We claim:

1. A method of ~~in a computer system, the computer system comprising a processing unit and a compiler, the compiler configured with knowledge of software transactional memory operations, the method performed to~~ compiling~~e~~ a program, the program including software transactional memory blocks~~,~~ and the method comprising:

by a computer, optimizing the program to create an optimized program containing software transactional memory instructions;

wherein optimizing the program comprises:

(a) inserting one or more word-based direct-access software transactional memory instructions at the software transactional memory blocks;

(b) replacing the one or more word-based direct-access software transactional memory instructions with decomposed software transactional memory instructions in the program at the software transactional memory blocks; and

(c) performing optimizations on ~~ing (460)~~ the program to create the~~an~~ optimized program, such that at least some of the optimizations are configured to operate specifically on the decomposed ~~containing~~ software transactional memory instructions; ~~and~~

wherein performing optimizations comprises performing a common subexpression elimination procedure which is modified to operate on decomposed software transactional memory instructions such that a decomposed software transactional memory instruction to open an object for read, that comprises a parameter that references the object, is eliminated which takes place after a decomposed software transactional memory instruction to open the object for update within a same transaction;

wherein the optimizations are performed within software transactional memory constraints, the software transactional memory constraints comprising data dependencies which are introduced into the decomposed software transactional memory instructions;

wherein the data dependencies enforce a call order based on output variables for at least some of the decomposed software transactional memory instructions being input variables for subsequent decomposed software transactional memory instructions; and

compiling ~~(340)~~ the optimized program.

~~2. The method of claim 1, wherein optimizing the program comprises:~~

~~inserting direct-access software transactional memory instructions into the program at the software transactional memory blocks; and~~

~~performing optimizations on the program, including the software transactional memory instructions, to create an optimized program, such that at least some of the optimizations are configured to operate specifically on the direct-access software transactional memory instructions.~~

2. The method of claim 1, further comprising compiling the program into an intermediate representation, and wherein performing the optimizations comprises performing optimizations on the intermediate representation.

3. The method of claim 2, wherein the intermediate representation comprises representations of the decomposed software transactional memory instructions.

~~3. The method of claim 2, wherein performing optimizations comprises performing a common subexpression elimination procedure which is modified to operate on decomposed software transactional memory instructions.~~

~~4. The method of claim 3, wherein the common subexpression elimination procedure is modified such that an instruction to open an object for read is eliminated after an instruction to open an object for update within the same transaction.~~

4~~5~~. The method of claim 1~~3~~, wherein the common subexpression elimination procedure is further modified such that redundant reads and writes to memory addresses and redundant logs for memory addresses within a transaction are removed.

5~~6~~. The method of claim 4~~5~~, wherein the common subexpression elimination procedure is further modified such that reads and writes in a first transaction are eliminated if they are made redundant by reads or writes in a second transaction, wherein the first transaction is nested within the second transaction.

67. The method of claim 32, wherein performing the optimizations comprises performing code motion optimizations.

7. The method of claim 4, wherein the software transactional memory constraints comprise constraints on a control flow graph of the program.

8. A compiler system for compiling a program containing software transactional memory blocks, the system comprising:

a processor;

a memory;

a first compilation module configured to represent source code of the program as an intermediate representation comprising, at least in part, representations of decomposed software transactional memory instructions, the decomposed software transactional memory instructions replacing at least one or more word-based software transactional memory instructions; and

an optimization module configured to optimize an intermediate representation for the source code of the program, the intermediate representation comprising, at least in part, decomposed software transactional memory instructions;

wherein the optimization module is further configured to perform a common subexpression elimination on the decomposed software transactional memory instructions, the common subexpression elimination comprising removing a redundant decomposed software transactional memory instruction to open an object for reading, that comprises a parameter that references the object, located after a decomposed software transactional memory instruction to open the object for updating within a same transaction;

wherein the representations of decomposed software transactional memory instructions are optimized at least in part according to software transactional memory optimization rules, the software transactional memory optimization rules comprising data dependencies in the decomposed software transactional memory instructions;

wherein the data dependencies enforce a call order based on output variables for at least some of the decomposed software transactional memory instructions being input variables for subsequent decomposed software transactional memory instructions; and

a second compilation module configured to compile the optimized intermediate representation into an executable form.

9. The system of claim 8, wherein the optimization module is further configured to:

remove redundant memory and log accesses for software transactional memory instructions; and

perform code motion optimizations on software transactional memory instructions.

10. One or more computer-readable storage memories or magnetic disk containing instructions which, when executed by a computer, cause the computer to perform a method for optimizing a program comprising decomposed direct-access software transactional memory instructions, the method comprising:

receiving an intermediate representation of the program comprising a control-flow graph, the intermediate representation including representations of the decomposed direct-access software transactional memory instructions located in the program in software transactional memory blocks marked as atomic, wherein the decomposed direct-access software transactional memory instructions replace at least one or more word-based software transactional memory instructions in one or more transactions that perform operations directly to a heap; and

applying optimization rules specific to decomposed direct-access software transactional memory instructions in order to optimize the representations of the decomposed direct-access software transactional memory instructions, wherein applying optimization rules comprises:

performing a modified common subexpression elimination on the decomposed direct-access software transactional memory instructions which eliminates read open instructions which occur after update open instructions, wherein a respective read open instruction of the eliminated read open instructions comprises a parameter that references an object and the respective read open instruction is subsumed by a respective update open instruction of the update open instructions;

wherein the modified common subexpression elimination procedure also eliminates read open instructions of the decomposed direct-access software transactional

memory instructions in a first transaction which are made redundant and subsumed by read open instructions or update open instructions of the decomposed direct-access software transactional memory instructions in a second transaction, wherein the first transaction is nested within the second transaction;

removing redundant load-store elimination for decomposed direct-access software transactional memory instructions which comprise reads, writes, and entries to logs; and

performing code motion optimizations for decomposed direct-access software transactional memory instructions;

wherein the optimization rules comprise restrictions on the control-flow graph for the intermediate representation, the restrictions on the control-flow graph comprising data dependencies between decomposed direct-access software transactional memory instructions, wherein the data dependencies enforce a call order, at least for atomicity, which enforces that a location is open for reading when read or the location is open for updating when updated or a store is logged for the location when updated, the enforcement based on output variables for decomposed direct-access software transactional memory instructions being input variables for subsequent decomposed direct-access software transactional memory instructions.

~~8. The method of claim 2, wherein performing optimizations comprises augmenting at least one non-transactional memory operation which accesses an object outside of a memory transaction with one or more transactional memory operations which ensure atomicity between transactional memory accesses to the object and the non-transactional memory operation.~~

~~9. The method of claim 8, wherein augmenting a non-transactional memory operation comprises:~~

~~inserting an open operation before the non-transactional memory operation which opens the object for access by the non-transactional memory operation; and~~

~~inserting a commit operation after the non-transactional memory operation which determines if there was a conflicting access to the object during execution of the non-transactional memory operation.~~

10. The method of claim 9, wherein:

the non-transactional memory operation is a read operation;

the open operation is configured to retrieve an indication of the state of the object before execution of the non-transactional memory operation; and

the commit operation is configured to:

retrieve an indication of the state of the object after execution of the non-transactional memory operation; and

if the state of the object indicates a conflicting access, cause the open, read, and commit operations to loop until a read is possible.

11. The method of claim 9, wherein:

the non-transactional memory operation is a write operation;

the open operation is configured to obtain write access to the object; and

the commit operation is configured to commit the write made to the object.

12. The method of claim 9, wherein the open and commit commands utilize synchronization on the object to ensure that the non-transactional memory operation performs atomically.

13. The method of claim 9, wherein the open command blocks until it can open the object for access.

14. The method of claim 8, further comprising identifying one or more memory operations which access objects outside of memory transactions.

15. The method of claim 14, wherein identifying one or more memory operations comprises:

analyzing transactional memory accesses to determine fields which may be accessed by transactional memory operations during execution of the software;

identifying one or more non-transactional memory operations which access fields which may be accessed by transactional memory operations.

~~16. The method of claim 15, wherein, for any field which is guaranteed not to be accessed by transactional memory operations, augmenting of non-transactional memory operations which access that field is avoided.~~

~~17. The method of claim 16, wherein augmenting a non-transactional memory operation which accesses an object outside of a memory transaction with one or more transactional memory operations comprises inserting atomic memory transaction blocks around the non-transactional memory operation.~~

~~18. The method of claim 2, wherein performing optimizations comprises:~~

~~identifying references to objects which are always bound to newly-allocated objects in transactions at a set of program points; and~~

~~preventing software transactional memory operations on objects reached through the identified references, wherein the operations have no effect outside of the transactions.~~

~~19. The method of claim 18, wherein identifying references to objects which are always bound to newly-allocated objects at a set of program points occurs across procedure calls.~~

~~20. The method of claim 19, wherein identifying references to objects which are always bound to newly-allocated objects at a set of program points comprises performing a forward dataflow analysis on the program.~~

~~21. The method of claim 20, wherein the forward dataflow analysis comprises maintaining a data structure for each basic block which represents at a location in the basic block, for each variable in the basic block, if the variable is either known to always be assigned to newly-allocated objects, may possibly be assigned to an object allocated outside the transaction, or if information is unknown for the variable.~~

~~22. The method of claim 21, wherein the data structure comprises a map from each reference to data kept for each reference, where the data kept for each reference~~

~~comprises an updateable cell containing a lattice value and dependencies between references.~~

~~23. The method of claim 22, wherein maintaining the data structure comprises propagating information, using the dependencies between references, through the updateable cells of lattice values as the dataflow analysis proceeds.~~

~~24. The method of claim 23, wherein propagating information comprises spreading information to unmapped variables in the map from variables to updateable cells of lattice values before program statements for those unmapped variables are analyzed.~~

~~25. The method of claim 18, wherein preventing software transactional memory operations on objects reached through the identified references, wherein the operations have no effect outside of the transaction, comprises, for each reference, replacing update operations through the reference with update operations which, when decomposed into decomposed software transactional memory operations, do not include update log operations.~~

~~26. The method of claim 18, wherein preventing software transactional memory operations on objects reached through the identified references, wherein the operations have no effect outside of the transaction, comprises, after software transactional memory operations are decomposed into decomposed software transactional memory operations, removing decomposed log operations which operate through the identified references.~~

~~27. The method of claim 18, further comprising determining which of the identified references are to objects which are not accessible after the transaction; and~~

~~wherein preventing software transactional memory operations on objects reached through the identified references, wherein the operations have no effect outside of the transaction, comprises removing software transactional memory operations to open objects for read or update through the determined references.~~

~~28. The method of claim 2, wherein performing optimizations comprises:~~

~~locating a software transactional memory instruction to open a reference for reading which is known to be followed by a first software transactional memory instruction to open the reference for update during execution; and~~

~~substituting, for the instruction to open the reference for reading, a second instruction to open the reference for update.~~

~~29. The method of claim 28, the method further comprising removing the first instruction to open the reference for update.~~

~~30. The method of claim 29, removing the first instruction comprises using common subexpression elimination to remove redundant instructions, including the first instruction.~~

~~31. The method of claim 28, wherein the software is represented by a control flow graph and the method is performed on the control flow graph.~~

~~32. The method of claim 31, wherein locating an instruction to open a reference for reading comprises, for basic blocks in the control flow graph:~~

~~identifying an instruction to open a reference for reading;~~

~~recording the reference;~~

~~scanning forward through the basic block from the identified instruction;~~

~~when an assignment to the reference is found, stopping scanning for the identified instruction; and~~

~~when an instruction to open the reference for update is found, proceeding to substitute for the identified instruction.~~

~~33. The method of claim 31, wherein locating an instruction to open a reference for reading comprises, for basic blocks in the control flow graph:~~

~~identifying an instruction to open a reference for update;~~

~~recording the reference;~~

scanning backward through the basic block from the identified instruction;

when an assignment to the reference is found, stopping scanning for the identified instruction; and

when an instruction to open the reference for reading is found, proceeding to substitute for the found instruction to open the reference for reading.

34. The method of claim 31, wherein locating an instruction to open a reference for reading comprises performing a dataflow analysis which searches across basic block boundaries for instructions to open a reference for reading which are made redundant by instructions to open a reference for update.

35. The method of claim 34, wherein performing a dataflow analysis comprises:

maintaining a set of variables at the boundary of each basic block in the control flow graph which are known to reference references which are opened for update;

for each basic block:

scanning backwards through the control flow graph;

when an instruction to open a reference for update is found for a particular variable, adding that variable to the set of variables for the basic block;

when a definition is found for a particular variable, removing that variable from the set of variables for the basic block;

when the beginning of the basic block is found, intersecting the variable set for the basic block with variable sets stored at the end of basic blocks which lead directly to the basic block;

repeating the process until a stable set of variables is created for each basic block; and

for each basic block, considering instructions to open references referenced by the set of variables for the basic block for reading as redundant instructions.

36. The method of claim 2, wherein performing optimizations comprises:

identifying one or more procedures in the program which comprise one or more

~~software transactional memory operations which are able to be optimized outside of the procedure;~~

~~creating, for each of the one or more procedures, a cloned version of the procedure which allows the calling of the software transactional memory operations which are able to be optimized outside of the cloned version;~~

~~moving the one or more software transactional memory operations which are able to be optimized outside of the procedures; and~~

~~replacing calls to each of the one or more procedures with calls to its cloned version.~~

~~37. The method of claim 36, further comprising optimizing the program by removing redundancies amongst the one or more software transactional memory operations which are able to be optimized.~~

~~38. The method of claim 37, wherein optimizing the program comprises performing common subexpression elimination on the program.~~

~~39. The method of claim 37, wherein the one or more software transactional memory operations which are able to be optimized comprise operations to get a transaction manager.~~

~~40. The method of claim 39, wherein cloned versions of procedures comprising operations to get a transaction manager comprise procedures which take instances of a transaction manager as inputs.~~

~~41. The method of claim 37, wherein the one or more software transactional memory operations which are able to be optimized comprise operations to open references for reading or update, the references being passed to the procedures as inputs.~~

~~42. The method of claim 41, wherein cloned versions of procedures comprising operations to open references for reading or update comprise procedures which rely on the references being opened before the procedures are called.~~

~~43. The method of claim 42, wherein creating cloned versions of procedures comprising operations to open references for reading or update comprises performing a backwards dataflow analysis to determine which procedures comprise operations to open references for reading or update for references which are known at the beginnings of the procedures.~~