Lerna: Parallelizing Dependent Loops Using Speculation

Mohamed M. Saad
Alexandria University
Alexandria, Egypt
msaad@alexu.edu.eg

Roberto Palmieri
Lehigh University
Bethlehem, PA, USA
palmieri@lehigh.edu

Binoy Ravindran
Virginia Tech
Blacksburg, VA, USA
binoy@vt.edu

ABSTRACT

We present Lerna, an end-to-end tool that automatically and transparently detects and extracts parallelism from data dependent sequential loops using speculation combined with a set of techniques including code profiling, dependency analysis, instrumentation, and adaptive execution. Speculation is needed to avoid conservative actions and detect actual conflicts. Lerna targets applications that are hard-to-parallelize due to data dependency. Our experimental study involves the parallelization of 13 applications with data dependencies. Results on a 24-core machine show an average of 2.7x speedup for micro-benchmarks and 2.5x for the macro-benchmarks.

CCS CONCEPTS

• **Theory of computation → Concurrency**; • **Computing methodologies → Parallel computing methodologies**; **Concurrent algorithms**;

KEYWORDS

Code Parallelization, LLVM, Transactions

1 INTRODUCTION

Sequential code parallelization is a widely studied research field (e.g., [1, 24, 28, 56, 58]) that aims at extracting parallelism from sequential (often legacy) applications; it has gained particular traction in the last decade given the diffusion of multicore architectures as commodity hardware (offering affordable parallelism). Techniques for parallelizing sequential code are classified as manual, semi-automatic, and automatic. The classification indicates the amount of effort needed to rewrite/annotate the original application as well as the level of knowledge required on the codebase.

In this paper we focus on the automatic class, where the programmer is kept out of the parallelization process. We target sequential applications whose source code is no longer actively maintained, for these would benefit most from an automatic solution. In this class, effective solutions have been proposed in the past, but most assume that (or are well-behaved when) the application itself has no data dependencies, or dependencies can be identified [22, 25, 34] and handled prior the parallel execution [26, 56]. In practice, this supposes the possibility of identifying regions of the code that have no data dependencies [32, 49] through static analysis or that can be activated in parallel after having properly partitioned the dataset [36, 53, 56].

Static analysis of code is less effective if the application contains sections that could be activated in parallel but enclose computation that may affect their execution flow and accessed memory locations. This uncertainty leads the parallelization process to take the conservative decision of executing those sections serially, nullifying any possible gain. For convenience, we say that an application has *non-trivial* data dependencies if accessed data cannot be partitioned according to the threads' access pattern, and therefore the application execution flow cannot be disjoint.

In this paper we follow the direction proposed by Mehrara *et al.* in [38], which consists of *speculating* over those sections to capture the actual data dependencies at run time, thus if the current execution does not exhibit data dependencies, parallelism will be exploited. We go beyond the original intuition in [38] presenting *Lerna*, an integrated software tool that parallelizes sequential applications with non-trivial data dependencies, automatically. The main difference between Lerna and [38] (as well as many other existing solutions) is that, in the latter, programmer is required to annotate programs to be parallelized, Lerna deduces it with a combination of code profiling and refactoring. With Lerna, data-dependent applications can be parallelized and their performance increased thanks to the exploitation of parallelism on multicores.

In a nutshell, Lerna is a system that works with the source code's intermediate representation, compiled using

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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SYSTOR '18, June 4–7, 2018, HAIFA, Israel
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5849-1/18/06…$15.00
https://doi.org/10.1145/3211890.3211897

LLVM [35], and produces ready-to-run parallel code. Its parallelization process is preceded by a profiling phase that discovers blocks of code that are prone to be parallelized (i.e., loops). After that, the refactoring step uses the Transactional Memory (TM) [28] abstraction to mark each possible parallel task. Such TM-style transactions are then automatically instrumented by Lerna to make the parallel execution equivalent to the serial execution. This is how Lerna handles the parallelization of data dependent tasks.

Despite the high-level goal, deploying the above idea leads application performance to be slower than the sequential, non-instrumented execution without fine-grain optimizations. As an example, a blind loop parallelization entails wrapping the whole body of the loop within a transaction. By doing so, we either generate an excessive amount of conflicts on variables depending on the actual iteration count or the level of instrumentation produced to guarantee a correct parallel execution becomes high. Also, variables that have never been modified should not be transactionally accessed; and local processing should be taken out from the transaction execution to alleviate the cost of abort/retry. Lerna's transactifier pass provides all of these. It instruments a small subset of code instructions, which is enough to preserve correctness, and optimizes the processing by a mix of static optimizations and runtime tuning.

We evaluated Lerna's performance using a set of 13 applications including micro-benchmarks, STAMP [9], a suite of applications, and a subset of the PARSEC [8] benchmark. Lerna is on average 2.7× faster than the sequential version using micro-benchmarks (with a peak of 3.9×), and 2.5× faster considering macro-benchmarks (with a top speedup of one order of magnitude reached with STAMP). These results have been collected on a 24-core machine.

Lerna is the first end-to-end automatic parallelization tool that exploits TM. Its main contribution is on the design and development of a solution that integrates novel (e.g., the ordered TM algorithms) and existing (e.g., the static analysis) techniques in order to serve the goal of parallelizing sequential applications with non-trivial data dependencies.

2 RELATED WORK

Automatic parallelization has been extensively studied in the past. The papers in [14, 20] overview some of the most important contributions in the area.

Optimistic concurrency techniques, such as Thread-Level Speculation and Transactional Memory, have been proposed as a means for extracting parallelism from legacy code. Both techniques split an application into sections and run them speculatively on parallel threads. A thread may buffer its state or expose it. Eventually, the executed code becomes safe and it can proceed as if it was executed sequentially. Otherwise, the code's changes are reverted, and the execution is restarted. Some efforts combined TLS and TM through a unified model [7, 44, 45] to get the best of the two techniques.

Parallelization using thread-level speculation (TLS) has been studied using hardware [13, 27, 33, 54] and software [12, 17, 37, 38, 46]. It was originally proposed by Rauchwerger *et al.* [46] for parallelizing loops with independent data access – primarily arrays. The common characteristics of TLS implementations are that they largely focus on loops as a unit of parallelization, they mostly rely on hardware support or changes to the cache coherence protocols, and the size of parallel sections is usually small.

Regarding code parallelization and TM, Edler von Koch *et al.* [21] proposed an epoch-based speculative execution of parallel traces using hardware transactional memory (HTM). Parallel sections are identified at runtime based on binary code. The conservative nature of the design does not allow the full exploitation of all cores. Besides, relying only on runtime supports for parallelization introduces a non-negligible overhead to the framework. Similarly, DeVuyst *et al.* [18] uses HTM to optimistically run parallel sections, which are detected using special hardware.

STMLite [38], shares the same sweet-spot we aim for; namely, applications with non-partitionable accesses and data dependencies. STMLite provides a low-overhead access by eliminating the need for locks and constructing a read-set; instead, it uses signatures to represent accessed addresses. A central transactional manager orchestrates the in-order commit process with the ability of having concurrent commits. In contrast with Lerna, it requires user interventions to support the parallelization and it has centralized components forming possible performance bottlenecks.

Sambamba [55] showed that static optimization at compile-time does not exploit all possible parallelism. It relies on user input for defining parallel sections. Gonzalez *et al.* [24] proposed a user API for defining parallel sections and the ordering semantics. Based on user input, STM is used to handle concurrent sections. In contrast, Lerna does not require special hardware, it is fully automated with an optional user interaction, and it improves the parallel processing itself with specific pattern-dependent (e.g., loop) optimization.

The study in [57] classified applications into: sequential, optimistically parallel, or truly parallel, and tasks into: ordered (speculative iterations of loop), and unordered (critical sections). It introduces a TM model that captures data and inter-dependencies. The study showed important per-application [6, 9, 11, 41] features as the size of read and write sets, dependency density, and size of parallel sections.

Most of the methodologies, tools and languages for parallelizing programs target scientific and data parallel computation applications, where the actual data sharing is very limited and the data-set is precisely analyzed by the compiler

and partitioned so that the parallel computation is possible. Examples of those approaches include [30, 37, 40, 47]. Worth to mention ASC [58], a system that automatically predicts the evolution of a program and whether the produces jobs have partitioned accesses. It does that by leveraging speculation and a fast-forwarding technique that shares cached values among threads running subsequent jobs. Lerna does not require the programmer and offers innovations effective when the application exposes data dependencies with non-partitionable access patterns.

Lerna builds upon an initial concept named HydraVM [50]. HydraVM relies on a java virtual machine and uses transactions for parallelization. Unlike Lerna: HydraVM reconstructs the code at runtime through recompilation and reloading class definition. The extensive instrumentation for establishing a relation between basic blocks and their accessed memory addresses limits its usage to small size applications and prevents the achievement of high performance. Lerna overcomes all the above limitations.

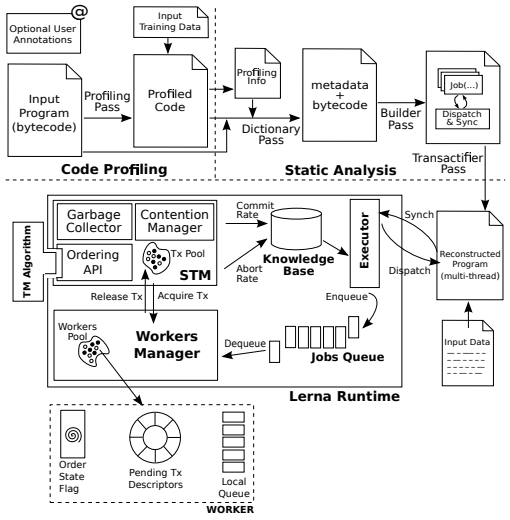


Figure 1: Lerna's Architecture and Workflow

3 LERNA

3.1 General Architecture and Workflow

Lerna splits the code of loops into parallel *jobs*. For each job, we create a synthetic method that: *i)* contains the code of the job; *ii)* receives variables accessed by the job as input parameters; *iii)* and returns the *exit* point of the job (i.e., the point where the loop break). Synthetic methods are executed by separate threads as memory transactions, and our TM library is used for managing their contention. While executing, each transaction operates on a private copy of the accessed memory. Upon a *successful* completion of the transaction, all modified variables are exposed to the memory.

We define a *successful execution* of a job as an execution that satisfies the following two conditions: *1)* it is reachable, meaning it is not preceded by a job that terminates early (e.g., using *break* instruction); and *2)* it does not cause a memory conflict with any other job having an older chronological order. Any execution of Lerna's parallel program is made of a sequence of jobs committed after a successful execution.

Lerna contains:

- an *automated software tool* that performs a set of transformations and analysis steps (*passes*) that run on the LLVM intermediate representation of the application code, and produces a refactored multi-threaded version of the program;
- a *runtime library* that is linked dynamically to the generated program, and is responsible for the following: organizing the transactional execution of dispatched jobs so that the original program order (i.e., the chronological order) is preserved, selecting the most effective number of worker threads according to the actual deployment and the feedbacks collected from the online execution, scheduling jobs to threads based on threads' characteristics (e.g., stack size, priority), and performing memory and computational housekeeping.

Figure 1 shows the architecture and the workflow of Lerna. Lerna relies on LLVM, thus it does not require the application to be written in one specific programming language. In this paper we focus on the fully automated process without considering any programmer intervention; however, although automated, Lerna's design does not preclude the programmer from providing hints that can be leveraged to make the refactoring process more effective, which will be discussed separately in Section 4.

Lerna's workflow includes the three steps in this order: *Code Profiling*, *Static Analysis*, and *Runtime*.

In the first step, our software tool executes the original application by activating our own profiler that collects some important parameters used later by the Static Analysis.

The goal of the Static Analysis is to produce a multi-threaded (also called reconstructed) version of the input program. This process follows the passes below.

Dictionary Pass. It scans the input program to provide a list of the functions of the byte-code (or *bitcode* as in LLVM) that we can analyze to determine how to transform them. By default, any call to an external function is flagged as *unsafe*. This information is important because transactions cannot contain unsafe calls, such as I/O system calls.

Builder Pass. It detects the code eligible for parallelization, it transforms this code into a callable synthetic method, and it defines the transaction's boundaries, meaning the transaction's begin and end.

Transactifier Pass. It applies the alias analysis [15] and detects depended memory operations in order to reduce the number of transactional reads and writes. It also provides the

instrumentation of memory operations invoked within the body of a transaction by wrapping them into transactional calls for read, write, and allocate.

Once the Static Analysis is complete, the reconstructed version of the program is linked to the application through the Lerna runtime library, which is mainly composed of the following three components:

- *Executor*. It dispatches the parallel jobs and provides the exit of the last job to the program. To exploit parallelism, the executor dispatches multiple jobs at-a-time by grouping them as a batch. Once a batch is complete, the executor simply waits for the result of this batch. Not all the jobs are enclosed in a single batch, thus the executor could need to dispatch more jobs after the completion of the previous batch. If no more job should be dispatched, the executor finalizes the execution of the parallel section.
- *Workers Manager*. It extracts jobs from a batch and it delivers ready-to-run transactions at available worker threads.
- *STM*. It provides the handlers for transactional accesses performed by executing jobs. In case a conflict is detected, it also behaves as a contention manager by aborting the conflicting transactions with the higher chronological order, this way the original program's order is respected. Lastly, it handles the garbage collection of the memory allocated by a transaction after it completes.

The runtime library makes use of two additional components: the *jobs queue*, which stores the (batch of) dispatched jobs until they are executed; and the *knowledge base*, which maintains the feedbacks collected from the execution in order to enable the adaptive behavior.

3.2 Code Profiling

Lerna uses the code profiling technique for identifying hotspot sections of the original code, namely those most visited during the execution. For example, it would not be effective to parallelize a for-loop with only two iterations. To do that, we consider the program as a set of *basic blocks*, where each is a sequence of non-branching instructions that ends either with a branch instruction, conditional or non-conditional, or a return.

Our goal is to identify the context, frequency and reachability of each basic block. To determine that information, we profile the input program by instrumenting its LLVM byte-code at the boundaries of any basic blocks to detect whenever a basic block is reached. This code modification does not affect the behavior of the original program. We call this version of the modified program *profiled byte-code*.

3.3 Program Reconstruction

Here we illustrate in detail the transformation from sequential code to parallel made during the static analysis phase. The LLVM intermediate representation is in the static single assignment (SSA) form. With SSA, each variable is defined before it is used, and it is assigned exactly once. Therefore, any use of such a variable has one definition, which simplifies the program analysis [42].

Dictionary Pass. In the dictionary pass, a full byte-code scan is performed to determine the list of accessible code (i.e., the dictionary) and, as a consequence, the external calls. Any call to an external function that is not included in the input program prevents the enclosing basic block from being included in the parallel code. However, the user can override this rule by providing a list of *safe* external calls. An external call is defined as *safe* if: *i)* it is revocable; *ii)* it does not affect the state of the program; and *iii)* it is thread safe. Examples of safe calls are stateless random generators.

Builder Pass. This pass is one of the core steps made by the refactoring process because it takes the code to transform along with the output of the profiling phase and makes it parallel by matching the outcome of the dictionary pass. In fact, if the profiler highlights an often invoked basic block that contains calls not in the dictionary, then that block cannot be parallelized.

The actual operation of building the parallel code takes place after the following two transformations.

Loop Simplification analysis. A *natural loop* has one entry block *header* and one or more back edges (*latches*) leading to the header. The predecessor blocks for the loop header are called *pre-header* blocks. We say that a basic block α dominates another basic block β if every path in the code β go through α. The *body* of the loop is the set of basic blocks that are dominated by its header, and reachable from its latches. The *exits* are basic blocks that jump to a basic block that is not included in the loop body. A *simple loop* is a natural loop with a single pre-header and single latch; and its index (if exists) starts from zero and increments by one. Loop simplification puts the loop into its simple form.

Induction Variable analysis. An *induction* variable is a variable of a loop whose value changes by a fixed amount every iteration or is a linear function of another induction variable. Affine (linear) memory accesses are commonly used in loops (e.g., arrays, recurrences). The index of the loop, if any, is often an induction variable, and the loop can contain more than one induction variable. The *induction variable substitution* [43] is a transformation to rewrite any induction variable in the loop as a closed form of its index. It starts by detecting the candidate induction variables; it then sorts them topologically and creates a closed symbolic form for each. Finally, it substitutes their occurrences with the corresponding symbolic form.

As a part of our transformation, a loop is simplified, and its induction variable is transformed into its canonical form

where it starts from zero and is incremented by one. A simple loop with multiple induction variables is a very good candidate for parallelization. However, induction variables introduce dependencies between iterations, which does not maximize parallelism. To solve this problem, the value of such induction variables is calculated as a function of the index loop prior to executing the loop body, and it is sent to the synthetic method as a runtime parameter.

Next, we extract the body of the loop as a synthetic method. The return value of this method is a numeric value representing the exit that should be used. Also, addresses of all accessed variables are passed as parameters.

The loop body is replaced by two basic blocks: *Dispatcher* and *Sync*. In *Dispatcher*, we prepare the arguments for the synthetic method, calculate the value of the loop index and invoke an API of our library, named *lerna_dispatch*, providing it with the address of the synthetic method and the list of the just-computed arguments. Each call to that API adds a job to our internal jobs queue, but it does not start the actual job execution. The *Dispatcher* keeps dispatching jobs until our API decides to stop. When this happens, the control passes to the *Sync* block. *Sync* immediately blocks the main thread and waits for the completion of the current jobs.

Regarding the exit of a job, we define two types of exits: *normal exit* and *breaks*. A normal exit occurs when a job reaches the loop latch at the end of its execution. In this case, the execution should go to the header and the next job should be dispatched. If there are no more dispatched jobs to execute and the last one returned a normal exit, then the *Dispatcher* will invoke more jobs. On the other hand, when the job exit is a break, then the execution needs to leave the loop body, and hence ignore all later jobs. For example, assume a loop with N iterations. If the Dispatcher invokes B jobs before moving to the Sync, then $\lceil N/B \rceil$ is the maximum number of transitions that can happen between Dispatcher and Sync. In summary, the Builder Pass turns the execution into the job-driven model, which can exploit parallelism.

Transactifier Pass. After turning the byte-code into executable jobs, we employ additional passes to encapsulate jobs into transactions. Each synthetic method is demarcated by *tx_begin* and *tx_end*, and any memory operation (i.e., load, stores or allocation) within the synthetic method is replaced by the corresponding transactional handler.

It is quite common that memory reads are numerous, and outnumber writes, thus it would be highly beneficial to minimize those performed transactionally. That is because, the read-set maintenance and the validation performed at commit time, which iterates over the read-set to preserve transaction correctness, is the primary source of overhead.

In the case of code parallelization, for which Lerna is designed, all parallel transactions have the characteristic that the code to be executed is the same. In fact, when Lerna parallelizes a loop, any application code executed after the loop is postponed until the parallelization of the loop itself terminates. Therefore, there cannot be any transaction executing code that does not belong to the body of the parallelized loop. We name such transactions as *symmetric*. The transactifier pass makes use of this unique characteristic of having symmetric transactions by relaxing the need to support TM strong atomicity [3], and by eliminating unnecessary transactional reads as explained below. This improves performance because non-transactional memory reads are even three times faster than transactional reads [10, 52].

Clearly, local addresses defined within the scope of the loop are not required to be accessed transactionally. Global addresses allow iterations to share information, and thus they need to be accessed transactionally. We perform the *global alias analysis* as a part of our transactifier pass to exclude some of the loads to shared addresses from the instrumentation process. To reduce the number of transactional reads, we apply the global alias analysis between all loads and stores in the transaction body. A load operation that will never alias with any store operation does not need to be read transactionally. For example, when a memory address is always loaded and never written in *any path* of the symmetric transaction code, then the load does not need to be performed transactionally. Note that this optimization can be applied only because our transactions are symmetric.

Sometimes the induction variable substitution cannot produce a closed form of the loop index, if it exists. For example, if a variable is incremented (or decremented) based on any arbitrary condition. If the address value is used only after the parallelized loop, then it is eligible for the *Read-Modify-Write* [48] optimization. With it, the increments (or decrements) are postponed at the transaction commit time. The modified locations are not part of the read-set, therefore, transactions do not conflict on these updates.

Transactions may contain calls to functions. As these functions may manipulate memory locations, they must be handled. When possible, we inline the called functions; otherwise we create a transactional version of the function called within a transaction. In that case, instead of calling the original function, we call its transactional version. Inlined functions are preferable because they permit the detection of dependent loops, or of dependencies between variables.

Finally, to avoid overhead in the presence of single-threaded computation or a single job executed at a time, we also keep a non-transactional version of synthetic methods.

3.4 Transactional Execution

The atomicity of transactions is mandatory, as it guarantees the consistency of the refactored code when run in parallel. However, if no additional care is taken, transactions commit

in any order (i.e., the fastest commits first), and that could revert the chronological order of the program, which must be preserved to ensure application correctness because Lerna's parallelization process is entirely automated.

A transaction in general (and not in Lerna) is allowed to commit whenever it finishes. This property is desirable to increase thread utilization and avoid fruitless stalls, but it can easily lead to erroneous computation where: *i)* transactions make memory modifications as a result of the computation of an unreachable iteration; or *ii)* transactions executing iterations with lower indexes read future values from committed transactions. If materialized, these scenarios clearly violate the logic of the original sequential program.

Motivated by that, we propose an ordered transactional execution model based on the original program's chronological order. Lerna's engine for executing transactions works as follows. Each transaction has an *age* identifier defining its chronological order in the program. Transactions can be in five states: *idle*, *active*, *completed*, *committed*, and *aborted*. A transaction is idle because it is still in the transactional pool waiting to be attached to a job to dispatch. A transaction becomes active when it is attached to a thread and starts its execution. When a transaction finishes the execution, it becomes completed. That means that the transaction is ready to commit, and it completed its execution without conflicting with any other transaction; otherwise it would abort and restart with the same age. Note that, a transaction in the complete state still holds its lock(s) until its commit. Finally, the transaction is committed when it becomes reachable from its predecessor transaction. Decoupling completed and committed states, permits threads to process next transactions.

Preserving Commit Order. Lerna requires an enhanced TM design for enforcing a specific commit order (i.e., lower-age transactions must not observe the changes made by higher-age transactions). We identified the following requirements needed to support ordering:

Supervised Commit. Threads are not allowed to commit once they complete their execution. Instead, there must be a single stakeholder at-a-time that performs transaction commits, namely the *committer*. It is not necessary to have a dedicated committer because worker threads can take over this role according to their age. For example, the thread executing the transaction with the lowest age could be the committer, and thus it is allowed to commit. While a thread is committing, other threads can proceed by executing next transactions speculatively, or wait until the commit completes. Allowing threads to proceed with their execution is risky because it can increase the contention probability given that the life of an uncommitted transaction enlarges (e.g., the holding time of their locks increases); therefore, this speculation must be limited by a certain (tunable) threshold. Upon a successful commit, the committer role is delegated to the subsequent thread with lowest age. This strategy allows only one thread to commit its transaction(s) at a time.

An alternative approach is to use a single committer (as in [38]) to monitor the completed transactions, and to permit non-conflicting threads to commit in parallel by inspecting their read- and write-set. Although this strategy allows for concurrent commits, the performance is bounded by the committer execution time.

Age-based Contention Management. Algorithms should implement a contention manager that favors transactions with lower age because they have to commit earlier and unblock waiting subsequent transactions.

Lerna currently integrates four TM implementations with different designs: NOrec [16], which executed commit phases serially without requiring any ownership record; TL2 [19], which allows parallel commit phases at the cost of maintaining an external data structure for storing meta-data associated with the transactional objects; *UndoLog* [23] with visible readers, which uses encounter time versioning and locking for accessed objects and maintains a list of accessors transactions; and STMLite [38], which replaces the need for locking objects and maintaining a read-set with the use of signatures. STMLite is the only TM library designed for supporting loop parallelization.

Among TM algorithms, NOrec has some interesting characteristics which nominate it as the best match for our framework. This is because, NOrec offers low memory access overhead with a constant amount of global meta-data. Unlike most STM algorithms, NOrec does not associate ownership records (e.g., locks or version number) with accessed addresses; instead, it employs a value-based validation technique during commit. It permits a single committing writer at a time, which matches the need of Lerna's concurrency control. Our modified version of NOrec decides the next transaction to commit according to the chronological order (i.e., age). Recently, in [51] the design of a new TM algorithm that fits Lerna's requirement has been presented.

High-priority Transactions. A transaction performs a read-set validation at commit time to ensure that its read-set has not been overwritten by any other committed transaction. Let Tx_n be a transaction that has just started its execution, and let Tx_{n-1} be its immediate predecessor (i.e., Tx_{n-1} and Tx_n process consecutive iterations of a loop). If Tx_{n-1} has been committed before that Tx_n performs its first transactional read, then we can avoid the read-set validation of Tx_n when it commits because Tx_n is now the highest priority transaction at this time, thus no other transaction can commit its changes to the memory. We do that by flagging Tx_n as *high-priority* transaction. A transaction is high-priority if: *i)* it is the first and thus does not have a predecessor; *ii)* it is a

retried transaction of the single committer thread; *iii)* there is a sequence of transactions with consecutive age running on the same thread.

4 ADAPTIVE RUNTIME AND OPTIMIZATION

The Adaptive Optimization System (AOS) [5] is an architecture that allows for online feedback-directed optimizations. In Lerna, we apply the AOS to optimize the runtime environment by tuning some important parameters (e.g., the batch size, the number of workers) and by dynamically refining sections of code already parallelized statically according to the characteristics of the actual application execution.

The Workers Manager (Figure 1) is the component responsible for executing jobs. Jobs are evenly distributed over workers. Each worker keeps a local queue of its slice of dispatched jobs and a circular buffer of completed transactions' descriptors. It is in charge of executing transactions and keeping them in the completed state once they finish. As stated before, after the completion of a transaction, the worker can speculatively begin the next transaction. However, to avoid unmanaged behaviors, the number of speculative jobs is limited by the size of its circular buffer. Its size is crucial as it controls the transaction lifetime. A larger buffer allows a worker to execute more transactions, but it also increases the transaction lifetime, and thus the conflict probability.

For the non-dedicated committer algorithms, the ordering is managed by a per-worker flag called *state flag*. This flag is read by the current worker, but is modified by its predecessor worker. After completing the execution of each job, the worker checks its local state flag to determine if it is permitted to commit or proceed to the next transaction. If there are no more jobs to execute, or the transactions buffer is full, the worker spins on its state flag. Upon successful commit, the worker resets its flag and notifies its successor to commit its completed transactions. Finally, if one of the jobs has a break condition (i.e., not the *normal exit*) the workers manager stops other workers by setting their flags to a special value. This approach maximizes the cache locality as threads operate on their own transactions and access thread-local data structures, which also reduces bus contention.

Batch Size. The static analysis does not always provide information about the number of iterations; hence, we cannot accurately determine the best size for batching jobs. A large batch may cause many aborts due to unreachable jobs, meaning jobs that should not be executed given that the execution terminated in some prior job. However, small batches increase the number of iterations between the *dispatcher* and the *executor* and therefore the number of pauses to perform due to Sync. The current implementation, evaluated in Section 5, uses an exponentially increasing batch size, meaning we add an exponentially increasing number of jobs to a batch until a predefined threshold that depends upon the number of executors in the system. Once a loop is executed, we record the last batch size used so that, if the execution goes back and calls the same loop, we do not need to perform again the initial tuning.

Jobs Tiling and Partitioning. Here we discuss an optimization, named *jobs tiling*, that allows the association of multiple jobs to a single transaction. Increasing jobs allows for assigning enough computation to threads, which outweighs the cost of transactional instrumentation. However, increasing tiles too much increases the size of read and write sets, which might degrade performance. Tiling is a runtime technique; we tune it by taking into account the number of instructions per job, and the commit rate of past executions using the *knowledge base*. A similar known technique is *loop unrolling* [4], in which a loop is re-written at compile time as a repeated sequence of its iteration code. Lerna employs static unrolling and runtime tiling according to the loop size. Figure 4 shows the impact in performance of tiling.

In contrast to tiling, a job may perform a considerable amount of non-transactional work. In this case, enclosing the whole job within the transaction boundaries makes the abort operation very costly. Instead, the transactifier pass checks the basic blocks with transactional operations and finds the nearest *common dominator* basic block for all of them. Given that, the transaction start (*tx_begin*) is moved to the common dominator block, and *tx_end* is placed at each *exit* basic block that is dominated by the common dominator. That way, the job is partitioned into non-transactional work, which is now moved out of the transaction scope, and the transaction itself, so that aborts become less costly.

Workers Selection. Figure 1 shows how the *workers manager* module handles the concurrent executions. The number of worker threads in the pool is not fixed during the execution, and it can be changed by the *executor* module. The number of workers affects directly the transactional conflict probability. The smaller the number of concurrent workers, the lower the conflict probability. However, optimistically increasing the number of workers can increase the overall parallelism (thus performance). In practice, at the end of the execution of a batch of jobs, we calculate the throughput and we record it into the *knowledge base* along with the commit rate, tiles and the number of workers involved. We apply a greedy strategy to find an effective number of workers by matching with the obtained throughput (Figure 2g in the evaluation section contrasts the variation of number of workers with batch size).

Finally, in some situations such as high contention or very small transactions, it is better to use a single worker. For this reason, if it is decided by our heuristic, then we use

the non-transactional version as a fast path of the synthetic method to avoid the unnecessary overhead.

Read-Only Methods. The alias analysis technique (Section 3.3) helps in detecting dependencies between loads and stores; however, in some situations [15] it results in conservative decisions, which limit parallelization. It is non-trivial for the static analysis to detect aliases throughout nested calls. To assist the alias analysis, we try to inline the called functions within the transactional context. Nevertheless, it is common in many programs to find a function that does only loads of immutable variables (e.g., reading memory input). Marking that as read-only can significantly reduce the number of transactional reads, as we will be able to use the non-transactional version of the function.

5 EVALUATION

In this section we evaluate Lerna and measure the effect of the key performance parameters (e.g., job size, worker count, tiling) on the overall performance. Our evaluation involves 13 applications grouped into micro- and macro-benchmarks.

We compare the speedup of Lerna over the sequential, and the manual (*manual unordered* in the plots) transactional version of the code, when available. Note that the latter it's provided directly by the benchmark (e.g., applications in micro-benchmarks and STAMP are already developed using transactions), thus it can leverage optimizations, such as the out-of-order completion, that cannot be caught by Lerna automatically (this is why we name it manual). Lerna's performance goal is twofold: providing a substantial speedup over the sequential code, and being as close as possible to the manual transactional version. It is worth noting that the reported performance of Lerna have been produced without any programmer intervention. The process is all automated.

The testbed consists of a server equipped with 2 AMD Opteron 6168 processors, each with 12-cores running at 1.9GHz. The memory available is 12GB and the cache sizes are 512KB for the L2 and 12MB for the L3. On this machine, the overall refactoring process, from the results of the profiling phase to the generation of the binary, takes ~10s for simple applications and ~40s for the more complex ones.

5.1 Micro-benchmarks

We consider micro-benchmarks [2] to evaluate the effect of different workload characteristics, such as the amount of transactional operations per job and its length, and the read/write ratio. Figure 2 reports the speedup over sequential code by varying the number of used threads. We report two versions of Lerna: one adaptive, where the most effective number of workers is selected at runtime and we report the ending setting in the plot, and one with a fixed number of workers. We also reported the percentage of aborted transactions (right y-axis). To improve the clarity of the presentation, in the plots we report the best results achieved with the different TM algorithms (often NOrec).

As a general comment, Lerna is very close to the manual transactional version. Unlike shown, the adaptive version of Lerna would never be slower than the single-threaded execution because, as fallback path, it would set the number of workers as one. The slow-down for the single thread is related to the fact that the thread adaptation is disabled for the competitor labeled "Lerna". Our adaptive version gains on average 2.7× over the original code and it is effective because it finds (or is close to) the configuration where the top performance is reached.

In *ReadNWrite1Bench* (Figures 2a, 2b), transactions read 1k locations and write 1 location. Thus, the transaction write-set is very small, and hence it implies a fast commit of a lazy TM as ours. The abort rate is low, and the transaction length is proportional to the read-set size. With long transactions, Lerna performs closer to the manual unordered; however, when transactions become smaller, the ordering overhead slightly outweighs the benefit of more parallel threads.

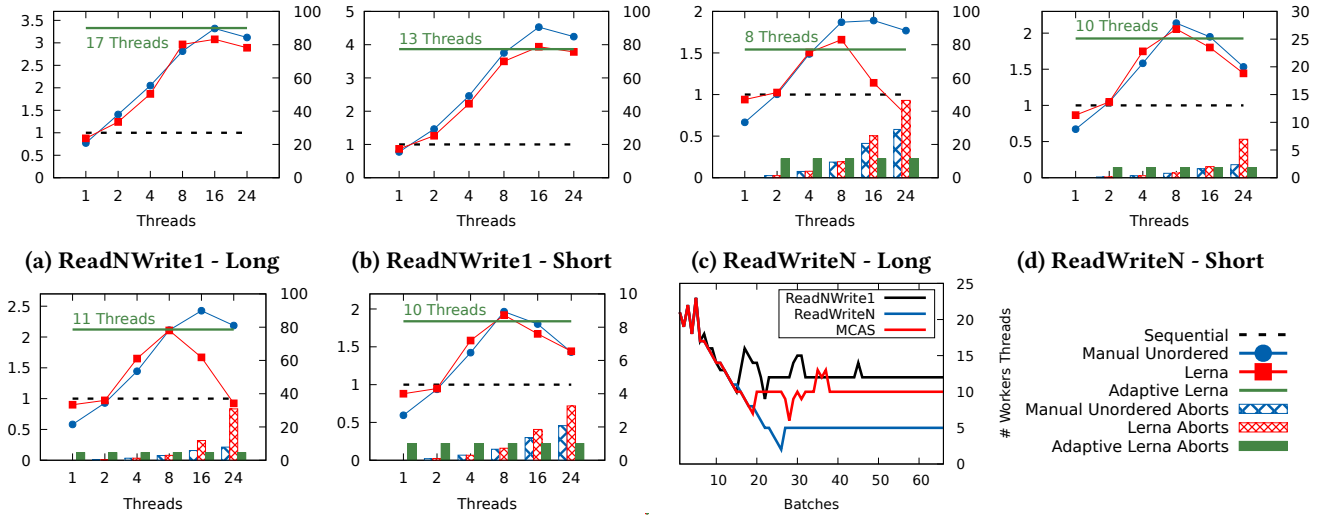
In *ReadWriteN* (Figures 2c and 2d), each transaction reads N locations, and then writes to another N locations. The large transaction write-set introduces a delay at commit time and increases the number of aborts. Both Lerna and manual unordered incur performance degradation at high numbers of threads due to the high abort rate (up to 50%). In addition, the commit phase of long transactions for Lerna forces some (ready to commit) workers to wait for their predecessor, thus degrading the overall performance. For that, the adaptive worker selection helps Lerna avoid this degradation.

MCASBench performs a multi-word compare and swap by reading and then writing N consecutive locations. Similar to ReadWriteN, the write-set is large, but the abort probability is lower than before because each pair of read and write acts on the same location. Figures 2e and 2f illustrate the impact of increasing workers with long and short transactions. Interestingly, unlike the manual unordered, Lerna performs better at single thread because it uses the fast path version of the jobs (non-transactional) to avoid any overhead.

Figure 2g shows the adaptive selection of the number of workers while varying the size of the batch. The procedure starts by trying different worker counts within a fixed window (i.e., 7), then it picks the best according to the calculated throughput. Changing the worker count shifts the window looking for a more effective setting.

5.2 The STAMP Benchmark

STAMP [9] is a suite with eight applications covering different domains (Yada and Bayes have been excluded because they expose non-deterministic behaviors). Figure 3 shows



(a) ReadNWrite1 - Long (b) ReadNWrite1 - Short (c) ReadWriteN - Long (d) ReadWriteN - Short

(e) MCAS - Long (f) MCAS - Short (g) Adaptive workers selection

Figure 2: Micro-benchmarks. In (a) to (f), left y-axis shows the speedup; right y-axis is the percentage of aborts.

the speedup of Lerna's transformed code over the sequential code, and against the manual transactional version of the applications, which exploits unordered commits. In this plot, the automatic tiling optimization is disabled.

Kmeans, a clustering algorithm, iterates over a set of points and associates them to clusters. The main computation is in finding the nearest point, while shared data updates occur at the end of each iteration. Using job partitioning, Lerna achieves 21× (Low contention) and 7× (High contention) speedup over the sequential code, using NOrec. Under high contention, NOrec is 3× slower compared to the manual unordered transactional version (more data conflicts and stalling overhead); however, they are very close in the low contention scenario. TL2 and STMLite suffer from false conflicts, which limits their scalability.

Genome, a gene sequencing program, reconstructs the gene sequence from segments of a larger gene. It uses a shared hash-table to organize the segments and eliminate duplicates. Lerna has 16-19× speedup over sequential. Genome conducts a large number of read-only transactions (*exists* operation), a friendly behavior for implemented algorithms. TL2 is just 10% slower than the manual competitor.

Vacation is a travel reservation system where the workload consists of clients reservation. This application emulated an OLTP workload. Lerna improves the performance by 2.8× faster than sequential, and it is very close to the manual.

SCAA2 is a multi-graph kernel application. The core of the kernel uses a shared graph structure updated at each iteration. The transformed kernel outperforms the original by 2.1× using NOrec, while dropping the in-order commit allows up to 4.4×. It is worth noting that NOrec is the only algorithm that manages to achieve speedup because it tolerates high contention and is not affected by false sharing as it deploys a value-based validation.

Lerna exhibits no speedup using *Labyrinth* and *Intruder* because, from the analysis of the application code, they use an internal shared queue for storing the processed elements and they access it at the beginning of each iteration to dispatch (i.e., a single contention point). While our jobs execute as a single transaction, the manual transactional version creates multiple transactions per iteration. The first iteration handles just the queue synchronization, while others do the processing. Adverse behaviors like this are discussed in later.

As explained in Section 4, selecting the number of jobs per each transaction (jobs tiling) is crucial for performance. Figure 4 shows the speedup and abort rate with changing the number of jobs per transaction from 1 to 10000 using the Genome benchmark. Although the abort rate decreases when reducing the number of jobs per transaction, it does not achieve the best speedup. The reason is that the overhead for setting up transactions nullifies the gain of executing small jobs. For this reason, we dynamically set the job tiling according to the job size and the gathered throughput.

The manual tuning further assists Lerna for improving the code analysis and eliminating avoidable overheads. An evidence of this is reported in Figure 5 where we show the speedup of Kmeans High against different numbers of workers using two variants of the transformed code: the first is the normal automatic transformation, and the second leverages user hints about memory locations that can be accessed safely (see Section 3.3). These results show that

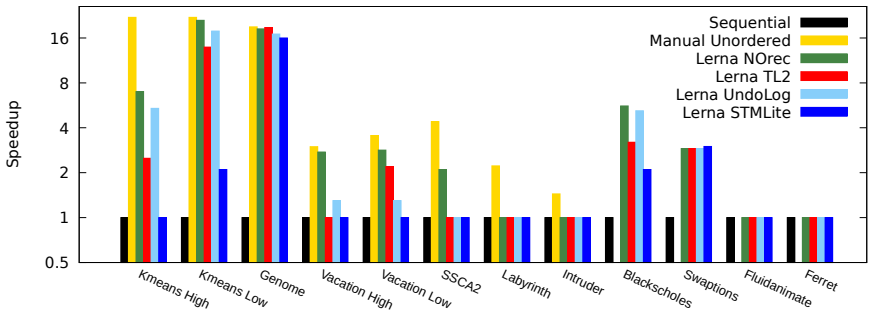


Figure 3: STAMP & PARSEC Benchmarks Speedup. (y-axis in log-scale)

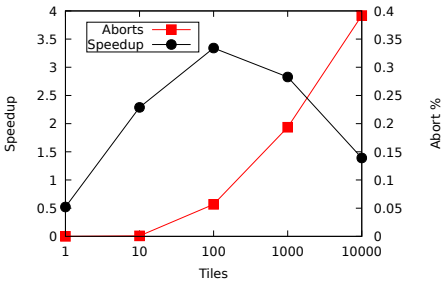


Figure 4: Effect of Tiling using 8 workers and Genome.

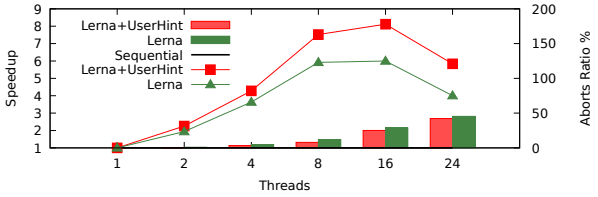


Figure 5: Kmeans performance.

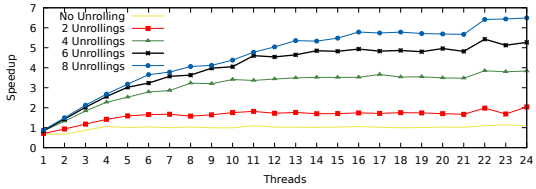


Figure 6: Impact of unrolling using Black-Scholes.

Lerna's framework can be deployed even more effectively if the programmer knows aspects of the original code.

5.3 The PARSEC Benchmark

PARSEC [8] is a benchmark suite for shared memory chip-multiprocessors architectures. For these applications, the manual unordered version is not included because PARSEC does not provide a transactional version of the code.

The Black-Scholes equation [31] is a differential equation that describes how the value of an option changes as the price of the underlying asset changes. This benchmark calculates Black-Scholes equation for different inputs. Iterations are relatively short, which generates many jobs in Lerna's transformed code. However, jobs can be tiled (Section 4). The speedup achieved is 5.6×. Figure 6 shows the speedup with different configurations of the loop unrolling.

Swaptions benchmark contains routines to compute various security prices using Heath-Jarrow-Morton (HJM) [29]. Swaptions employs Monte Carlo (MC) simulation to compute prices. The workload produced by this application provide similar speedup with all TM algorithms integrated.

The following two applications have some workload characteristic that disallow Lerna to produce an effective parallel code. *Fluidanimate* [39] is an application performing physics simulations (about incompressible fluids) to animate arbitrary fluid motion by using a particle-based approach. The main computation is spent on computing particle densities and forces, which involves six levels of loops nesting updating a shared array structure. However, iterations updates a global shared matrix of particles, which makes every concurrent transaction conflict with its preceding transactions.

Ferret is a toolkit used for content-based similarity search. The benchmark workload is a set of queries for image similarity search. Similar to *Labyrinth* and *Intruder*, Ferret uses a shared queue to process its queries; which represents a single contention point and prevents any speedup with Lerna.

5.4 Lerna Limitations

As confirmed by our evaluation study, there are scenarios where Lerna encounters obstacles that cannot be overcome due to the lack of semantics. Examples include, data structure operations, loops with few iterations because the actual application parallelization degree is limited, there is an irreducible global access at the beginning of each loop iteration, and the workload is heavily unbalanced across iterations.

6 CONCLUSION

We presented Lerna, an end-to-end automated system that combines a tool and a runtime library to extract parallelism from sequential applications with data dependencies, efficiently. Lerna overcomes the pessimism of the static analysis of the code by exploiting speculation.

ACKNOWLEDGMENTS

Authors would like to thank the shepherd Prof. Margo Seltzer, Henry Daly, and the anonymous reviewers for their valuable comments. This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0367.

REFERENCES

[1] [n. d.]. Intel Parallel Studio. ([n. d.]). https://software.intel.com/\en-us/intel-parallel-studio-xe.

[2] [n. d.]. RSTM: The University of Rochester STM. ([n. d.]). www.cs.rochester.edu/research/synchronization/\rstm/.

[3] Martín Abadi, Tim Harris, and Mojtaba Mehrara. 2009. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *ACM Sigplan Notices*, Vol. 44. ACM, 185–196.

[4] Alfred V Aho, Jeffrey D Ullman, et al. 1977. *Principles of compiler design*. Addision-Wesley Pub. Co.

[5] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. 2000. Adaptive optimization in the Jalapeno JVM. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '00)*. ACM, New York, NY, USA, 47–65. https://doi.org/10.1145/353171.353175

[6] David A Bader and Kamesh Madduri. 2005. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. In *High Performance Computing–HiPC 2005*. Springer, 465–476.

[7] Joao Barreto, Aleksandar Dragojevic, Paulo Ferreira, Ricardo Filipe, and Rachid Guerraoui. 2012. Unifying thread-level speculation and transactional memory. In *Proceedings of the 13th International Middleware Conference*. Springer-Verlag New York, Inc., 187–207.

[8] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 72–81. https://doi.org/10.1145/1454115.1454128

[9] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford Transactional Applications for Multi-Processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*.

[10] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. 2007. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*.

[11] B Chan. 2002. The UMT Benchmark Code. *Lawrence Livermore National Laboratory, Livermore, CA* (2002).

[12] Michael Chen and Kunle Olukotun. 2003. TEST: a tracer for extracting speculative threads. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*. IEEE, 301–312.

[13] Michael K Chen and Kunle Olukotun. 2003. The Jrpm system for dynamically parallelizing Java programs. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*. IEEE, 434–445.

[14] Doreen Y Cheng. 1993. A survey of parallel programming languages and tools. *Computer Sciences Corporation, NASA Ames Research Center, Report RND-93-005 March* (1993).

[15] Rezaul A Chowdhury, Peter Djeu, Brendon Cahoon, James H Burrill, and Kathryn S McKinley. 2004. The limits of alias analysis for scalar optimizations. In *Compiler Construction*. Springer, 24–38.

[16] Luke Dalessandro, Michael F Spear, and Michael L Scott. 2010. NOrec: streamlining STM by abolishing ownership records. In *ACM Sigplan Notices*, Vol. 45. ACM, 67–78.

[17] Francis Dang, Hao Yu, and Lawrence Rauchwerger. 2001. The R-LRPD test: Speculative parallelization of partially parallel loops. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002*. IEEE, 10–pp.

[18] Matthew DeVuyst, Dean M Tullsen, and Seon Wook Kim. 2011. Runtime parallelization of legacy code on a transactional memory system. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*. ACM, 127–136.

[19] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional Locking II. In *In Proc. of the 20th Intl. Symp. on Distributed Computing*.

[20] Nicholas DiPasquale, T Way, and V Gehlot. 2005. Comparative survey of approaches to automatic parallelization. *MASPLASâĂŹ05* (2005).

[21] Tobias JK Edler von Koch and Björn Franke. 2013. Limits of region-based dynamic binary parallelization. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 13–22.

[22] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *International journal of parallel programming* 21, 5 (1992), 313–347.

[23] Pascal Felber, Christof Fetzer, and Torvald Riegel. 2008. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, February 20-23, 2008*, Siddhartha Chatterjee and Michael L. Scott (Eds.). ACM, 237–246.

[24] MA Gonzalez-Mesa, Eladio Gutierrez, Emilio L Zapata, and Oscar Plata. 2014. Effective Transactional Memory Execution Management for Improved Concurrency. *ACM Transactions on Architecture and Code Optimization (TACO)* 11, 3 (2014), 24.

[25] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. 2011. Polly-Polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Vol. 2011.

[26] Manish Gupta, Sayak Mukhopadhyay, and Navin Sinha. 2000. Automatic parallelization of recursive procedures. *International Journal of Parallel Programming* 28, 6 (2000), 537–562.

[27] Lance Hammond, Mark Willey, and Kunle Olukotun. 1998. Data Speculation Support for a Chip Multiprocessor. *SIGOPS Oper. Syst. Rev.* 32, 5 (Oct. 1998), 58–69.

[28] Tim Harris, James Larus, and Ravi Rajwar. 2010. Transactional Memory, 2nd edition. *Synthesis Lectures on Computer Architecture* 5, 1 (2010), 1–263. https://doi.org/10.2200/S00272ED1V01Y201006CAC011

[29] David Heath, Robert Jarrow, and Andrew Morton. 1992. Bond pricing and the term structure of interest rates: A new methodology for contingent claims valuation. *Econometrica: Journal of the Econometric Society* (1992), 77–105.

[30] Shan Shan Huang, Amir Hormati, David F. Bacon, and Rodric M. Rabbah. 2008. Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary. In *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*. 76–103.

[31] Natanael Karjanto, Binur Yermukanova, and Laila Zhexembay. 2015. Black-Scholes equation. *arXiv preprint arXiv:1504.03074* (2015).

[32] Hironori Kasahara, Motoki Obata, and Kazuhisa Ishizaka. 2001. Automatic coarse grain task parallel processing on smp using openmp. In *Languages and Compilers for Parallel Computing*. Springer, 189–207.

[33] Venkata Krishnan and Josep Torrellas. 1999. A chip-multiprocessor architecture with speculative multithreading. *Computers, IEEE Transactions on* 48, 9 (1999), 866–880.

[34] Leslie Lamport. 1974. The parallel execution of DO loops. *Commun. ACM* 17, 2 (1974), 83–93.

[35] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 75–86.

[36] Amy W Lim and Monica S Lam. 1997. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 201–214.

[37] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. 2006. POSH: a TLS compiler that exploits program structure. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 158–167.

[38] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. 2009. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI '09)*. ACM, New York, NY, USA, 166–176. https://doi.org/10.1145/1542476.1542495

[39] Matthias Müller, David Charypar, and Markus Gross. 2003. Particle-based Fluid Simulation for Interactive Applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA '03)*. 154–159.

[40] Stefan C. Müller, Gustavo Alonso, Adam Amara, and André Csillaghy. 2014. Pydron: Semi-Automatic Parallelization for Multi-Core and the Cloud. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 645–659.

[41] AB MySQL. 1995. *MySQL: the world's most popular open source database*. MySQL AB.

[42] Nomair A Naeem and Ondrej Lhoták. 2009. Efficient alias set analysis using SSA form. In *Proceedings of the 2009 international symposium on Memory management*. ACM, 79–88.

[43] William Morton Pottenger. 1995. *Induction variable substitution and reduction recognition in the Polaris parallelizing compiler*. Ph.D. Dissertation. Citeseer.

[44] Arun Raman, Hanjun Kim, Thomas R Mason, Thomas B Jablin, and David I August. 2010. Speculative parallelization using software multi-threaded transactions. In *ACM SIGARCH Computer Architecture News*, Vol. 38. ACM, 65–76.

[45] Ravi Ramaseshan and Frank Mueller. 2008. Toward thread-level speculation for coarse-grained parallelism of regular access patterns. In *Workshop on Programmability Issues for Multi-Core Computers*. 12.

[46] Lawrence Rauchwerger and David A Padua. 1999. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *Parallel and Distributed Systems, IEEE Transactions on* 10, 2 (1999), 160–180.

[47] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. 2013. Dandelion: a compiler and runtime for heterogeneous systems. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 49–68. https://doi.org/10.1145/2517349.2522715

[48] Wenjia Ruan, Yujie Liu, and Michael Spear. 2015. Transactional read-modify-write without aborts. *ACM Transactions on Architecture and Code Optimization (TACO)* 11, 4 (2015), 63.

[49] Radu Rugina and Martin Rinard. 1999. Automatic parallelization of divide and conquer algorithms. In *ACM SIGPLAN Notices*, Vol. 34. ACM, 72–83.

[50] Mohamed M. Saad, Mohamed Mohamedin, and Binoy Ravindran. 2012. HydraVM: Extracting Parallelism from Legacy Sequential Code Using STM. In *4th USENIX Workshop on Hot Topics in Parallelism, HotPar'12, Berkeley, CA, USA, June 7-8, 2012*.

[51] Mohamed M. Saad, Roberto Palmieri, and Binoy Ravindran. 2016. On ordering transaction commit. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, Rafael Asenjo and Tim Harris (Eds.). ACM, 46:1–46:2.

[52] Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. 2006. Architectural Support for Software Transactional Memory. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, Washington, DC, USA, 185–196. https://doi.org/10.1109/MICRO.2006.9

[53] Joel H Saltz, Ravi Mirchandaney, and K Crowley. 1991. The Preprocessed Doacross Loop.. In *ICPP (2)*. 174–179.

[54] J Greggory Steffan, Christopher B Colohan, Antonia Zhai, and Todd C Mowry. 2000. *A scalable approach to thread-level speculation*. Vol. 28. ACM.

[55] Kevin Streit, Clemens Hammacher, Andreas Zeller, and Sebastian Hack. 2013. Sambamba: runtime adaptive parallel execution. In *Proceedings of the 3rd International Workshop on Adaptive Self-Tuning Computing Systems*. ACM, 7.

[56] Hans Vandierendonck, Sean Rul, and Koen De Bosschere. 2010. The Paralax infrastructure: automatic parallelization with a helping hand. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 389–400.

[57] Christoph von Praun, Rajesh Bordawekar, and Calin Cascaval. 2008. Modeling optimistic concurrency using quantitative dependence analysis. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 185–196.

[58] Amos Waterland, Elaine Angelino, Ryan P. Adams, Jonathan Appavoo, and Margo I. Seltzer. 2014. ASC: automatically scalable computation. In *ASPLOS*, Rajeev Balasubramonian, Al Davis, and Sarita V. Adve (Eds.). ACM, 575–590.