpenJMS client simply by publishing each queue with a unique RPC address. This set-up alone can improve the processing throughput from 7% to 25% by our measurements. This is unanticipated scenario that can be achieved purely through deployment configurations. In such configurations, since there are no shared states, we can simply instruct the FlexSync weaver not to weave any synchronization mechanisms to achieve further speed-ups.

Scenario two: Event-driven RPC The RPC engine can make use of the reactor-based [20] event-driven concurrency models for its capability of handling problems such as C10k[17]. Since such models typically make no use of threads, the RPC engine serially dispatches requests, rendering the thread-safety property of the upper layer messaging mechanism in OpenJMS redundant. Again, this is a per-deployment scenario hard to be anticipated by the design of OpenJMS. This scenario can be created by using the CAL-based ORBacus implementation from our earlier research [23]. The FlexSync sync weaver scans through the bytecode image of the entire system and is able to detect that all shared states of the messaging layer does not escape from the executing thread of the reactor. Therefore, no synchronization mechanisms are applied as the result.

In Figure 7, we compare the following configurations: the canonical concurrency model (*shared original*), original OpenJMS configured to run dedicated queues (*dedicated original*), original OpenJMS using reactor (*shared reactor*), dedicated queue using reactor (*dedicated reactor*), and the afore-mentioned two optimized versions using FlexSync (*dedicated flex* and *dedicated reactor flex*). Our measurements first justify the validity of the case study where queue dedications and the use the reactor can produce speedups from approximately 7% to 25% at 500 clients. In queue dedication scenarios, the FlexSync-

[17]The C10K problem. `http://www.kegel.com/c10k.html`

optimized version produces 23% speedups compared to the original version. For the use of the single-threaded reactor, the FlexSyncversion produces 25% speedups. The largest speedup, considering all configurations, is around 40%. These measurements prove the functional advantage of FlexSync-based synchronization implementations.

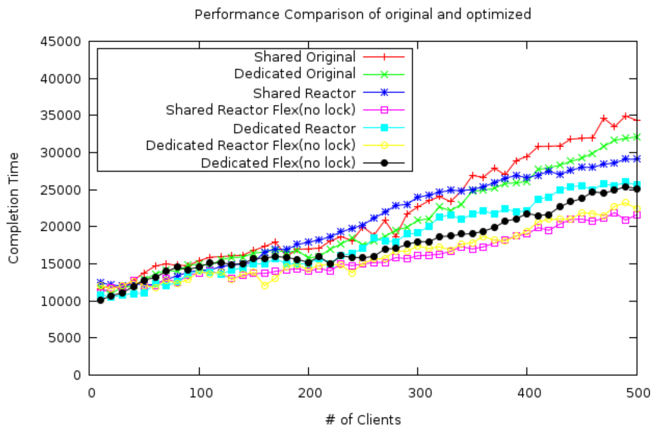

Figure 7. Performance optimization

5 Related Work

As a very active research area, research projects in the context of lock, atomicity and software transactional memory are beyond enumeration. We focus on presenting research addressing the programming aspect of synchronization challenges. We first covers the aspect oriented approaches to synchronization. We then present the research work on the synthetic approaches for conventional programming languages. We conclude with the discussion on various new language proposals.

AOP implementations
Lopes and Lieberherr [15] presented one of the earliest AOP treatments of synchronization using *adaptive programming* [14]. In their approach, the structure of a program and its behavior, including the lock-based synchronization, are expressed in separate modules. Code generation is required to produce the final executable system. As a pioneering work, they focused on illustrating a benefit of AOP-based synchronization implementation as compared to the conventional approach. Our work is built on these insights and going one step further in considering how different synchronization mechanisms can coexist, be customized, and interact with the core program consistently in complex Java systems.

SyncGen [5] focuses on generating synchronization implementation from high-level specifications. These specifications (aspects) are invariant formulae, which are translated into byte-code instructions that are inserted into (weaving) the demarcation points of the synchronized region. Compared to our approach, aside from the lock-only approach, SyncGen introduce a new programming paradigm of specifying synchronization in high-level logic formulae. Our approach relies on FlexSync APIs to re-express the synchronization intention, therefore, does not fundamentally deviate from how the synchronization design is reasoned conventionally.

Lock synthesis
Emmi et al [6] presented an automatic technique that takes a program annotated with atomic sections and produces a lock assignment for global variables that provides atomicity and deadlock free guarantees. Their work provides evidence that synchronization can be reasoned independently if we can know the programmers' intentions, in their case, through annotations. Research such as [4, 2] eliminates unnecessary lock placements through static analysis techniques. Our work directly leverages these results in performing selective aspect weaving.

Language approach
There have been a proliferation of new language proposals, such as [16, 8, 22, 3] and many others, that provide new language design and the semantic guarantees to help programmers in writing safe, correct, and performant synchronization code. Rewriting complex applications with new languages is not always straightforward. The majority of the language proposals, being focusing on specific synchronization mechanisms, also inherit their limitations. We believe that the capability of customization is still a desired property for systems written in these new languages.

6 Conclusion

In the multi-core era, concurrency plays critical roles in improving software efficiency. The synchronization mechanisms of reusable Java systems are challenging to build because each of these mechanisms has unique strengths and weaknesses which are sensitive to specific usage requirements. In conventional approaches, synchronization is reasoned locally within the designed application and in a non-modular way. As a result, applications pay significant performance costs due to the mismatch between the non-flexibility of the systems and the diversity of deployment scenarios.

We have presented FlexSync, an aspect oriented synchronization library, to alleviate this problem by physically decoupling the synchronization implementation from the operational logic of Java systems. This is based on our observation that the design intention of synchronization and the specific choice of synchronization mechanisms can be explicitly separated and, in practice, most of the intentions can be represented by function-like structures. The FlexSync API fosters the modular reasoning of synchronization by essentially enabling programmers to give different interpretations of the same program structure according to the different synchronization semantics. The FlexSync library encapsulates reusable logic about how synchronization mechanisms and the operational logic interact. The FlexSync loadtime weaver performs the global reasoning of synchronization by applying the system-wide deployment-time analysis to achieve consistency and optimization.

We evaluated FlexSync with commercially used complex Java concurrent systems, and FlexSync is capable of supporting the functionalities of these systems. We quantify both the programming effort and the functional characteristics of FlexSync-based implementations. We found that programming synchronization in FlexSync is commonly declarative and specification-like. The FlexSync approach in general does not incur significant runtime overhead. In addition, systems using FlexSync also has the capability of making customization choices regarding domain-specific or deployment-specific requirements. We showed that, the FlexSyncapproach does not significant incur runtime overhead and can produce speed ups as much as 40% in various deployment-time configurations.

As future work, we aim to make programming with FlexSync a lot easier by focus on the automation of intention refactoring and state duplication. We also plan to study the synergistic effects among customizable concurrency models [23] and synchronization mechanisms. Our long term research goal is to significantly increase the customization capability of complex systems.

References

[1] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI*, pages 26–37, New York, NY, USA, 2006. ACM.

[2] Jonathan Aldrich, Craig Chambers, Emin Gun Sirer, and Susan Eggers. Static analyses for eliminating unnecessary synchronization from java programs. In *SAS*, page pages, 1999.

[3] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: a dialect of java without data races. In *OOPSLA*, pages 382–400, New York, NY, USA, 2000. ACM.

[4] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Stack allocation and synchronization optimizations for java using escape analysis. *ACM TOPLAS*, 25(6):876–910, 2003.

[5] Xianghua Deng, Matthew B. Dwyer, John Hatcliff, and Masaaki Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *ICSE*, pages 442–452. ACM Press, 2002.

[6] Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *POPL*, pages 291–296, New York, NY, USA, 2007. ACM.

[7] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, pages 256–267, New York, NY, USA, 2004. ACM.

[8] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI*, pages 338–349, New York, NY, USA, 2003. ACM.

[9] Jan Hannemann and Gregor Kiczales. Design Pattern Implementation in Java and AspectJ. In *ACM OOPSLA*, pages 161–173. ACM Press, 2002.

[10] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *OOPSLA*, pages 253–262, New York, NY, USA, 2006. ACM.

[11] Elizabeth A. Kendall. Role model designs and implementations with aspect-oriented programming. In *ACM OOPSLA*, pages 353–369. ACM Press, 1999.

[12] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[13] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *ICSE*, pages 49–58, New York, NY, USA, 2005. ACM.

[14] Karl Lieberherr, Doug Orleans, and Johan Ovlinger. Aspect oriented programming with adaptive methods. In *Communications of the ACM*, volume 10. ACM, 2001.

[15] Cristina Videira Lopes and Karl J. Lieberherr. Abstracting process-to-function relations in concurrency object-oriented applications. In *ECOOP*, pages 81–99, London, UK, 1994. Springer-Verlag.

[16] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. In *POPL*, pages 346–358, New York, NY, USA, 2006. ACM.

[17] L. Northrop, P. Feiler, R. P. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, D. Schmidt, K. Sullivan, and K. Wallnau. Ultra-Large-Scale Systems: The Software Challenge of the Future. Software Engineering Institute, 2006.

[18] Venkatesh Prasad Ranganath and John Hatcliff. Pruning interference and ready dependences for slicing concurrent java programs. In *CC*, pages 39–56. Springer, 2004.

[19] Erik Ruf. Effective synchronization removal for java. In *PLDI*, pages 208–218, New York, NY, USA, 2000. ACM.

[20] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture Patterns for Concurrent and Networked Objects*, volume 2 of *Software Design Patterns*. John Wiley & Sons, Ltd, 1 edition, 1999.

[21] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC*, pages 204–213, New York, NY, USA, 1995. ACM.

[22] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, pages 334–345, New York, NY, USA, 2006. ACM.

[23] Charles Zhang and Hans-Arno Jacobsen. Externalizing Java Server Concurrency with CAL. In *ECOOP*, pages 362–386. Lecture Notes in Computer Science 5142. Springer, 2008.