Lerna: Transparent and Effective Speculative Loop Parallelization

Mohamed M. Saad   Roberto Palmieri   Binoy Ravindran
Virginia Tech
{msaad, robertop, binoy}@vt.edu

Abstract

In this paper, we present Lerna, a system that automatically and transparently detects and extracts parallelism from sequential code using speculation combined with a set of techniques including code profiling, dependency analysis, instrumentation, and adaptive execution. Lerna is cross-platform and independent of the programming language. The parallel execution exploits memory transactions to manage concurrent and out-of-order memory accesses. This scheme allows Lerna to parallelize sequential applications with data dependencies. Our experimental study involves the parallelization of 13 applications. Results show an average of 2.7x speedup for micro-benchmarks and 2.5x for the macro-benchmarks.

1. Introduction

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

In this paper we focus on the automatic class, where the programmer is kept entirely out of the parallelization effort because we believe there is a large number of legacy applications that are sequential and whose source code is not actively maintained anymore, thus they could benefit most from an effective solution. In this class, effective solutions have been proposed in the past, but most of them assume that (or are well-behaved when) the application itself has no data dependencies, or dependencies can be identified [23, 25, 34] and taken into account prior to the parallel execution [26, 56]. In practice, this entails that there is the possibility of identifying regions of the code (e.g., loops) that do not have data dependencies [32, 50], through the static analysis of the code, or that can be activated in parallel after having partitioned the dataset properly [36, 53, 56].

Unfortunately, the analysis of the code is not effective if the application contains sections that could be activated in parallel but enclose computation that may affect the execution flow, as well as the input values (e.g., accesses to a shared data structure without specific patterns). This uncertainty leads the parallelization process to take the conservative decision of executing those sections serially, nullifying any possible gain.

In this paper, we take a different direction by *speculating* over those sections to capture the actual data dependencies at run time, thus if the current execution pattern does not produce data dependencies, parallelism can be still exploited. We encapsulate this idea into *Lerna*, the first integrated software tool that parallelizes sequential applications with *non-trivial* data dependencies (i.e., if data cannot be partitioned according to the threads' access pattern, and therefore the application execution flow cannot be disjoint) automatically, consequently the programmer is not required to annotate the code or to have a-priori knowledge on the application business logic. Lerna lets hard-to-parallelize sequential applications benefit from the real parallelism of multicore architectures.

Lerna deploys a building block fundamental to support the targeted type of parallel execution, i.e., *optimistic synchronization* (or speculation). With this technique, sequential sections of the code run in parallel optimistically (or speculatively), guarded by a compensating mechanism for handling operations violating the application consistency. This mechanism is carried out by Transactional Memory (TM) [28], a programming abstraction that permits developers to define critical sections as simple *atomic* blocks, which are internally managed by the TM itself. TM's adoption grew in recent years, especially after the integration with the popular *GCC* compiler (starting with version 4.7). A similar mechanism is also provided by Thread-Level Speculation (TLS) [27]. However, it is intrusive and more difficult to decouple from Lerna's design than TM. In fact, Lerna uses a TM implementation as an internal support to manage the data dependencies (which becomes *contention* when the code executes in parallel).

In a nutshell, Lerna is a system that does not require analysis of the application's source code; it works with its intermediate representation, compiled using *LLVM* [35], and produces ready-to-run parallel code as output. Its parallelization process targets those blocks of code that are prone to be parallelized (i.e., loops) and uses the TM abstraction to mark them. Such TM-style transactions are then automatically instrumented by us to make the parallel execution correct (i.e., equivalent to the execution of the original serial application) even in the presence of data conflicts (e.g., the case of two iterations of one loop activated in parallel and that modify the same part of a shared data structure).

Despite the high-level goal, without fine-grain optimizations and innovations, deploying the above idea leads the application performance to be slower (often much slower [39, 52]) than the sequential, non-instrumented execution. As an example of that, a blind parallelization of a loop would mean wrapping the whole body of the loop within a transaction. By doing so, we either generate an excessive amount of conflicts on those variables that depend on the actual iteration count, or the level of instrumentation produced to guarantee a correct parallel execution becomes (fruitlessly) high.

In addition: variables that have never been modified within the loop may be uselessly transactionally accessed; the transaction commit order should be the same as the completion order of the iterations if they would have executed sequentially[47]; and aborts could be costly as it involves retrying the whole transaction including local processing work. The combination of these factors nullifies any possible gain due to parallelization, thus letting the application just pay the overhead of the transactional instrumentation and, as a consequence, providing performance slower than the sequential execution. Lerna does not suffer from the above issues. 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 dynamic (at runtime) tuning.

We evaluated Lerna's performance using a set of 13 applications including micro-benchmarks from the RSTM [2] framework, STAMP [10], a suite of applications designed for evaluating in-memory concurrency controls, and a subset of the PARSEC [9] benchmark. The reason we selected them is because they provide (except for PARSEC) a performance upper-bound for Lerna. In fact, they are released with a version that provides synchronization by using manually defined transactions. This way, besides the speedup over the sequential implementation, we can show the performance of Lerna against the same application with an efficient, hand-crafted solution. 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).

Lerna has been designed to be a framework with pluggable components. We provide an interface for integrating different TM algorithms. That way, TM designers can focus on developing specific algorithms and can rely on Lerna for integrating their solution, transparently. Lerna's main contribution is on the design and development of a unique tool 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, and with performance from 2× to 21× faster than the sequential code, and only 1.6× slower than the manual hand-crafted unordered parallel version.

2. Related Work

Automatic parallelization has been studied in the past. Papers in [15, 21] overview some of the most important contributions.

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 [8, 44, 45] to get the best of the two techniques.

Parallelization using thread-level speculation (TLS) has been extensively studied using both hardware [14, 27, 33, 54] and software [13, 18, 37, 38, 46]. It was originally proposed by Rauchwerger *et. al.* [46] for identifying and parallelizing loops with independent data access – primarily arrays. The common characteristics of TLS implementations are: 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 (e.g., the inner-most loop).

Regarding code parallelization and TM, Edler von Koch *et. al.* [22] 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 fully exploitation of all cores. Besides, relying only on runtime supports for parallelization introduces a non-negligible overhead to the framework. Similarly, DeVuyst *et. al.* [19] uses HTM to optimistically run parallel sections, which are detected using special hardware.

STMLite [38], shares the same sweet-spot we aim for, 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.

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 improves the parallel processing itself with specific pattern-dependent (e.g., loop) optimization.

The study at [57] classified applications into: sequential, optimistically parallel, or truly parallel, and classify 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 [7, 10, 12, 42] features as the size of read and write sets, dependency density, and the 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 that those approaches include [30, 37, 41, 48]. Lerna does not require the programmer and offers innovations effective when the application exposes data dependencies with non-partitionable access patterns.

HydraVM has been presented in [51]. It is an initial concept of a virtual machine that exploits transactions to parallelize loops. Unlike Lerna: HydraVM reconstructs code at runtime through recompilation and reloading class definition, and it is obligated to run the application through the virtual machine. The profiling phase relies on establishing a relation between basic blocks and their accessed memory addresses, which limits its usage to small size applications. Lerna is a complete system, which overcome all the above limitations and embeds innovations to provide high performance.

3. Lerna

3.1 General Architecture and Workflow

Lerna splits the code of loops into parallel *jobs*. This operation is made aiming to maximize the independence between jobs, which makes it one of the most important steps to enable high performance (clearly the presence of more conflicts leads to less parallelism and thus poor performance). 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 in separate threads as memory transactions, and a 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 main memory.

We define a *successful execution* of an invoked job as an execution that satisfies the following two conditions: *1)* it is reachable by future executions of the program (e.g., the case of three iterations of the same loop activated in parallel and while the third executes speculatively, it becomes not reachable anymore because the second performs a *brake* instruction); and *2)* it does not cause a memory conflict with any other job having an older chronological order. As we will detail in Section 3.4, any execution of a parallel

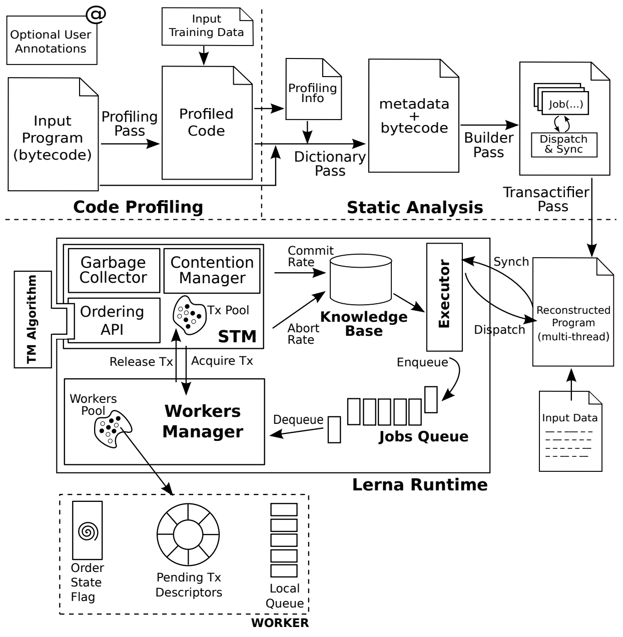


Figure 1: Lerna's Architecture and Workflow

program produced after our transformations is made of a sequence of jobs committed after a successful execution.

Summarizing, *Lerna* is a container of:

- an *automated software tool* that performs a set of transformations and analysis steps (called *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: *1)* organizing the transactional execution of dispatched jobs so that the original program order (i.e., the chronological order) is preserved; *2)* selecting the most effective number of worker threads according to the actual deployment, and to the feedbacks collected from the online execution; *3)* scheduling jobs to threads based on threads' characteristics (e.g., stack size, priority); and *4)* 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 any 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 following three steps in this order: *Code Profiling*, *Static Analysis*, and *Runtime*.

In the first step, our software tool executes the original (sequential) application by activating our own profiler that collects some important parameters (e.g., execution frequencies) 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 evolves by following the below passes:

Dictionary Pass. It scans the input program to provide a list of the *accessible* (i.e., which is not either a system-call or a native-library call) functions of the byte-code (or the *bitcode* as named by LLVM) that we can analyze to determine how to transform. By default, any call to an external function is flagged as *unsafe*. This information is important because transactions cannot contain unsafe calls as they may include irrevocable (i.e., which cannot be further aborted) operations, 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 (i.e., transaction's begins and ends).

Transactifier Pass. It applies the alias analysis [16] (i.e., it detects if multiple references point to the same memory location) and some memory dependency techniques (e.g., given a memory operation, extracts the preceding memory operations that depend on it) 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 or allocate.

Once the Static Analysis is complete, the reconstructed version of the program is linked to the application through our runtime library, which is mainly composed of the following 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 to dispatch, 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.
- *TM*. It provides the handlers for transactional accesses (read and write) 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). Also, 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. This information is fundamental for letting the refactoring process focus on the real parts of the code that are fruitful to parallelize (e.g., 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 basic block is a sequence of non-branching instructions that ends either with a branch instruction (conditional or non-conditional) or a return. Given that, any program can be represented as a graph in which nodes are basic blocks and edges reproduce the program control.

In this phase, 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 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

In the following, we illustrate in detail the transformation from sequential code to parallel made during the static analysis phase. The LLVM intermediate representation (i.e., the byte-code) 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 [43].

3.3.1 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 (e.g., it does not perform input/output operations); *ii)* it does not affect the state of the program; and *iii)* it is thread safe. A common example of safe calls are stateless random generators, or mathematical basic functions such as trigonometric functions.

3.3.2 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 the parallelization cannot be performed on that basic block.

In this work we focus on loops as the most appropriate blocks of code for being parallelized. However, our design is applicable (unless stated otherwise) for any independent sets of basic blocks. 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. We apply the loop simplification to put the loop into its simplest form.

Induction Variable analysis. An *induction* variable is a variable within a loop whose value changes by a fixed amount every iteration (i.e., the loop index) or is a linear function of another induction variable. Affine (linear) memory accesses are commonly used in loops (e.g., array accesses, 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* is a transformation to rewrite any induction variable in the loop as a closed form (function) of its index. It starts by detecting the candidate induction variables, then it sorts them topologically and creates a closed symbolic form for each of them. Finally, it substitutes their occurrences with the corresponding symbolic form.

As a part of our transformation, a loop is simplified, and its induction variable (i.e., the index) 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, any induction variables introduce dependencies between iterations, which are not desirable to 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. This approach avoids unnecessary conflicts on the induction variables.

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

The loop body is replaced by two basic blocks: *Dispatcher* and *Sync*. In the *Dispatcher*, we prepare the arguments for the synthetic method, calculate the value of the loop index and invoke an API of our library ($lerna\_dispatch$), providing it with the address of the synthetic method and the list of the just-computed arguments. Each call to $lerna\_dispatch$ adds a job to our internal jobs queue, but it does not start the actual execution of the job. The *Dispatcher* keeps dispatching jobs until our API decides to stop. When it 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.

Summarizing, the Builder Pass turns the execution model into the job-driven model, which can exploit parallelism. This strategy abstracts the processing from the source code.

3.3.3 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 common that memory reads outnumbers 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 for preserving the correctness of the transaction, which iterates over the read-set, is the primary source of TM's overhead. Our transactifier pass eliminates unnecessary transactional reads, thus significantly improving the performance of the transaction execution due to the following reasons:

- direct memory read is even three times faster than transactional read [11, 52]. In fact, reading an address transactionally requires: *1)* checking if the address has been already written before (e.g., check the write-set); *2)* adding the address to the read-set; and *3)* returning the value to the caller.
- the size of the read-set is limited, thus extending it requires copying entries into a larger read-set, which is costly. Keeping the read-set small reduces the resizing overhead.
- read-set validation is mandatory during the commit. The smaller the read-set, the faster the commit operation.

In our model, concurrent transactions can be described as "symmetric", which means that the code executed in all active transactions is the same. That is because each transaction executes one or more iterations of the same loop. We take advantage of this characteristic by: 1) relaxing the need to support TM strong atomicity [4], and 2) reducing the number of transactional calls as follows.

Clearly, local addresses defined within the scope of the loop are not required to be accessed transactionally. On the other hand, 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 technique is specific for parallelizing loops and cannot be applied to the normal transaction processing where concurrent transactions do not necessarily execute the same code as for the symmetric transactions.

In some situations, the induction variable substitution cannot produce a closed form (function) 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 loop completes the whole execution, then it is eligible for the *Read-Modify-Write (RMW)* [49] optimization. Using RMW, the increments (or decrements) are delayed till 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 other 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 dependencies between variables, which can be leveraged to reduce transactional calls, or the detection of dependent loop iterations, which is useful to exclude them from the parallelization.

Finally, to avoid unnecessary overhead in the presence of single-threaded computation or a single job executed at a time, we create another non-transactional version of the synthetic method. This way we provide a fast version of the code without unnecessary transactional accesses.

3.4 Transactional Execution

The atomicity of transactions is mandatory as it guarantees the consistency of the code, even after its refactoring to run in parallel. However, if no additional care is taken, transactions run and commit independently of each other, and that could revert the chronological order of the program, which must be preserved to avoid incorrect executions.

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 lead to transactions corresponding to unreachable iterations (e.g., a break condition that changes the execution flow), and transactions executing iterations with lower indexes may read future values from committed transactions. Although these scenarios are admissible under generic concurrency controls (where the order of transactions is not enforced), it clearly violates the logic of the 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. Transactions have five states: *idle*, *active*, *completed*, *committed*, and *aborted*. Initially a transaction is idle because it is still in the transactional pool waiting to be attached to a job to dispatch. Each transaction has an *age* identifier that defines its chronological order in the program. 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. A transaction in this state still holds its lock(s). Finally, the transaction is committed when it becomes reachable from its predecessor transaction. Decoupling *completed* and *committed* states, permits threads to process next transactions.

3.4.1 Preserving Commit Order

Lerna is decoupled from the actual TM implementation deployed, but it requires to enhance the TM design itself for enforcing a specific commit order (i.e., earlier transactions must not observe the changes made by later transactions). To allow such a decoupling, we identified the following requirements needed by a TM implementation 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 permits transactions to commit, namely the *Commit Manager* (CM). It is not necessary having a dedicated CM 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 CM 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, or the timestamp validity decreases), therefore this speculation must be limited by a certain (tunable) threshold. Upon a successful commit, the CM 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 CM [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 CM execution time.

Age-based Contention Management (CM). Algorithms with eager conflict detection (i.e., at encounter time) should favor lower age transactions, while algorithms that use lazy conflict detection (i.e., at commit time) should employ an aggressive CM that favors the transaction that is committing using the single committer.

Lerna currently integrates four TM implementations with different designs: NOrec [17], which executed commit phases serially without requiring any ownership record; TL2 [20], which allows parallel commit phases but at the cost of maintaining an external data structure for storing meta-data associated with the transactional objects; *UndoLog* [3] 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 designed for enabling code 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. The characteristic of this algorithm is that it permits a single committing writer at a time, which matches the need of Lerna's concurrency control: having a single committer. Our modified version of NOrec decides the next transaction to commit according to the chronological order (i.e., age).

3.4.2 Irrevocable 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 an *irrevocable transaction*. Also, a transaction is *irrevocable* if: *i)* it is the first, thus it 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

The Adaptive Optimization System (AOS) [6] is a general virtual machine architecture that allows 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. The buffer size is crucial as it controls the lifetime of transactions. A larger buffer allows the worker to execute more transactions, but it increases also the transaction life time, and consequently the conflict probability.

For the non-dedicated CM, the ordering is managed by a worker-local flag called *state flag*. This flag is read by the current worker, but is modified by its predecessor worker. Initially, only the first worker (executing the first job) has its state flag set, while others have their flag cleared. 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 use of cache locality as threads operate on their own transactions and access thread-local data structures, which also reduces bus contention. Regarding the dedicated CM, we rely on the ordering design of STMLite [38].

4.1 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 size may cause many aborts due to unreachable jobs, while having small batches increases the number of iterations between *dispatcher* and the *executor*, and, as a consequence, the number of pauses to perform due to Sync. The current implementation uses an exponentially increasing batch size. Initially, we dispatch a single job, which covers the common set of loops with zero iterations; if loops are longer, then we increase the number of dispatched jobs exponentially until reaching a threshold. Once a loop is entirely 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.

4.2 Jobs Tiling and Partitioning

As explained in Section 3.3.2, the transformed program dispatches iterations as jobs, and our runtime runs jobs as transactions. Here we discuss an optimization, named *jobs tiling*, that allows the association of multiple jobs to a single transaction. Increasing jobs per transaction reduces the total number of commit operations. Also, it allows assigning enough computation power to the threads, which outweigh the cost of transactional setup. Nevertheless, tiling is a double-edged sword. Increasing tiles increases the size of read and write sets which can 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* [5], a loop is rewritten at compile time as a repeated sequence of its iteration code. Lerna employs the static unrolling and the runtime tiling according to the loop size.

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.

4.3 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), and the underlying hardware utilization.

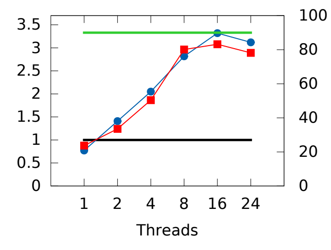
In practice, at the end of the execution of a batch of jobs, we calculate the throughput and we record it into the *knowledge base*, along with the commit rate, tiles and the number of workers involved. We apply a greedy strategy to find an effective number of workers by matching with the obtained throughput. In some situations (e.g., high contention or very small transactions) it is better to use a single worker. For that reason, if our heuristic decides that, then we use the non-transactional version (as a fast path) of the synthetic method to avoid the unnecessary transaction overhead.

5. Evaluation

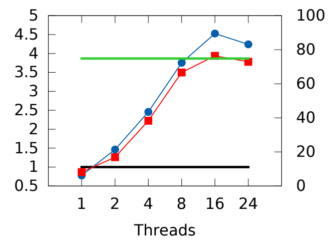
In this section we evaluate Lerna and measure the effect of the key performance parameters (e.g., job size, worker count, tiling) on the overall performance. Our evaluation involves a total of 13 applications grouped into micro- and macro-benchmarks.

We compare the speedup of Lerna over the (original) sequential and the manual, optimized transactional version of the code (if available). Note that the latter is not coded by us; it is released along with the application itself, and is made by knowing the details of the application logic, thus it can leverage optimizations, such as the out-of-order commit, that cannot be caught by Lerna automatically. The results with the manual transactional version represent a practical upper bound for Lerna, which provides a sense of how good is the automatic transformation of the code without any programmer's hint. As a result, 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.

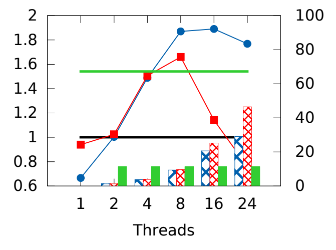
The testbed consists of an AMD multicore machine equipped with 2 Opteron 6168 processors, each with 12-cores running at 1.9GHz of clock speed. The total memory available is 12GB and the cache sizes are 512KB for the L2 and 12MB for the L3. On



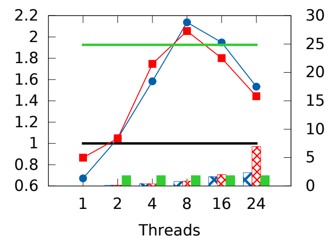
(a) ReadNWrite1 - Long



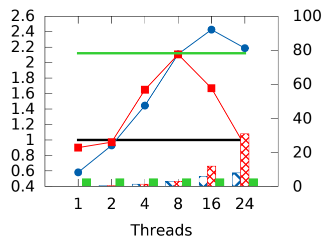
(b) ReadNWrite1 - Short



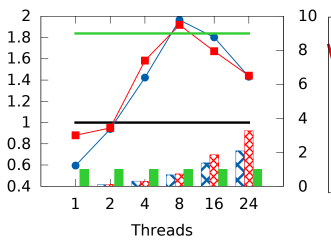
(c) ReadWriteN - Long



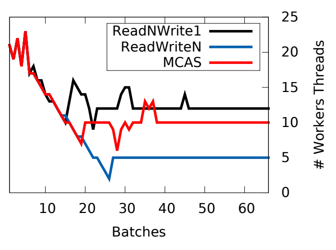
(d) ReadWriteN - Short



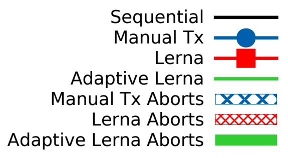
(e) MCAS - Long



(f) MCAS - Short



(g) Adaptive workers selection



(h)

Figure 2: Performance with micro-benchmarks. The left y-axis shows the speedup, while the right y-axis is the % of abort.

this machine, the overall refactoring process, from profiling to the generation of the binary, takes ∼10s for simple applications and ∼40s for the more complex ones.

5.1 Micro-benchmarks

Here we consider the RSTM micro-benchmarks [2] to evaluate the effect of different workload characteristics Figure 2 reports the speedup over the sequential code by varying the number of threads used. We show the performance of two versions of Lerna: one adaptive, where the most effective number of workers is selected at runtime (thus its performance do not depend on the number of threads reported in the x-axis), 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 integrated in Lerna (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 when we report the performance by fixing the number of threads. 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 and 2b), transactions read 1k locations and write 1 location. Given that, the transaction write-set is very small, 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 Tx version; 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 aborts. Both Lerna and manual Tx incur performance degradation at high numbers of threads due to the high abort rate (up to 50%). In addition, for Lerna the commit phase of long transactions forces some (ready to commit) workers to wait for their predecessor, thus degrading the overall performance. In such scenarios, 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. Similarly 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 Tx, Lerna performs better at single thread because it uses the fast path version of the jobs (non-transactional) to avoid needless 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 (7), then it picks the best according to the actual throughput. Changing the worker count shifts the window so that the most effective setting can be found.

5.2 The STAMP Benchmark

STAMP [10] is a benchmark covering different domains (Yada and Bayes have been excluded because they expose non-deterministic behaviors). Figure 3 shows the speedup of Lerna's transformed code over the sequential code, and against the manual transactional version of the applications, which exploits unordered commits.

Kmeans, a clustering algorithm, iterates over a set of points and associate 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 (given the limited lock table or signatures) 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 eliminates duplicates, which requires synchronization. Lerna has 16-19× speedup over sequential. Genome conducts a many read-only transactions (Hashtable *exists* operations); a friendly behavior for all implemented algorithms. TL2 is just 10% slower than the manual competitor. *Vacation* is a travel reservation system using an in-memory

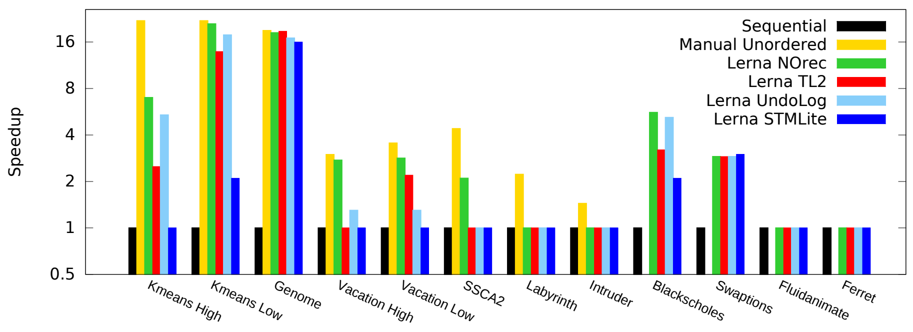


Figure 3: STAMP & PARSEC Benchmarks Speedup. (The y-axis is log-scale)

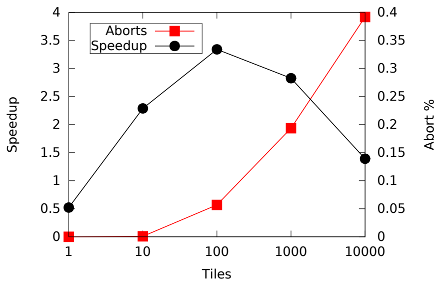


Figure 4: Effect of Tiling on abort and speedup using 8 workers and Genome.

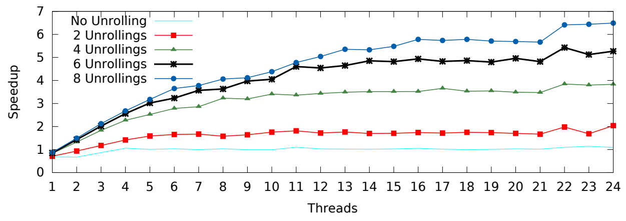


Figure 5: Effect of Unrolling on speedup using Black-Scholes.

database. The workload consists of clients reservation. This application emulated an OLTP workload. Lerna improves the performance by 2.8× faster than the sequential system, and it is very close to the manual. *SSCA2* is a multi-graph kernel that is commonly used in domains such as biology and security. The core of the kernel uses a shared graph structure that is 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 worth noting that NOrec is the only algorithm that manage to achieve speedup because it tolerates high contention and isn't 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 4.2, 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.

5.3 The PARSEC Benchmark

PARSEC [9] is a benchmark suite for shared memory chip-multiprocessors architectures.

The Black-Scholes equation [31] is a differential equation that describes how, under certain assumptions, the value of an option changes as the price of the underlying asset changes. This benchmark calculates Black-Scholes equation for input values. The iterations are relatively short, which causes producing a lot of jobs in Lerna's transformed code. However, jobs can be tiled (see Section 4.2). The top speedup achieved here is 5.6×. Figure 5 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] framework. 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* [40] 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 conflicts with its preceding transactions.

Ferret is a toolkit which is 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 Discussion

As confirmed by our evaluation study, there are scenarios where, without the programmer handing the application's logic on the refactoring process, Lerna encounters some hindrance (e.g., single point of contention) that cannot be automatically broken due to the lack of "semantics" knowledge. Relevant examples of that include complex data structure operations, and centric global shared variables updates (i.e., shared variables updated by all loop iterations).

In addition, Lerna becomes less effective when: there are 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, thus increasing the chance of invalidating most transactions from the very beginning; and the workload is heavily unbalanced across iterations. Anyway, in all the above cases, at worst, Lerna's code performs as the original.

6. Conclusion

In this paper we presented Lerna, a completely automated system that combines a software tool and a runtime library to extract parallelism from sequential applications with data dependencies, efficiently and without programmer interventions. Lerna overcomes the pessimism of the static analysis of the code by exploiting speculation. Lerna also represents a framework and testbed for the research community to develop and evaluate TM algorithms for code parallelization.

7. Acknowledgments

This work is partially supported by Air Force Office of Scientific Research (AFOSR) under grant FA9550-14-1-0187.

References

[1] Intel Parallel Studio. https://software.intel.com/en-us/intel-parallel-studio-xe.

[2] RSTM: The University of Rochester STM. http://www.cs.rochester.edu/research/synchronization/rstm/.

[3] TinySTM: A time-based STM. http://tinystm.org/tinystm.

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

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

[6] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the jalapeno jvm. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '00, pages 47–65, New York, NY, USA, 2000. ACM.

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

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

[9] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[10] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[11] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, Jun 2007.

[12] B Chan. The umt benchmark code. *Lawrence Livermore National Laboratory, Livermore, CA*, 2002.

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

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

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

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

[17] Luke Dalessandro, Michael F Spear, and Michael L Scott. Norec: streamlining stm by abolishing ownership records. In *ACM Sigplan Notices*, volume 45, pages 67–78. ACM, 2010.

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

[19] Matthew DeVuyst, Dean M Tullsen, and Seon Wook Kim. 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*, pages 127–136. ACM, 2011.

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

[21] Nicholas DiPasquale, T Way, and V Gehlot. Comparative survey of approaches to automatic parallelization. *MASPLAS05*, 2005.

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

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

[24] MA Gonzalez-Mesa, Eladio Gutierrez, Emilio L Zapata, and Oscar Plata. Effective transactional memory execution management for improved concurrency. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(3):24, 2014.

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

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

[27] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. *SIGOPS Oper. Syst. Rev.*, 32(5):58–69, October 1998.

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

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

[30] Shan Shan Huang, Amir Hormati, David F. Bacon, and Rodric M. Rabbah. Liquid metal: Object-oriented programming across the hardware/software boundary. In Jan Vitek, editor, *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, volume 5142 of *Lecture Notes in Computer Science*, pages 76–103. Springer, 2008.

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

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

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

[34] Leslie Lamport. The parallel execution of do loops. *Communications of the ACM*, 17(2):83–93, 1974.

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

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

[37] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. Posh: a tls compiler that exploits program

structure. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 158–167. ACM, 2006.

[38] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. 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, pages 166–176, New York, NY, USA, 2009. ACM.

[39] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 69–80. ACM, 2007.

[40] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '03, pages 154–159, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[41] Stefan C. Müller, Gustavo Alonso, Adam Amara, and André Csillaghy. Pydron: Semi-automatic parallelization for multi-core and the cloud. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 645–659, Broomfield, CO, October 2014. USENIX Association.

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

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

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

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

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

[47] Yoav Raz. The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource managers using atomic commitment. In *VLDB*, volume 92, pages 292–312, 1992.

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

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

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

[51] Mohamed M. Saad, Mohamed Mohamedin, and Binoy Ravindran. Hydravm: Extracting parallelism from legacy sequential code using STM. In Hans-Juergen Boehm and Luis Ceze, editors, *4th USENIX Workshop on Hot Topics in Parallelism, HotPar'12, Berkeley, CA, USA, June 7-8, 2012*. USENIX Association, 2012.

[52] Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural support for software transactional memory. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 185–196, Washington, DC, USA, 2006. IEEE Computer Society.

[53] Joel H Saltz, Ravi Mirchandaney, and K Crowley. The preprocessed doacross loop. In *ICPP (2)*, pages 174–179, 1991.

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

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

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

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