Exploring opportunities for future multicore scalability requires fully harnessing potential concurrency that is latent in software interfaces. Recent researches theorize that software systems *can be implemented* in a way that scales when their interface operations commute [**?**, **?**]. This posits an baseline for software scalability. Nevertheless, *how to implement* software systems that exceeds the baseline remains an open research challenge. Based on the PI's past successful experience with wait-free data structure libraries [**?**], the PI suggests a holistic approach for developing future non-blocking transactional data structures — building an open source library of transactional data structures based on existing non-blocking data structures. This library is not only a compilation of a large number of fundamental data structures for multiprocessor applications, but also a framework for enabling composable transactions, and moreover, an infrastructure for continuous integration of new data structures. By taking such a top-down approach, the PI will be able to identify and consider the interplay of data structure interface operations as a whole, which allows for scrutinizing their commutativity rules and hence opens up possibilities for design optimizations.

The proposed research plan outlines the development of three infrastructure components, which will provide the technologies and tools needed to build an reusable high-performance software library of transactional data structures. The components include 1) a methodology to enable wait-free transactions for both linked and non-linked data structures, 2) a software framework for supporting composable transaction across multiple data structures, and 3) a source-to-source translation toolset, which automatically transforms existing non-blocking data structures into non-blocking transactional data structures. All three components will be built upon the foundation of lock-free transactional transformation. This previously described methodology embeds the transaction metadata in each node, which enables resolving transaction conflicts cooperatively through thread-level synchronization. It allows operations to correctly interpret the logical status of nodes left over by aborted transactions, and thus streamlines transaction execution by abolishing undo logs and rollbacks. The proposed research will be able to fully exploit the considerable amount of research effort devoted to the development of numerous highly scalable concurrent data structures in the past two decades.

As the first step, the PI will augment the lock-free transactional transformation methodology to support dictionary data types, non-linked data structures, and wait-free data structures. This make it possible to include a large quantity and variety of data structures that encompass most aspects of multiprocessor programming. To the best of the PI's knowledge, no existing transaction execution approach preserves wait-freedom, an important progress guarantee for real-time applications. A dictionary can be seen as an extended set where each key has a binding value, forming a key-value pair. A data structure implementing dictionary data type must support UPDATE operation in addition to the INSERT, FIND, and DELETE operations found in set data types. The update operation update the bond value for a given key without changing the presence of key itself. In short, the described methodology can be improved to encompasses dictionary data type as follows. The operation type enumeration for UPDATE will be added, as well as associated interpretation of the logical status of a nodes lastly accessed by an update operation. Two additional fields (old, and current value) will be added to the NodeInfo structure so that the previous version of the value can be recovered if update fails. The proposed methodology can also be extended to support non-linked data structures. Some of the previously described techniques, such as logical rollback via inverse logical status interpretations and co-operative transaction execution, are readily applicable. The key change is to devise an element-based conflict detection scheme, which is a generalization of the node-based conflict detection developed for linked data structures. The challenge lies in associating transaction descriptors with each element in non-linked data structure. In the experimental implementations of linked data structures, a transaction description reside in a node, co-locating with its associated key. This provides technical convenience as the transaction descriptor can be introduced by simply adding a new field in the existing node structure. For non-linked data structures where elements may be stored without a node, data encapsulation is needed in order to introduce the transaction descriptor. For example, in the case of an integer array data structure, each integer is stored adjacent to each other in a contiguous memory block. Each element in the transformed array will be converted into a memory cell in order to accommodate both the integer value and its associated transaction descriptor. Wait-freedom is the strongest progress guarantee for multiprocessor applications; it ensures that every thread makes progress in a finite amount of time. Since the proposed methodology employs base data structures methods for manipulating low-level memory blocks, it respects the base data structures progress guarantees in that regard. On the other hand, the proposed methodology also introduces a new code path for atomically updating logical statues. This code path is built on a CAS-based retry loop, which guarantees lock-free progress. The key challenge to achieve wait-freedom is that there is no guarantee that the CAS will eventually succeed. Although unlikely, other threads may be delayed indefinitely if one thread updates the value at an extreme high frequency. To prevent this potential danger, the PI will implement a progress assurance scheme using *announcement table* presented by Herlihy [**?**]. In short, each thread posts its operation in a shared announcement table. All threads need to scan the announcement table for any outstanding operations and perform helping if needed. The PI will further improve the performance of the wait-free version by adopting a construction technique called *fast-path-slow-path* proposed by Kogan and Petrank [**?**]. This efficient wait-free construction will execute the (fast) lock-free code path most of the time and fall back to the wait-free code path only when a thread is delayed for an extended period of time.

Secondly, supporting composable transaction for concurrent data structures is of paramount importance for building reusable software systems. Unfortunately, existing concurrently libraries are not composable: users cannot extend or compose the provided operations safely without breaking atomicity. Despite that concurrent data structures guarantee linearizability for individual method invocation, using concurrent data structures in client code is error pronec̃iteshacham2011testing. The problem stems from the inability of concurrent data structure to ensure atomicity of transactions composed from operations on several different concurrent data structures. This greatly hinders software reuse because users are only allowed to invoke the concurrent operations in a limited number of ways. For example, consider the case of money transfer between accounts, where a user who attempts to *move* a record from one concurrent container A to another B. Without knowing the nuances of using concurrent data structures, he constructs the move operation from back-to-back invocation of $A.remove()$ and $B.insert()$. This move operation is likely to fail. Software transaction memory (STM) [**?**, **?**] can be applied to obtain correct serialization of such transaction executions. However, recent research works indicate that STM is ineffective due to the overhead to track every single read and write accesses [**?**, **?**], which becomes especially cumbersome when handling larger transactions involving more operations. Furthermore, more and more relaxed STM systems are being proposed in pursuit of efficiency. This quest for improving STM parallelism has led to an unexpected and undesirable side effect: operations built with relaxed transaction may not be able to compose correctly [**?**, **?**]. The proposed research will produce the first software framework that allows for efficient execution of composable non-blocking transactions across multiple data structures. The previously described lock-free transactional transformation supports transaction execution of operations for a single data structure. The transaction execution and descriptor management is a built-in module of the data structure, the design of which is shaped by the focus on standalone application — granting users flexibility to obtain self-contained transaction data structure that has minimal external dependencies. To allow for executing transactions that involves operations across multiple data structures, the PI will externalize the transaction execution and create a dedicated framework for transaction descriptor management. A library of transactional data structures serves as a unified data structure pool where different instances of data structures have mutual awareness of the existence of each other. The PI will also develop an extended transaction descriptor format that encodes the data structure instances besides operation and operands. A transaction execution will first locate or create the required data structure before performing any operations. All transactional data structures within the same library will support the same transaction descriptor format, which enables co-operative transaction execution across multiple data structures.

Lastly, creating and supporting open source libraries are labor intensive and prone to human error. In order to minimize production efforts, the PI will develop an automated code transformation toolset to further improve the usability of the proposed methodology. More specifically, the PI will employ ROSE compiler framework to access program analysis algorithms and code generation tools to implement a source-to-source code translator. Due to the shear variety of data structure operation interfaces, programmers' annotation is needed to complement the information obtained from pure static analysis. In literature, static analysis with programmer annotation is also used to automatically infer atomic regions for transaction synchronization [?]. However, existing approaches synthesis mutual exclusion locks for each code block, which results in pessimistic synchronization. The proposed toolset is the first of its kind to generate non-blocking transactional data structure in a semi-automatic manner. As mentioned earlier, the lock-free transactional transformation involves two steps. The first step of encapsulating exiting methods to locate, insert and delete a node in the base data structure can be accomplished with the help of programmer annotations. In the proposed methodology, the encapsulated Do_LocatePred, Do_Insert, and Do_Delete methods are generalizations of existing node manipulation operations found in most concurrent linked data structures. When invoked in the context of transformed data structures, the internal mechanisms of these node manipulation operations are treated as black boxes. In the automated transactional transformation process, programmers will be asked to annotate exiting code blocks for node manipulations, i.e. expressing expected input and output arguments as well as start and end of an atomic block in an annotation language. The annotated API specifications are compared with the API specifications required by the proposed methodology. The necessary function mappings for arguments conversion can then be determined accordingly. Finally, the encapsulated Do_LocatePred, Do_Insert, and Do_Delete methods are generated automatically through source-to-source translation. The second step of applying code templates to construct transformed operations can be trivially automated in programming languages that support *Macro* or *Template* meta-programming (e.g., C/C++). The pre-defined code templates are provided as either macros of template functions. The transformed operations can be obtained by expanding the macros or instantiating the templates with correct instances of Do_LocatePred, Do_Insert, and Do_Delete produced by the first step. In programming languages that lack support of macro of template, the code templates can also be applied using pre-process scripts.