Extracting Parallelism from Legacy Sequential Code Using Software Transactional Memory

Mohamed M. Saad

Preliminary Examination Proposal submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Binoy Ravindran, Chair
Anil Kumar S. Vullikanti
Paul E. Plassmann
Robert P. Broadwater
Roberto Palmieri
Sedki Mohamed Riad

May 5, 2015
Blacksburg, Virginia

Keywords: Transaction Memory, Software Transaction Memory (STM), Automatic Parallelization, Low-Level Virtual Machine, Optimistic Concurrency, Speculative Execution, Legacy Systems

Extracting Parallelism from Legacy Sequential Code Using Software Transactional Memory

Mohamed M. Saad

(ABSTRACT)

Increasing the number of processors has become the mainstream for the modern chip design approaches. On the other hand, most applications are designed or written for single core processors; so they do not benefit from the underlying computation resources. Moreover, there exists a large base of legacy software which requires an immense effort and cost of rewriting and re-engineering.

Transactional memory (TM) has emerged as a powerful concurrency control abstraction. TM simplifies parallel programming to the level of coarse-grained locking while achieving fine-grained locking performance. In this dissertation, we exploit TM as an optimistic execution approach for transforming a sequential application into parallel. We design and implement two frameworks that support automatic parallelization: Lerna and HydraVM.

HydraVM is a virtual machine that automatically extracts parallelism from legacy sequential code (at the bytecode level) through a set of techniques including code profiling, data dependency analysis, and execution analysis. HydraVM is built by extending the Jikes RVM and modifying its baseline compiler. Correctness of the program is preserved through exploiting software transactional memory (STM) to manage concurrent and out-of-order memory accesses.

Lerna is a compiler framework that automatically and transparently detects and extracts parallelism from sequential code through a set of techniques including code profiling, 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 makes Lerna very effective for sequential applications with data sharing. Additionally, we introduce the general conditions for embedding any transactional memory algorithm into our system.

While prior research shows that transactions must commit in order to preserve program semantics, placing the ordering enforces scalability constraints at large number of cores. Our first major post-preliminary research goal is to eliminate the need for ordering transactions without affecting program consistency. We propose building a cooperation mechanism in which transactions can forward some changes safely. This approach eliminates false conflicts and increases concurrency opportunities. As our second contribution, we propose transactional checkpointing as a technique for reducing the wasted processing time in abort and code retrial. With checkpointing, a transaction can be partially aborted and only re-execute the invalid portion.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the last decade, parallelism has gained a lot of attention due to the physical constraints which prevent increasing processor operating frequency. The runtime of a program is measured by the time required to execute its instructions. Decreasing the runtime requires reducing the execution time of a single instruction, which implies increasing the operating frequency. From the mid of 1980s until the mid of 2000s[1] this approach, namely *frequency scaling*, was the dominant force in commodity processor performance to improve programs runtime. However, operating frequency is proportional to the power consumptions and consequently heat generation. This obstacles put an end to the era of frequency scaling and force the chip designers to find an alternative approach for maintain the growth of applications performance.

Gordon E. Moore put an empirical observation that the number of transistors in a dense integrated circuit has doubled approximately every two years. With the power consumption issues, these additional transistors are moved to add extra hardware that made multicore processors become the norm for microarchitecture chip design. Wulf *et. al.* [132] report that the rate of improvement in microprocessor speed exceeds the rate of improvement in memory speed, which constraints the performance of any program running on a single processing unit.

These combined factors (i.e., power consumption, memory wall and the availability of extra hardware resources) made *parallelism*, which primarily was employed for long time in high-performance computing, appears as an appealing alternative.

Parallelism is the execution of a sequential application simultaneously on multiple computation resources. The application is divided into multiple sub-tasks that can run in parallel. The communication between sub-tasks defines the parallelism granularity. An application is *embarrassingly parallel* when the communications between its sub-tasks are rare while an application with a lot of sub-tasks communications exhibits fine-grained parallelism. The

[1] At year 2004, because of the increase in processor power consumption Intel announced the cancellation of its Tejas and Jayhawk processors, the successors processors families for Pentium 4 and Xeon respectively.

maximum possible speedup of a single program as a result of parallelization is known as Amdahl's law. This law defines the relation between speedup and the time needed for the sequential fraction of the program. On the other hand, the vast majority of the applications and algorithms are designed or written for single core processors (often intentionally designed to be sequential to reduce development costs, while exploiting Moore's law of single-core chips).

1.1 Motivation

Many organizations with enterprise-class legacy software are increasingly faced with a hardware technology refresh challenge due to the ubiquity of chip multiprocessor (CMP) hardware. This problem is apparent when legacy codebases run into several million LOC and are not concurrent. Manual exposition of concurrency is largely non-scalable for such codebases due to the significant difficulty in exposing concurrency and ensuring the correctness of the converted code. In some instances, sources are not available due to proprietary reasons, intellectual property issues (of integrated third-party software), and organizational boundaries. Additionally, adapting these programs to exploit hardware parallelism requires a (possibly) massive amount of software rewriting operated by skilled programmers with knowledge and experience of parallel programming. They must take care of both high-level aspects, such as data sharing, race conditions, and parallel sections detection; and low-level hardware features, such as thread communication, locality, caching, and scheduling. This motivates techniques and tools for *automated concurrency refactoring*, or *automatic parallelization*.

Automatic parallelization simplifies the life of programmers, especially those not extensively exposed to the nightmare of developing efficient concurrent applications, and to allow a growing number of (even legacy) systems to benefit from the nowadays available cheap hardware parallelism. Automatic parallelization is targeting the programming transparency; The application is coded as sequential and the specific methodology used for handling concurrency is hidden to the programmer.

1.2 Current Research Contributions

In this thesis, we bridge the gap between the parallelism of existing multicore architectures and the sequential design of most (including existing) applications by presenting

- *HydraVM*, a virtual machine, built by extending the Jikes RVM and modifying its baseline compiler. It reconstructs programs at the bytecode level to enable more threads to execute in parallel. HydraVM is equipped with *ByteTM*, which is a Software Transaction Memory (STM) implementation at the virtual machine level.

- *Lerna*, a completely automated compiler framework that extracts parallelism from existing code, at the intermediate representation level, and executes it adaptively and efficiently on commodity multi-processor chips without any programmer intervention. Lerna is based on *LLVM*, which covers a wide set of programming languages with supports the generation of native executable as output.

Both implementations require no programmer intervention, they are completely automated, and do not need to expose source code; unlike previous approaches for parallelization that require programmer to identify parallel sections or additional information to help the parallelization process.

The common technique used in our two implementations is the use of Transactional Memory (TM) as an optimistic execution approach for transforming a sequential application into parallel "blindly", meaning without external interventions.

Transactional Memory (TM) introduces a simple parallel programming model while achieving performance close to fine-grained locking. With HydraVM, user benefits from TM seamlessly by running sequential programs and enjoys a safe parallel execution. Under-the-hood, HydraVM handles the problems of detecting parallel code snippets, concurrent memory access, synchronizations and resources utilization are handled.

From the experience with HydraVM, we learnt about parallel patterns and bottlenecks in the programs. Firstly, relying only on dynamic analysis introduces an overhead that can be easily avoided through pre-execution static analysis phase. Second, adaptive design should not be limited to the architecture (e.g., runtime recompilation), but it could be extended to: selecting best number of workers, assignment of parallel code to threads, and the depth of speculative execution. Last, except recursive calls, most of the detected parallel traces at HydraVM was primarily loops. Keeping these goals in sight, we designed Lerna as a compiler framework. With Lerna, the sequential code is transformed into parallel with best-efforts TM support to guarantee safety and to preserve ordering. Lerna produces a native executable, yet adaptive; thanks to Lerna runtime library that orchestrates and monitor the parallel execution. The programmer can interact with Lerna to aid the static analysis for the sake of producing a less overhead program.

1.2.1 Transactional Parallelization

Transactions were originally proposed by database management systems (DBMS) to guarantee atomicity, consistency, data integrity, and durability of operations manipulating shared data. This synchronization primitive has been recently ported from DBMS to concurrent applications (by relaxing durability), providing a concrete alternative to the manual implementation of synchronization using basic primitives such as locks. This new multi-threading programming model has been named Transactional Memory (TM) [67].

Transactional memory (TM) has emerged as a powerful concurrency control abstraction [59, 79] that permits developers to define critical sections as simple *atomic* blocks, which are internally managed by the TM itself. It allows the programmer to access shared memory objects without providing the mutual exclusion by hand, which inherently overcomes the drawbacks of locks. As a result, with TM the programmer still writes concurrent code using threads, but the code is now organized so that reads and writes to shared memory objects are encapsulated into atomic sections. Each atomic section is a *transaction*, in which the enclosed reads and writes appear to take effect instantaneously. Transactions speculatively execute, while logging changes made to objects–e.g., using an undo-log or a write-buffer. When two transactions conflict (e.g., read/write, write/write), one of them is aborted and the other is committed, yielding (the illusion of) atomicity. Aborted transactions are re-started, after rolling-back the changes–e.g., undoing object changes using the undo-log (eager), or discarding the write buffers (lazy).

Besides a simple programming model, TM provides performance comparable to highly concurrent, fine-grained locking implementations [43, 22], and allows composability [60], which enables multiples nested atomic blocks to be executed as a single all-or-nothing transaction. Multiprocessor TM has been proposed in hardware (HTM), in software (STM), and in hardware/software combination. TM's adoption is growing in the last years, specially after the integration with the popular *GCC* compiler (from the release 4.7), and due to the integration into the embedded cache-coherence protocol of commodity processors, such as Intel [72] and IBM [24], which naively allows transactions to execute directly on the hardware.

Given its easy-to-use abstraction, TM would seem the missing building block enabling the (transparent) parallelization of sequential code by automatically injecting atomic blocks around parallel sections. Unfortunately, such high-level abstraction comes with a price in terms of performance cost, which easily leads to a parallel code slower than its sequential version (e.g., in [26, 116] a performance degradation of between 3 to 7 times has been shown).

Motivated by TM's advantages, we designed and implemented: 1) a java virtual machine, named *HydraVM*, that detects eligible parallel hot-spot portion of the code at runtime, and reconstruct the program producing a parallel version of the code, and 2) *Lerna*, a language-independent compiler that transform the sequential application into an adaptive parallel version that utilities underlying computation resources based on the execution profiles. Each of these implementations is aided by novel techniques to overcome TM overhead, and to support efficient parallel execution. Additionally, we introduce the general conditions for preserving chronological ordering of the sequential program and embedding any transactional memory algorithm into our system.

1.2.2 HydraVM

This is a java virtual machine based on Jikes RVM. The user runs its sequential program (java classes) using the virtual machine, and internally we instrument the bytecode at runtime to profile the execution paths. The profiling detects which places of the code are suitable for parallelization and does not have data dependency. Next, we reconstruct the bytecode by extracting portions of the code to run as separate threads. Each thread runs as a transaction, which means it operates on its private copy of memory. Conflicting transactions are aborted and retried while successful transactions commit its changes at the chronological time of the code it executes.

HydraVM inherits the adaptive design of Jikes RVM. The profiling and code reconstruction continue while the program is running. We monitor the performance and abort rate of transformed code and use this information to repeatedly reconstruct new versions of the code. For example, assume a loop is parallelized and each iteration runs in a separate transaction. When every three consecutive iterations of loop conflict with each other, then a better reconstruction is to combine them and makes the transaction runs three iterations instead one. It worth noting that the reconstruction process occurs at runtime by reloading the classes definition.

Finally, we developed ByteSTM a software transactional memory implementation at the bytecode level. With ByteSTM we have access to low-level memory (e.g., registers, thread stack), so we can create a memory *signature* for memory accessed by the current transaction. Comparing concurrent transaction signatures allows us to quickly detect conflicting transaction.

1.2.3 Lerna

Leran is a compiler that runs on the intermediate representation level, which makes it independent of the source code and programming language used. The generated code is a task-based multi-threaded version of the input code.

Unlink HydraVM, Lerna employs static analysis techniques, such as alias analysis and memory dependency analysis, to reduce the overhead of transactional execution, and here we focus on parallelizing loops. Lerna is not limited to a specific TM implementation, and the integration of any TM algorithm can be done through a well-defined APIs. Additionally, in Lerna the mapping between extracted tasks and transactions is not one-to-one. A transaction can run multiple tasks (tiling), or task can have multiple transactions (partitioning).

Lerna and HydraVM share the idea of exploiting transactional memory, profiling and producing adaptive code. However, as Lerna supports the generation of native executable as output we can't rely on an underlying layer (the virtual machine in HydraVM) to support the adaptive execution. Instead, we link the program with Lerna runtime library that is able

to monitor and modify the key performance parameters of the our executor module such as: number of workers threads, the mapping between transactions and tasks, and executing in transaction mode or go sequentially.

1.3 Proposed Post-Prelim Work

With our experience with HydraVM and Lerna, we learnt that preserving program order hampers scalability. A thread that completes its execution must either stalls waiting its correct chronological order in the program or proceeds executing more transactions, and consequently increases the lifetime of these pending transactions and makes them subject to conflict with other transactions. Additionally, transactional reads and stores are sandboxed. This prevents any possible *cooperation* between transactions that could still produce a correct program results. For example, in DOACROSS loops [36] iterations are data or control dependent, however, they can run in parallel with exchanging some data between them. We propose a novel technique for cooperation between concurrent transactions and execute them out-of-order. With this technique, transactions can expose their changes to other transactions without waiting for their chronological commit times. Nevertheless, transactions with the earlier chronological order can abort completed transactions (and cascade abort to any other affected transactions) whenever a conflict exists.

Aborting a transaction is a costly operation, not only because the rollback cost, but retrying the code execution doubles the cost and wastes precious processing cycles. Additionally, transactions may do a lot of local processing work that does not use any protected shared variables. We propose *transaction checkpointing* as a technique that creates multiple checkpoints at some execution points. Using checkpoints, a transaction saves the state of the work done at certain times after doing a considerable amount of work. Later, if the transaction experienced a conflict due to an inconsistent read or because writing a value that should be read by another transaction executing an earlier chronological order code, then it can return to the last state wherein the execution was valid (i.e., before doing the invalid memory operation). Checkpointing introduces an overhead for creating the checkpoints (e.g., saving the current state), and for restoring the state (upon abort). Unless the work done by the transaction is large enough to outweigh this overhead, this technique is not recommended. Also, checkpointing should be employed when the abort rate exceeds a predefined threshold. A good example that would get use of this technique is when a transaction finishes its processing by updating a shared data structure with a single point of insertion (e.g., linked list, queue, stack); Two concurrent transactions will conflict when trying to use this shared data structure. With checkpointing, the aborted transaction can jump back till before the last valid step and retry the shared access step.

1.4 Outline

This thesis is organized as follows. In Chapter 2 we survey the techniques for solving the parallelization problem. We overview past and related efforts in Chapter 3. In Chapter 4 and 5, we detail our the architecture, transformation, optimization techniques, and the experimental evaluation for our two implementations: *HydraVM* and *Lerna*. Finally, we conclude and present post-preliminary work in Chapter 6.

Chapter 2

Background

Parallel computations can be done on multiple levels such as instructions, branch targets, loops, execution traces, and subroutines. Loops have gained a lot of interest as it is by nature a major source of parallelism, and usually contains most of processing.

Instruction Level Parallelization (ILP) is a measure of how many instructions that can run in parallel. ILP is application specific as it involves reordering the execution of instructions to run in parallel and utilize the underlying hardware. ILP can be implemented using software (compiler), or hardware (pipelining). Common techniques for supporting ILP are: *out-of-order execution*; where instructions execute in any order that does not violate data dependencies, *register renaming*; which is renaming instructions operands to avoid unnecessary reuse of registers, and *speculative execution*; where an instruction is executed before it should take place according to the serial control flow.

Since on average 20% of instructions are branches, branch prediction was extensively studied to optimize branch execution and run targets in parallel with the evaluation of branching condition. Branch predictor is usually implemented in hardware with different variants: Early implementations of SPARC and MIPS used a static prediction by always predicting that a conditional jump would not be taken, and execute the next instruction in parallel to evaluating the condition. Dynamic prediction is implemented through a state-machine that keeps the history of branches (per each branch, or globally). For example, Intel Pentium processor uses a four state-machine to predict branches. The state-machine can be locally (per branch instruction) as in Intel Pentium MMX, Pentium II, and Pentium III; or globally (shared history of all conditional jumps) as in AMD, Intel Pentium M, Core and Core 2.

Loops can be classified as sequential loops, parallel loops (DOALL), and loops of intermediate parallelism (DOACROSS) [36], see Figure 2.1. In DOALL loops, iterations are independent and can run in parallel as no data dependency exists between loops. DOACROSS loops exhibits inter-iterations data dependency. When data from lower index iterations used by iterations with higher index, lexically-backward, the processors executing higher index iter-

```
For i = 1 to n {
    Sum = Sum + A[i];
}
```

(a) Sequential Loop

```
For i = 1 to n {
    D[i] = A[i] * B[i] + c
}
```

(b) DOALL Loop

```
For i = 1 to n-1 {
    D[i] = A[i] + B[i]
    C[i] = D[i+1] * B[i]
}
```

(c) Lexically-forward DOACROSS Loop

```
For i = 2 to n {
    D[i] = A[i] + B[i]
    C[i] = D[i-1] * B[i]
}
```

(d) Lexically-backward DOACROSS Loop

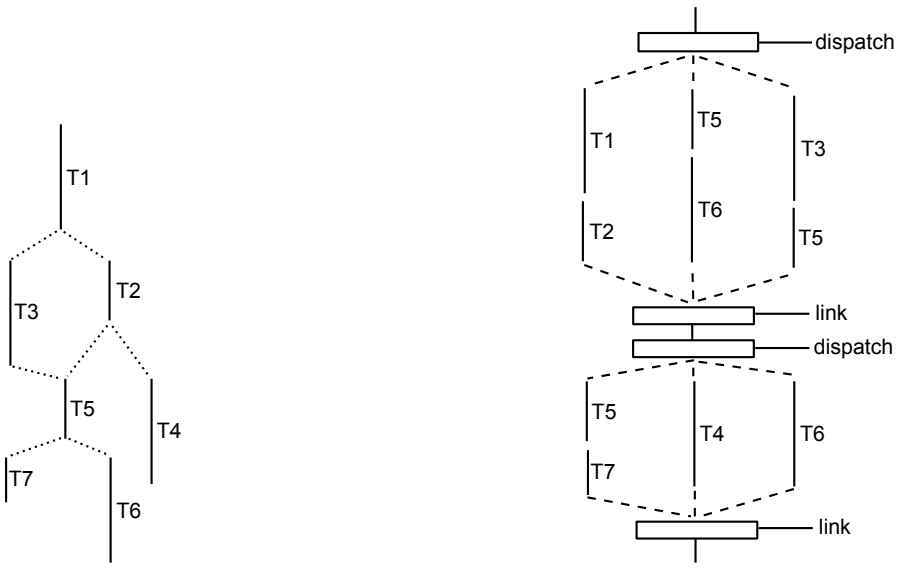
Figure 2.1: Loops classification according to inter-iterations dependency

ations must *wait* for the required calculations from lower index iterations to be evaluated (See Figure 2.1d). In contrast, with lexically-forward loops, lower index iteration access data used at higher index. Loops can run in parallel without delay if both iterations loaded their data at the beginning of the iteration.

Traces are defined as the hot paths in program execution. Traces can be parts of loops, or spans multiple iterations, can be parts of individual methods, or spans multiple methods. Traces are detected dynamically at runtime and is defined as a set of basic blocks (set of instructions ended with a branch). Similarly, traces can run in parallel on multiple threads, then are linked together according to their entries and exits points. Figure 2.2 shows an example of traces that run in parallel on three processors.

In contrast to data parallelism (e.g., a loop operating on the same data as in Figure 2.1), *Task parallelism* is a form of parallelization wherein tasks or subroutines are distributed over multiple processors. Each task has a different control flow and processing a different set of data, however, they can communicate with each other through passing data between threads. Running different tasks in parallel reduces the overall runtime of the program.

Another classification of parallel techniques is according to user intervention. In *manual* parallelization, the programmer uses a programming language constructs to define parallel portions of the code and protects shared data through synchronization primitives. An alternative approach is that programmer designs and develops the program to run sequentially, and uses interactive tools that analyze the program (statically or dynamically), and provide the programmer with hints about data dependency and eligible portions for parallelization. Iteratively, the programmer modifies the code and compares the performance scaling of different threading designs to get the best possible speedup. A clear examples of this *semi-automated* technique are Intel Parallel Advisor [1], and Paralax compiler [129]. Lastly, *automatic parallelization* aims to parallelize program without any user intervention.



(a) Traces (b) Parallel Traces

Figure 2.2: Running traces over three processors

2.1 Manual Parallelization

Designing a parallel program that run efficiently in a multi-processor environment is not trivial procedure as it involves multiple factors and requires a skilled programmer. Maintenance and debugging of a parallel program is a nightmare and incredibly difficult, as it involves racing, invalid shared data access, live and deadlock situations, and non-deterministic scenarios.

Firstly, the programmer must understand the problem that he tries to solve, and the nature of input data. A problem can be partitioned according to the domain (i.e., input data decomposition), or through functional decomposition (cooperative parallel sub-tasks).

After partitioning the problem, usually there is some kind of communications and shared data access between different tasks. Both communications and shared access presents a challenge (and usually delay) to the parallel program. An application is *embarrassingly parallel* when the communications between its sub-tasks are rare and does not require a lot of data sharing. The following factors needs to be considered for the inter-task communications

1. Frequency; An application exhibits fine-grained parallelism if its sub-tasks communicate frequently while in coarse-grained parallelism, application experience less communication. As communication take place through a communication media (e.g., bus), the contention over the communication media directly affect the overall performance, especially if the media is being used for other purposes (e.g., transferring data from and to processors).

```
#pragma omp for private(n) shared(total) ordered schedule(dynamic)
for(int n=0; n<100; ++n)
{
    files[n].compress();

    total += get_size(files[n]);

    #pragma omp ordered
    send(files[n]);
}
```

Figure 2.3: OpenMP code snippet

2. Cost; the communication cost is determined by; the delay in sending and receiving the information and the amount of transferred data.

3. Blocking; communication can either synchronous or asynchronous. Synchronous communications present a blocking at the receiver (and sender as well when an acknowledgment is required). On the other hand, asynchronous communications allow both sides to proceed processing but introduces more complexity to the application design.

Protecting shared data from concurrent access requires a *synchronization* mechanism such as barriers and locks. Synchronization primitives usually introduce a bottleneck and a significant overhead to the processing time. Besides, a misuse of locks can lead the application to deadlock (i.e, two or more tasks are each waiting for the other to finish), livelock (i.e, tasks are not blocked, but busy responding to each other to resume work), or starvation (i.e., greedy set of tasks keep holding the locks for long time).

Resolving data and control dependencies between tasks is the responsibility of the programmer. A good understanding of the inputs and underlying algorithm helps in determining independent tasks, however, there are tools that could help in this step [1].

Finally, the programmer should maintain a load-balance between the computation resources. For example, static assignment of tasks to processors (e.g., round-robin) is a light technique, but may lead to low utilization; while dynamic assignment of tasks requires monitoring the state of tasks (i.e., start and end times) and handling a shared queue of ready to execute tasks.

As an example of parallel programming languages, OpenMP is a compiler extension for C, C++ and Fortran languages that support adding parallelism to existing code without significant changes. OpenMP primarily focuses on parallelizing loops and running iterations in parallel with optional ordering capabilities. Figure 2.3 shows an example of OpenMP parallel code that compress a hundred files, but sending them in order, and calculate the

total size after compression. Programmer can define shared variables (e.g., *total* variable), and local thread variables (e.g., the loop counter n). Shared variables are transparently protected from concurrent access. A private copy is created for variables that are defined as thread local (i.e using *private* primitive).

2.2 Automatic Parallelization

Past efforts on parallelizing sequential programs can be broadly classified into *speculative* and *non-speculative* techniques. Non-speculative techniques, which are usually compiler-based, exploit loop-level parallelism, and differ on the type of data dependency that they handle (e.g., static arrays, dynamically allocated arrays, pointers) [17, 55, 113, 40].

In speculative techniques, parallel sections of the code run speculatively, guarded by a compensating mechanism for handling operations violating the application consistency. The key idea is to provide more concurrency where extra computing resources are available. Speculative techniques can be broadly classified based on

1. what program constructs they use to extract threads (e.g., loops, subroutines, traces, branch targets),
2. whether they are implemented in hardware or software,
3. whether they require source codes, and
4. whether they are done online, offline or both.

Of course, this classification is not mutually exclusive. The execution of speculative code is either *eager* or *predictive*. In eager speculative execution, every path of the code is executed (with the assumption of unlimited resources), however, only the correct value is committed. With predictive execution, selected paths are executed according to a prediction heuristics. If there is a misprediction, the execution is unrolled and re-executed.

Among speculative techniques, two appealing primary approaches: thread-level speculation (TLS), and transactional memory (TM).

2.3 Thread Level Speculation

Thread-Level Speculation (TLS) refers to the execution of out-of-order (unsafe) operations and caching the results to a thread-local storage or buffer (usually using the processor cache). TLS assigns an age to each thread according to how early the code it executes. The thread with the earliest age is marked as *safe*.

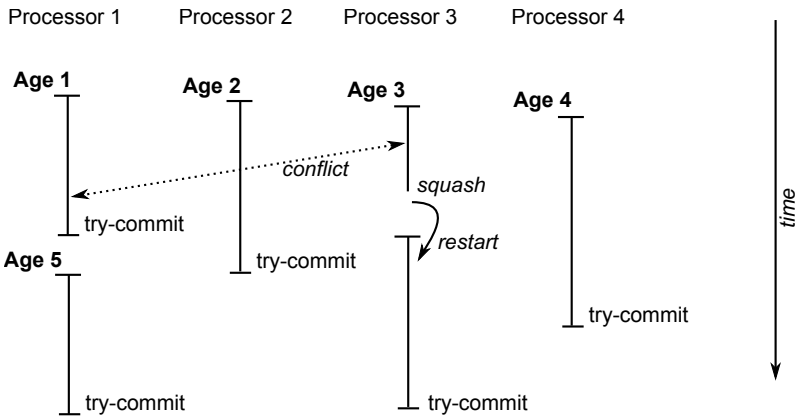


Figure 2.4: Example of thread-level speculation over four processors

The speculative thread is subject to overflow for its buffer. When a thread buffer is full the thread either stall (till it became the lowest age) or is squashed and restarted. An exception from this is the safe thread (the one executing the earliest code). TLS monitors dependency violations (e.g., through checking cache lines requests). For example, write-after-read (WAR) dependency violation occurs when a higher age speculative thread modifies a value before another lower age thread needs to read it. Similarly, when a higher age speculative thread reads an address, then a lower age thread changes the value of this thread, then this is another dependency violation named Read-after-write (RAW). A different type of violations is the control dependency when a speculative thread executing an unreachable code due to a change in the program control flow. Any dependency violations (data or control) causes the higher age thread to be squashed and restarted. Figure 2.4 shows an example of TLS using four processors.

To summarize, TLS runs code speculatively, and eventually the correct order is determined. The evaluated operations are detected to be correct or not. Incorrect results are discarded and the thread is restarted.

TLS is usually implemented through hardware. Speculative threads use processor cache as a buffer for thread memory changes. Cache coherence protocols are overloaded with the TLS monitoring algorithm. Squashing thread is done by aborting the thread and discard the cache content while committing thread changes is done by flushing the cache to the main memory. The same concept can be applied using software [78, 113, 40] (with performance issues) especially when low-level operations is not feasible [34, 29] (e.g., with Java based frameworks). TLS software implementations use a data access signature (or summaries) to detects conflicts between speculative threads. An access signature represents all accessed addresses using the current thread. Conflict is detected by intersecting signatures for partial matches.

2.4 Transactional Memory

Lock-based synchronization is inherently error-prone. Coarse-grained locking, in which a large data structure is protected using a single lock is simple and easy to use but permits little concurrency. In contrast, with fine-grained locking [89, 69], in which each component of a data structure (e.g., a bucket of a hash table) is protected by a lock, programmers must acquire necessary and sufficient locks to obtain maximum concurrency without compromising safety. Both these situations are highly prone to programmer errors. In addition, a lock-based code is non-composable. For example, atomically moving an element from one hash table to another using those tables' (lock-based) atomic methods is difficult: if the methods internally use locks, a thread cannot simultaneously acquire and hold the locks of the two tables' methods; if the methods were to export their locks, that will compromise safety. Furthermore, lock-inherent problems such as deadlocks, livelocks, lock convoying, and priority inversion haven't gone away. For these reasons, the lock-based concurrent code is difficult to reason about, program, and maintain [64].

Transactional memory (TM) [59] is a promising alternative to lock-based concurrency control. It was proposed as an alternative model for accessing shared memory addresses, without exposing locks in the programming interface, to avoid the drawbacks of locks. With TM, programmers write concurrent code using threads but organize code that read/write shared memory addresses as atomic sections. Atomic sections are defined as *transactions* in which reads and writes to shared addresses appear to take effect instantaneously. A transaction maintains its read-set and write-set, and at commit time, checks for conflicts on shared addresses. Two transactions conflict if they access the same address and one access is a write. When that happens, a contention manager [118] resolves the conflict by aborting one and allowing the other to proceed to commit, yielding (the illusion of) atomicity. Aborted transactions are re-started, often immediately. Thus, a transaction ends by either committing (i.e., its operations take effect) or by aborting (i.e., its operations have no effect). In addition to a simple programming model, TM provides performance comparable to highly concurrent, fine-grained locking implementations [43, 22] and is composable [60]. Multiprocessor TM has been proposed in hardware (HTM), in software (STM), and in hardware/software combination (HyTM).

Transactional Memory algorithms differ according to their design choices. An algorithm can apply its changes directly to the memory and maintain an undo-log for restoring the memory to a consistent state upon failure. Such an optimistic approach fits the situation with rare conflicts. In contrast, an algorithm can use a private thread-local buffer to keep its changes invisible from other transactions. At commit, the local changes are merged with the main memory. Another orthogonal design choice is the time of conflict detection. Transactions may acquire access (lock) on its write-set at encounter time or during the commit phase.

Figure 2.5 shows an example transactional code. Atomic sections are executed as transactions. Thus, the possible values of A and B are either 42 and 42, or 22 and 11, respectively.

A = 10, B = 20;

THREAD A

```
atomic{
    B = B + 1;
    A = B * 2;
}
....
```

THREAD B

```
atomic{
    B = A;
}
....
....
```

Figure 2.5: An example of transactional code using atomic TM constructs

An inconsistent view of a member (e.g., A=20 and B=10), due to atomicity violation or interleaved execution, causes one of the transactions to abort, rollback, and then re-execute.

Motivated by TM's advantages, several recent efforts have exploited TM for automatic parallelization. In particular, trace-based automatic/semi-automatic parallelization is explored in [20, 21, 27, 42], which use HTM to handle dependencies. [104] parallelizes loops with dependencies using thread pipelines, wherein multiple parallel thread pipelines run concurrently. [87] parallelizes loops by running them as transactions, with STM preserving the program order. [122] parallelizes loops by running a non-speculative "lead" thread, while other threads run other iterations speculatively, with STM managing dependencies.

2.4.1 NOrec

No Ownership Records Algorithm [38], or *NOrec*, is a lazy software transactional memory algorithm that uses a minimal amount of meta-data for accessed memory addresses. Unlike other STM algorithms, NOrec does not not associate ownership records (orecs) for maintaining its write-set. Instead, it uses a value based validation during commit time, to make sure the read values still have the same values the transaction already used during its execution.

Transactions use a write-buffer to store their updates, the implementation of the write-buffer is a linear-probed hash table with versioned buckets to support O(1) clearing (when transaction descriptor is reused). NOrec uses a *lazy* locking mechanism, which means written addresses are not locked until the commit time. This approach reduces the locking time for the accessed memory location which allows readers to proceed without writers interference.

NOrec employs a single global sequence lock. Whenever a transaction commits, the global lock is acquired and is incremented. This assumption limits the system to have a single committer transaction at a time. The commit procedure starts by incrementing the sequence lock atomically. Failing to increment the lock means another writer transaction are trying to commit, so the current transaction needs to validate its read-set and wait for the other transaction to finish the commit. Upon successful increment, the transaction acquires locks

on its write-set, expose the changes to the memory, and release the locks.

Another drawback of the algorithm is that before each read, it is required to validate the whole read-set. This is required to maintain opacity [53] – a TM feature which mandates that invalid transactions to always read consistent view of memory. NOrec tries to avoid unneeded validation by doing it whenever the global sequence got changed (i.e., a transaction commit take place).

NOrec is the default TM algorithm for Lerna. In Chapter 5 we describe our variant of the algorithm that preserve ordering between concurrent transactions.

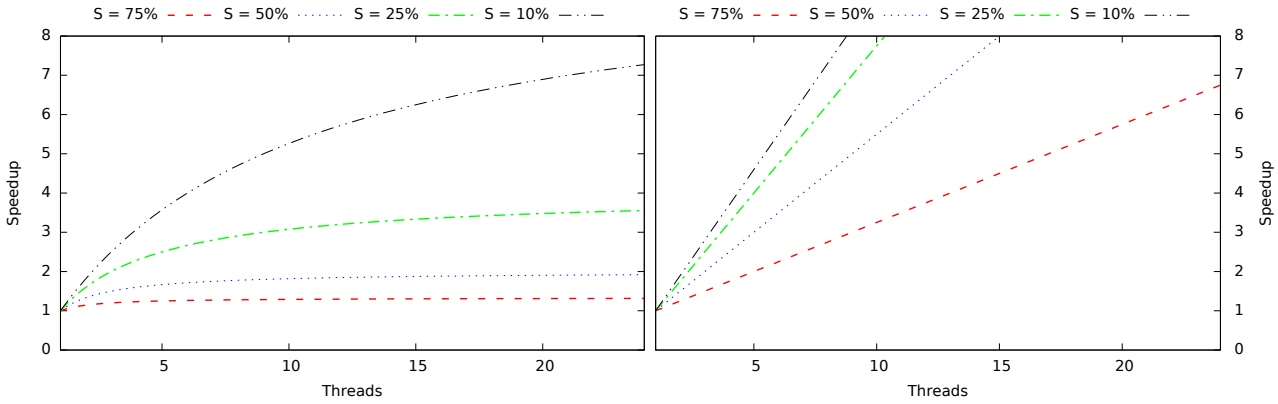
2.4.2 TinySTM

TinySTM [50] is a lock-based lightweight STM algorithm. It uses a single-version word-based variant of LSA [108] algorithm. Similar to other word-based locking algorithms [38, 43], TinySTM relies upon a shared array of locks that cover all memory addresses. Multiple addresses are covered by the same lock, and each lock uses a single bit as a state (i.e., locked or not), and the remaining bits as a version number. This version number indicates the timestamp of the last transaction the wrote to any of the addresses covered by this lock. As a lock covers a portion of the address space, false conflicts could occur when concurrent transactions access adjacent addresses.

Unlike NOrec, TinySTM uses encounter time locking (ETL); transaction acquires locks during the execution. The use of ETL is twofold: conflicts are discovered at an early time which avoids wasting processing time executing a doomed transaction; and simplifies the handling of read-after-write situations. The algorithm uses a time-based design [43, 108] by employing a shared counter as a clock. Update transactions acquire a new timestamp on commit, validate its read-set, then store the timestamp to the versioned locks of its write-set.

TinySTM is proposed with two strategies for memory access: write-through and write-back; each has its advantages and limitations. Using write-back strategy, updates are kept at a local transaction write-buffer until commit time, while in write-through updates go to the main memory and the old values are stored in an undo log. With write-through, transaction has lower commit-time overhead and faster read-after-write/write-after-write handling. However, the abort is costly as it requires restoring the old values of written addresses. On the other hand, in write-back the abort procedure simply discards the read and write sets, but commit requires validating the read-set and moving the write-set values from the local write-buffer to the main memory.

The use of ETL is interesting to our work as it enable early detection of conflicting transactions which save processing cycles. Additionally, the write-through strategy is perfect for low-contention workloads as it involves lightweight commit that could lead to a comparable performance to the sequential execution.



(a) Amdahl's Low (b) Gustafson's Low

Figure 2.6: Maximum Speedup according to Amdahl's and Gustafson's Lows

2.5 Parallelism Limits and Costs

Amdahl's law is a formula that defines the upper bound on expected speedup to an overall system when only part of the system is improved. Let S is the percentage of the program serial portion of the code (i.e., not included into the parallelism). When executing the program using n threads, then the expected execution time is

$$Time(n) = Time(1) * (S + 1/n * (1 - S))$$

Therefore, the maximum speedup is given by the following formula

$$Speedup(n) = 1/(S + 1/n * (1 - S)) = n/(1 + S * (n - 1))$$

Figure 2.6a shows the maximum possible speedup for different percentages of the sequential portion of the code with different number of threads. With only 10% of the sequential code, the maximum speedup using 20 threads is around 7× only. Thus, at some point adding an additional processor to the system will add less speedup than the previous one, as the total speedup heads toward the limit of $1/(1 - n)$. However, here we assume a fixed size of the input. Usually adding more processors permits solving larger problems (i.e., increasing the input size), consequently increases the parallel portion of the program. This fact is captured by Gustafson's Law which states that computations involving arbitrarily large data sets can be efficiently parallelized. Accordingly, the speedup can be defined by the following formula (See Figure 2.6b)

$$Speedup(n) = n - S * (n - 1)$$

In general, parallel applications are much more complex than sequential ones. The complexity appears in every aspect of the development cycle including: design, development, debugging, tuning, and maintenance. Add to that the hardware requirements for running a parallel system. The return value per added processor is not guaranteed to be reflected in the overall performance. On the contrary, sometimes splitting the workload over even more threads increases rather than decreases the amount of time required to finish. This is known as *parallel slowdown*.

Chapter 3

Past & Related Work

3.1 Transactional Memory

The classical solution for handling shared memory during concurrent access is lock-based techniques [7, 71], where locks are used to protect shared addresses. Locks have many drawbacks including deadlocks, livelocks, lock-convoying, priority inversion, non-composability, and the overhead of lock management.

TM, proposed by Herlihy and Moss [67], is an alternative approach for shared memory access, with a simpler programming model. Memory transactions are similar to database transactions: a memory transaction is a self-maintained entity that guarantees atomicity (all or none), isolation (local changes are hidden till commit), and consistency (linearizable execution). TM has gained significant research interest including that on STM [119, 90, 61, 58, 65, 66, 86], HTM [67, 57, 6, 18, 91], and HyTM [12, 39, 92, 77]. STM has relatively larger overhead due to transaction management and architecture-independence. HTM has the lowest overhead but assumes architecture specializations. HyTM seeks to combine the best of HTM and STM.

STM can be broadly classified as static or dynamic. In static STM [119], all accessed addresses are defined in advance, while dynamic STM [65, 66, 86] relaxes that restriction. The dominant trend in STM designs is to implement the single-writer/multiple-reader pattern, either using locks [44, 43] or obstruction-free (i.e., a single thread executed in isolation will complete its operation with a bounded number of steps) techniques [109, 65], while few implementations allow multiple writers to proceed under certain conditions [110]. In fact, it is shown in [48] that obstruction-freedom is not an important property and results in less efficient STM implementations than lock-based ones.

Another orthogonal TM property is address acquisition time: pessimistic approaches acquire addresses at encounter time [48, 22], while optimistic approaches do so at commit time [44,

43]. Optimistic address acquisitions generally provide better concurrency with acceptable number of conflicts [43]. STM implementations also rely on write-buffer [85, 86, 65] or undo-log [92] for ensuring a consistent view of memory. In write-buffer, address modifications are written to a local buffer and take effect at commit time. In the undo-log method, writes directly change the memory, and the old values are kept in a separate log to be retrieved at abort.

Nesting (or composability) is an important feature for transactions as it allows partial rollback and introduces semantics between the parent transaction and its enclosed ones. Earlier TM implementations did not support nesting or simply flattened nested transactions into a single top-level transaction. Harris *et. al.* [60] argue that closed nested transactions, supporting partial rollback, are important to implementing *composable* transactions, and presented an $orElse$ construct that relies upon closed nesting. In [4], Adl-Tabatabai *et. al.* presented an STM that provides both nested atomic regions and $orElse$, and introduced the notion of mementos to support efficient partial rollback. Recently, a number of researchers have proposed the use of open nesting. Moss described the use of open nesting to implement highly concurrent data structures in a transactional setting [93]. In contrast to the database setting, the different levels of nesting are not well-defined; thus different levels may conflict. For example, a parent and child transaction may both access the same memory location and conflict.

3.2 Parallelization

Parallelization has been widely explored with different levels of programmer intervention. Manual parallelization techniques rely on the programmer for analysis and design phases but assist him in the implementation and performance tuning. Open Multi-Processing (OpenMP) [37] defines an API that supports shared memory multiprocessing programming in C, C++, and Fortran. The programmer uses OpenMP directives to define parallel sections of the code and the execution semantics (e.g., ordering). Synchronization of shared access is handled through directives that define the scope of access (i.e., shared or private). OpenMP transparently and efficiently handles the low-level operations such as locking, scheduling and threads stalling. MPI [120], Message Passing Interface, is a message-based language-independent communications protocol used for parallel computing. MPI offers a basic concepts for messaging between parallel processes such as: grouping and partitioning of processes, type of communication (i.e., point-to-point, broadcasting or reduce), interchanged data types, and synchronization (global, pairwise, and remote locks). NVIDIA introduced CUDA [100] as a parallel programming and computing platform for its graphics processing units (GPUs); This enables programmers to access the GPU virtual instruction set and memory, and use it for general purpose processing (not exclusively graphics computation). GPUs rely on using *many* concurrent slow threads than using limited count of cores with high speed (as in CPUs), which makes GPUs suitable for data-parallel computations.

An alternative approach, semi-automatic parallelization, is to provide programmer with hints and design decisions that helps him to write a code that is more eligible for parallelization, or detects data dependencies and shared accesses during design phase when they are less expensive to fix. Paralax [129] is a compiler framework for extracting parallel code using static analysis. Programmers use annotations to assist the compiler in finding parallel sections. DiscoPoP [81] and Kremlin [51] are runtime tools that discover and present ranked suggestions for parallelization opportunities. Intel Parallel Advisor [1] (or Advisor XE) is a shared memory threading design and prototyping tool. Advisor XE helps programmers to detect parallel sections and compare the performance using different threading models. Additionally, it finds data dependencies which enables eliminating them, if possible.

Automatic parallelization aims to produce best-effort performance without any programmer intervention; This relieve programmers from designing and writing complex parallel applications, besides, it is highly useful for legacy code. Polaris [49], one of the earliest parallelizing compiler implementations, proposes a source-to-source transformation that generates a parallel version of the input program (Fortran). Another example is Standford SUIF compiler [55]; It determines parallelizable loops using a set of data dependency techniques such as data dependence analysis, scalar privatization analysis, and reduction recognization. SUIF optimizes cache usage through ensuring that processors re-use the same data and defragments shared addresses.

Parallelization techniques can be classified according to its granularity. Significant efforts have focused on enhancing fine-grain parallelism such as: parallelization of nested loops [19, 107, 36], regular/irregular array accesses [114], and scientific computations [134] (e.g., dense/sparse matrix calculations). Sohi *et. al.* [121] increase instruction-level parallelism by dividing tasks for speculative execution on functional units; [112] does so with a trace-based processor. Nikolov *et. al.* [99] use a hybrid processor/SoC architecture to exploit nested loop parallelism, while [5] uses postdominance information to partition tasks on a multithreaded architecture. However, fine-grain parallelism is insufficient to exploit CMP parallelism. Coarse-grain parallelism focuses on parallelizing tasks as a unit of work. In [126], programmer manually annotates concurrent and synchronized code blocks in C programs and then uses those annotations for runtime parallelization. Gupta *et. al.* [54] and Rugina *et. al.* [113] do a compile-time analysis to exploit parallelism in array-based, divide-and-conquer programs.

3.3 Optimistic Concurrency

Optimistic concurrency techniques, such as thread-level speculation (TLS) and Transactional Memory (TM), have been proposed as a way of extracting parallelism from legacy code. Both techniques split an application into sections using hardware or a compiler, and run them speculatively on concurrent threads. A thread may buffer its state or expose it and use a compensating procedure. At some point, the executed code may become safe and the code proceeds 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 [11, 106, 105] to get the best of the two techniques.

Parallelization using thread-level speculation (TLS) has been extensively studied using both hardware [124, 75, 56, 32] and software [107, 82, 107, 30, 88]. It was originally proposed by Rauchwerger *et. al.* [107] 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). In [88], authors share the same sweet-spot we aim for, applications with non-partitionable accesses and data sharing by providing a low-overhead STM algorithm.

3.3.1 Thread-Level Speculation

Automatic parallelization for thread-level speculation (TLS) hardware has been extensively studied, most of which largely focus on loops [107, 127, 46, 83, 123, 31, 33, 103, 131]. Loop parallelization using TLS is proposed in both hardware [102] and software [107, 13]. The LRPD Test [107] determines if the loop has any cross-iteration dependencies (i.e., DOALL loop) and runs it speculatively; At runtime, a validation step is performed to check if the accessed data is affected by any unpredictable control flow. Saltz and Mirchandane [117] parallelize DOACROSS loops by assigning iterations to processors in a wrapped manner. To prevent data dependency violations, processors have to stall till the correct values are produced. Polychronopoulos [101] proposes running the maximal set of continuous iterations with no dependency concurrently; subsequently this method does not fully utilize all processors. An alternative approach is to remove dependencies between the iterations. Krothapalli *et. al.* [76] propose a runtime method to remove anti write-after-read (WAR) and write-after-write (WAW) dependencies. This method helps to remove dependencies caused by reusing memory/registers, but does not remove computation dependencies.

Prabhu *et. al.* [102] present some guidelines for programmers to manually parallelize the code using TLS. In their work, a source-to-source transformation is performed to run the loops speculatively, and TLS hardware detects the data dependency violations and act accordingly (i.e., squash the threads and restart the iteration). Zilles *et. al.* [133] introduce an execution paradigm called Master/Slave Speculative Parallelization (MSSP). In MSSP, a master processor executes an approximate version of the program, based on common-case execution, to compute selected values that the full program's execution is expected to compute. The masters results are checked by slave processors that execute the original program.

Automatic and semi-automatic parallelization without TLS hardware have also been explored [78, 113, 40, 34, 29]. In [104], Raman *et. al.* propose a software approach that generalizes existing software TLS memory systems to support speculative pipelining schemes, and efficiently tunes it for loop parallelization. Jade [78] is a programming language that supports coarse-grain concurrency and exploits a software TLS technique. Using Jade, pro-

grammers augment the code with data dependency information, and the compiler uses this to determine which operations can be executed concurrently. Deutsch [40] analyzes symbolic access paths for interprocedural may-alias analysis, toward exploiting parallelism. In [34], Choi *et. al.* present escape analysis for Java programs for determining object lifetimes toward enhancing concurrency.

3.3.2 Parallelization using Transactional Memory

Tobias *et. al.* [47] propose 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 utilize all cores, besides, relying on only at runtime for parallelization introduce nonnegligible overhead to the framework. Similarly, DeVuyst *et. al.* [41] use HTM to optimistically run parallel sections, which are detected using special hardware. Sambamba [125] showed that static optimization at compile-time does not exploit all possible parallelism. Similar to our work, it used Software Transactional Memory (STM) with an adaptive runtime approach for executing parallel sections. It relies on user input for defining parallel sections. Gonzalez *et. al.* [52] 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, HydraVM does not require special hardware and is fully automated, with an optional user interaction for improving speedup.

The study at [130] 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 model that captures data and inter-dependencies. For a set of benchmarks [25, 9, 98, 28], the study showed important features for each like: size of read and write sets, dependencies density, and size of parallel sections.

Mehrara *et. al.* [87] present *STMlite*: a lightweight STM implementation customized to facilitate profile-guided automatic loop parallelization. In this scheme, loop iterations are divided into chunks and distributed over available cores. An outer loop is generated around the original loop body to manage parallel execution between different chunks.

In MTX, a transaction is running under more than one thread. Vachharajani *et. al.* [128] present a hardware memory system that supports MTXs. Their system changes hardware cache coherence protocol to buffer speculative states and recover from mis-speculation. Software Multi-threaded Transactions (SMTX) are used to handle memory access of speculated threads. Both SMTX and STMlite use a centralized transaction commit manager and conflict detection that is decoupled from the main execution.

Sambamba [125] showed that static optimization at compile-time does not exploit all possible parallelism. Similar to our work, it used Software Transactional Memory (STM) with an adaptive runtime approach for executing parallel sections. It relies on user input for defining parallel sections. Gonzalez *et. al.* [52] proposed a user API for defining parallel sections and

the ordering semantics. Based on user input, STM is used to handle concurrent sections.

3.4 Comparison with existing work

Most of the methodologies, tools and languages for parallelizing programs target scientific and data parallel computation intensive 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 [97, 111, 82, 68].

Despite the flexibility and ease of use of TM, the primary drawback of using it for parallelization is the significant overhead of code execution and validation [26, 116]. Previous work [52, 125] relied primarily on the programmer for specifying parallel sections, and defining ordering semantics. The key weaknesses with the programmer reliance are:

1. it requires a full understanding of the software (e.g., the implementation algorithm and the input characteristics) and mandates the existence of the source code;
2. it does not take into account TM characteristics and factors of overhead; and
3. it uses TM as a black box utility for preserving data integrity.

In contrast, the solutions proposed in this thesis target common programs and analyzes the code at its intermediate representation without the need for its original source code. This makes us independent of the language in which the code was written with, and does not enforce the user to expose the source code to our framework, especially when it is not available for him in some cases.

Additionally, In Lerna, we employ alias analysis and propose two novel techniques: *irrevocable transactions* and *transactional increment*, for reducing read-set size and validation overhead and eliminates some reasons of transaction conflicts. As far as we know, this is the first study that attempts to reduce transactional overhead based on program characteristics.

HydraVM and MSSP [133] shares the same concept of using execution traces (not just loops such as [87, 122]). However, MSSP uses superblock [70] executions for validating the main execution. In contrast, HydraVM splits execution equally on all threads and uses STM for handling concurrent memory accesses. Perhaps, the closest to our proposed work are [21] and [47]. Our work differs from [21] and [47] in the following ways. First, unlike [47], we propose STM for concurrency control, which does not need any hardware transactional support. Second, [21] is restricted to recursive programs, whereas we allow arbitrary programs. Third, [21] does not automatically infer transactions; rather, entire work performed in tasks (of traces) is packaged as transactions. In contrast, we propose compile and runtime program analysis techniques that identify traces, which are executed as transactions.

Our work is fundamentally different from past STM-based parallelization works in that: we benefit from static analysis for reducing transactional overheads, and automatically identify parallel sections (i.e., traces or loops) by compile and runtime program analysis techniques, which are then executed as transactions. Additionally, our work targets arbitrary programs (not just recursive such as [21]), is entirely software-based (unlike [21, 47, 41, 128]), and do not require program source code. Thus, our proposed work of STM-based parallelization (with the consequent advantages of STM's concurrency control; completely software-based; and no need for program sources), has never been done before.

Chapter 4

HydraVM

In this chapter, we present a virtual machine, called HydraVM, that automatically extracts parallelism from legacy sequential code (at the bytecode level) through a set of techniques including online and offline code profiling, data dependency analysis, and execution analysis. HydraVM is built by extending the Jikes RVM [8] and modifying its baseline compiler, and exploits software transactional memory to manage concurrent and out-of-order memory accesses.

HydraVM targets extracting parallel *jobs* in the form of code traces. This approach is different from loop parallelization [19], because a trace is equivalent to an execution path which can be a portion of a loop, or spans loops and method calls. Traces were invented in [10] as part of HP's Dynamo optimizer, which optimizes native program binary at runtime using a trace cache.

To handle potential memory conflicts, we develop ByteSTM, which is a VM-level STM implementation. In order to preserve original semantics, ByteSTM suspends completed transactions till their valid commit times are reached. Aborted transactions discard their changes and are either terminated (i.e., a program flow violation or a misprediction) or re-executed (i.e., to resolve a data-dependency conflict).

We experimentally evaluated HydraVM on a set of benchmark applications, including a subset of the JOlden benchmark suite [23]. Our results reveal speedup of up to 5× over the sequential code.

4.1 Program Reconstruction

The program can be represented 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. Thus, any program can be represented by a directed graph in which nodes represent basic blocks and edges represent the program control flow – i.e., control flow graph (CFG). Basic blocks can be determined at compile-time. However, our main goal is to determine the context and frequency of reachability of the basic blocks – i.e., when the code is revisited through execution.

Basic blocks can be grouped according to their runtime behavior and execution paths. A *Trace* is a set of connected basic blocks at the CFG, and it represents an execution path (See Figure 4.5a). A *Trace* contains one or more conditional branches that may transfer the control out of the trace boundaries, namely *exits*. Exists transfer control to the program or to another trace. It is possible for a *trace* to have more than one *entry*, and a trace with a single entry is named a *Superblock* [70].

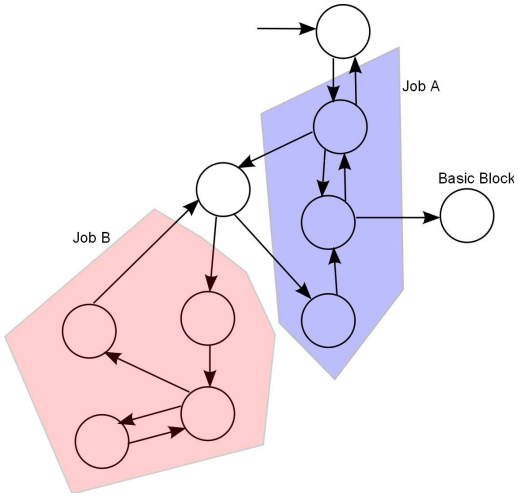
Our basic idea is to optimistically split code into parallel traces. For each trace, we create a synthetic method (See Figure 4.1b) that:

- Contains the code of the trace.
- Receives its entry point and any variables accessed by the trace code as input parameters.
- Returns the *exit* point of the trace (i.e., the point where the function returns).

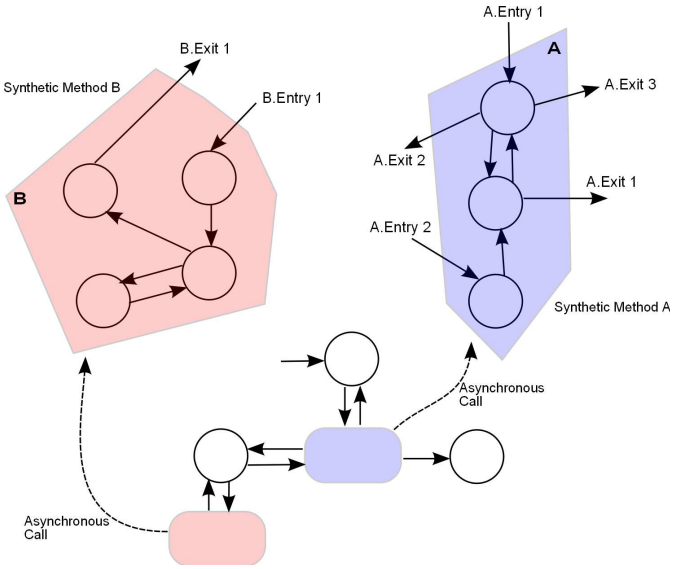
Synthetic methods are executed in separate threads as memory transactions, and a TM library is used for managing the contention. We name a call to the synthetic methods with a specific set of inputs as a *job*. Multiple jobs can execute the same code (trace), but with different inputs. As concurrent jobs can access and modify the same memory location, it is required to protect memory against invalid accesses. To do so, we employ TM to organize access to memory and to preserve memory consistency. Each transaction is mapped to a subset of the job code or spans multiple jobs.

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:

- It is reachable by future executions of the program; and
- It does not cause a memory conflict with any other job having an older chronological order.



(a) Control Flow Graph with two Traces



(b) Transformed Program

Figure 4.1: Program Reconstruction as Jobs

As we will detail in Section 4.2, any execution of a parallel program produced after our transformations is made of a sequence of jobs committed after a successful execution.

In a nutshell, the parallelization targets those blocks of code that are prone to be parallelized 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 presence of data-conflicts (e.g., the case of two iterations of one loop activated in parallel and modifying the same part of a shared data structure). Clearly the presence of more conflicts leads to less parallelism and thus poor performance.

4.2 Transactional Execution

TM encapsulates optimism: a transaction maintains its read-set and write-set, i.e., the objects read and written during the execution, and at commit time checks for conflicts on shared objects. Two transactions conflict if they access the same object, and one access is a write. When this happens, a contention manager [118] solves the conflict by aborting one and allowing the other to proceed to commit, yielding (the illusion of) atomicity. Aborted transactions are re-started, often immediately. Thus, a transaction ends by either committing (i.e., its operations take effect), or by aborting (i.e., its operations have no effect).

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.

In our model, jobs run (which contain transactions) speculatively, but a transaction, in general, 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 code (e.g., a break condition that changes the execution flow), and transactions executing code with earlier chronological order may read future values from committed transactions that are corresponding to code with later chronological order. The following example illustrates this situation.

Consider the example in Figure 4.2, where three jobs A, B, and C are assigned to different threads T_A, T_B, and T_C and execute as three transactions t_A, t_B, and t_C, respectively. Job A can have B or C as its successor, and that cannot be determined until runtime. According to the parallel execution in Figure 4.2(b), T_C will finish execution before others. However, t_C will not commit until t_A or t_B completes successfully. This requires that every transaction must notify the STM to permit its successor to commit.

Now, let t_A conflict with t_B because of unexpected memory access. STM will favor the older transaction in the original execution and abort t_B, and will discard its local changes.

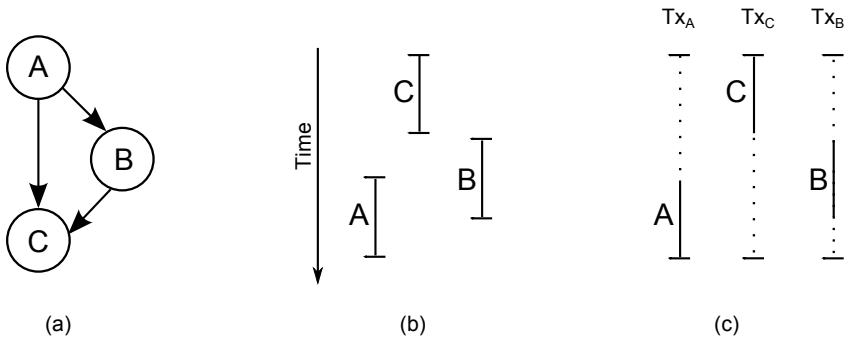


Figure 4.2: Parallel execution pitfalls: (a) Control Flow Graph, (b) Possible parallel execution scenario, and (c) Managed TM execution.

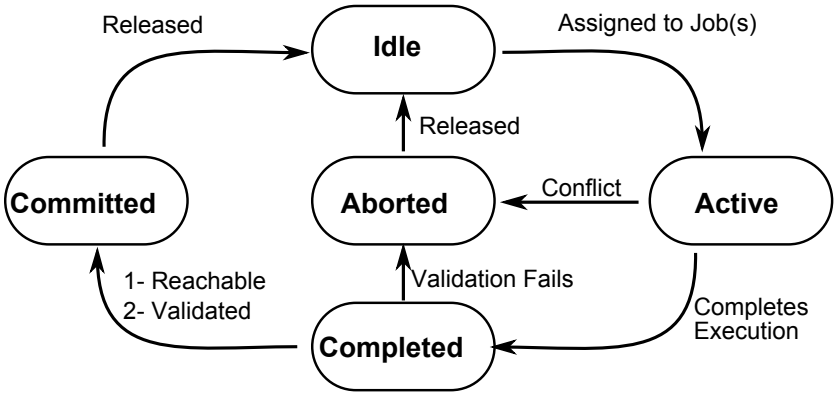


Figure 4.3: Transaction States

Later, t_B will be re-executed. A problem arises if t_A and t_C wrongly and unexpectedly access the same memory location. Under Figure 4.2(b)'s parallel execution scenario, this will not be detected as a transactional conflict (T_C finishes before T_A). To handle this scenario, we extend the lifetime of transactions to the earliest transaction starting time. When a transaction must wait for its predecessor to commit, its lifetime is extended till the end of its predecessor. Figure 4.2(c) shows the execution from the our managed TM perspective.

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. To resolve this situation, program order is maintained by deferring the commit of transactions that complete early till their valid execution time.

Motivated by that, we propose an ordered transactional execution model based on the original program's chronological order. Transactions execution 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 all locks on the written addresses. Finally, the transaction is committed when it becomes reachable from its predecessor transaction. Decoupling *completed* and *committed* states, permits threads to process next transactions without the need to wait for the transaction valid execution time.

4.3 Jikes RVM

Jikes Research Virtual Machine (Jikes RVM) is an open source implementation Java virtual machine (JVM). Jikes RVM has a flexible and modular design that support prototyping, testbed and doing the experimental analysis. Unlike most other JVM implementations, which is usually written in native code (e.g., C, C++), Jikes is written in Java. This characteristic provides portability, object-oriented design, and integration with the running applications.

Jikes RVM is divided to the following components:

- *Core Runtime Services*: This component is responsible for managing the execution of running applications, which includes the following:
  - Thread creation, management and scheduling.
  - Loading classes definitions and triggering compiler.
  - Handling calls to Java Native Interface (JNI) methods
  - Exception handling and traps
- *Magic*: This is a mechanism for handling low-level system-programing operations such as: raw memory access, uninterruptible codes, and unboxed types. Unlike all other components, this module is not written in pure Java, as it uses machine code to provide this functionality.
- *Compilers*: it reads bytecode and generates an efficient machine code that is executable for the current platform
- *The Memory Manager Toolkit (MMTK)* handles memory allocation and garbage collection.
- *Adaptive Optimization System (AOS)* allows online feedback-directed optimizations. It is responsible for profiling an executing application and triggers the optimizing compiler to improve its performance.

4.4 System Architecture

In HydraVM, we extend the AOS [8] architecture to enable parallelization of input programs, and dynamically refine parallelized sections based on execution. Figure 4.4 shows HydraVM's architecture, which contains six components:

- Profiler: performs static analysis and adds additional instructions to monitor data access and execution flow.
- Inspector: monitors program execution at runtime and produces profiling data.
- Optimization Compiler: recompiles bytecode at runtime to improve performance and triggers reloading classes definitions.
- Knowledge Repository: a store for profiling data and execution statistics.
- Builder: uses profiling data to reconstruct the program as multi-threaded code, and tunes execution according to data access conflicts.
- TM Manager: handles transactional concurrency control to guarantee safe memory and preserves execution order.

HydraVM works in three phases. The first phase focuses on detecting parallel patterns in the code, by injecting the code with hooks, monitoring code execution, and determining memory access and execution patterns. This may lead to slower code execution due to inspection overhead. *Profiler* is active only during this phase. It analyzes the bytecode and instruments it with additional instructions. *Inspector* collects information from generated instructions and stores it in the Knowledge Repository.

The second phase starts after collecting enough information in the *Knowledge Repository* about which blocks were executed and how they access memory. The *Builder* component uses this information to split the code into traces, which can be executed in parallel. The new version of the code is generated and is compiled by the *Recompiler* component. The *TM Manager* manages memory access of the execution of the parallel version, and organizes transactions commit according to the original execution order. The manager collects profiling data including commit rate and conflicting threads.

The last phase is tuning the reconstructed program based on thread behavior (i.e., conflict rate). The Builder evaluates the previous reconstruction of traces by splitting or merging some of them, and reassigning them to threads. The last two phases work in an alternative way till the end of program execution, as the second phase represents a feedback to the third one.

HydraVM supports two modes: *online* and *offline*. In the online mode, we assume that program execution is long enough to capture parallel execution patterns. Otherwise, the

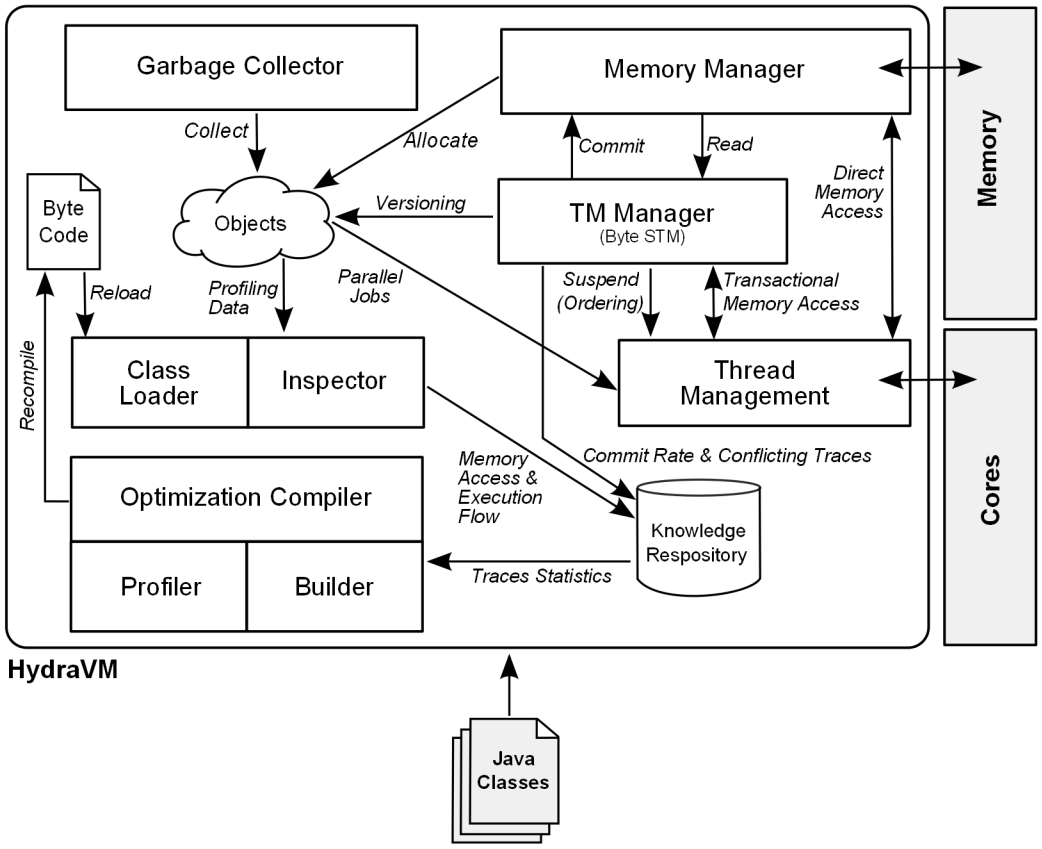


Figure 4.4: HydraVM Architecture

first phase can be done in a separate pre-execution phase, which can be classified as offline mode.

We now describe each of HydraVM's components.

4.4.1 Bytecode Profiling

To collect this information, we modify Jikes RVM's baseline compiler to insert additional instructions (in the program bytecode) at the edges of selected basic blocks (e.g., branching, conditional, return statements) that detect whenever a basic block is reached. Additionally, we insert instructions into the bytecode to:

- Statically detect the set of variables accessed by the basic blocks, and
- Mark basic blocks with irrevocable calls (e.g., input/output operations), as they need special handling in program reconstruction.

This code modification does not affect the behavior of the original program. We call this version of the modified program, *profiled bytecode*.

4.4.2 Trace detection

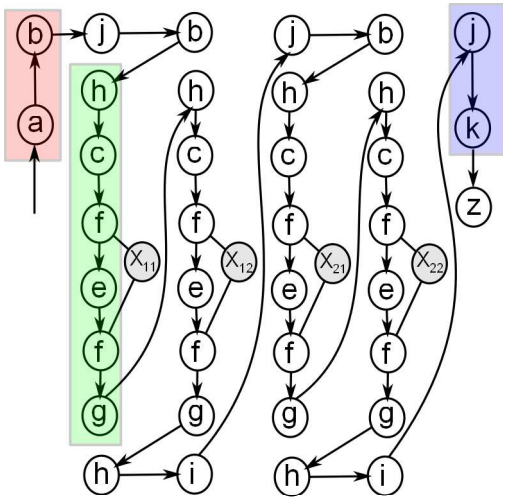
With the profiled bytecode, we can view the program execution as a graph with basic blocks and variables represented as nodes, and the execution flow as edges. A basic block that is visited more than once during execution will be represented by a *different* node each time (See Figure 4.5b). The benefits of execution graph are multifold:

- Hot-spot portions of the code can be identified by examining the graph's hot paths,
- Static data dependencies between blocks can be determined, and
- Parallel execution patterns of the program can be identified.

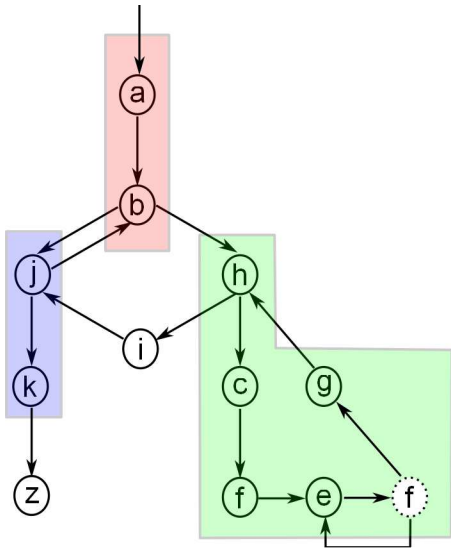
To determine traces, we use a string factorization technique: each basic block is represented by a character that acts like a unique ID for that block. Now, an execution of a program can be represented as a string. For example, Figure 4.5a shows a matrix multiplication code snippet. An execution of this code for a 2x2 matrix can be represented with the execution graph shown at Figure 4.5b, or as the string $abjbhcfefghcfefghijbhcfefghcfefghijk$. We factorize this string into its basic components using a variant of Main's algorithm [84]. The factorization converts the matrix multiplication string into $ab(jb(hcfefg)^2hi)^2jk$. Using this representation, combined with grouping blocks that access the same memory locations, we divide the code into multiple sets of basic blocks, namely *traces* (See Figure 4.5c). In our example, we detected three traces:

```
for (Integer i = 0; i < DIMx; i++)
    for (Integer j = 0; j < DIMx; j++)
        for (Integer k = 0; k < DIMy; k++)
            X[i][j] += A[i][k] * B[k][j];
```

(a) Matrix Multiplication Code



(b) 2x2 Matrix Multiplication Execution Graph with Traces



(c) Control Flow Graph with Traces

Figure 4.5: Matrix Multiplication

1. Trace ab with two entries (to a and to b), and two exits (to j and to h)
2. Trace jk with a single entry (to j), and two exits (to z and to b), and
3. Trace $hcfefg$ two entries (to h and to e) and three exits (to h, to e and to i). In this trace the inner most loop was unrolled, so each trace represents two iterations of the inner most loop. This is reflected in Figure 4.5c by adding an extra node f. Note that the transition from g to h is represented by an exit and an entry, not as an internal transition within the trace. This difference enables running multiple jobs concurrently executing the same trace code.

Thus, we divide the code, optimistically, into independent parts called traces that represent subsets of the execution graph. Each trace does not overlap with other traces in accessed variables, and represents a long sequence of instructions, including branch statements, that commonly execute in this pattern. Since a branch instruction has taken and not taken paths, the trace may contain one or both of the two paths according to the frequency of using those paths. For example, in biased branches, one of the paths is often considered; so it is included in the trace, leaving the other path outside the trace. On the other hand, in unbiased branches, both paths may be included in the trace. Therefore, a trace has multiple exits, according to the program control flow during its execution. A trace also has multiple entries, since a jump or a branch instruction may target one of the basic blocks that constructs it. The builder module orchestrates the construction of traces and distributes them over parallel threads. However, this may potentially lead to an out-of-order execution of the code, which we address through STM concurrency control (see Section 4.2). I/O instructions are excluded from parallel traces, as changing their execution order affects the program semantics, and they are irrevocable (i.e., at transaction aborts).

4.4.3 Parallel Traces

Upon detection of candidate trace for parallelization, the program is reconstructed as a producer-consumer pattern. In this pattern, two daemons threads are active, producer and consumer, which share a common fixed-size queue of jobs. Recall that a *job* represents a call to the synthetic methods executing the trace code with a specific set of inputs. The producer generates jobs and adds them in the queue, while the consumer dequeues the jobs and executes them. HydraVM uses a *Collector* module and an *Executor* module to process the jobs: the *Collector* has access to the generated traces and uses them as jobs, while the *Executor* executes the jobs by assigning them to a pool of core threads.

Figure 4.6 shows the overall pattern of the generated program. Under this pattern, we utilize the available cores by executing jobs in parallel. However, doing so requires handling of the following issues:

- Threads may finish in out of original execution order.

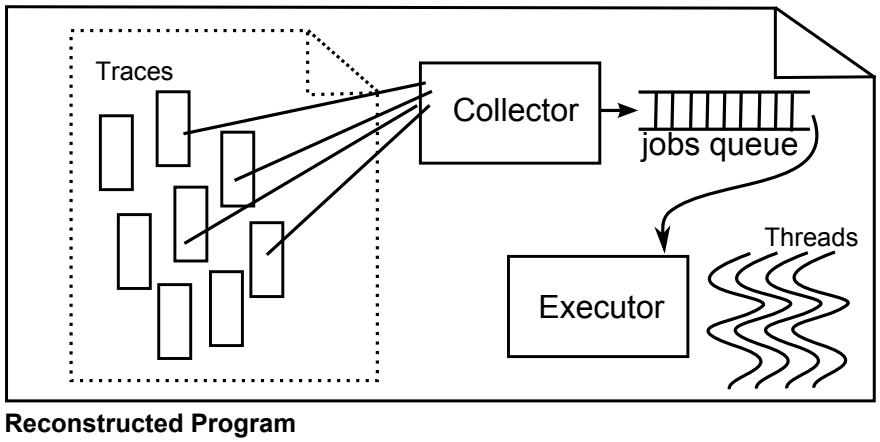


Figure 4.6: Program Reconstruction as a Producer-Consumer Pattern

- The execution flow may change at runtime causing some of the assigned traces to be skipped from the correct execution.
- Due to the differences between the actual execution flow in the profiling phase and the actual execution, memory access conflicts between concurrent accesses may occur. Also, memory arithmetic (e.g., arrays indexed with variables) may easily violate the program reconstruction (see example in Section 4.4.5).

To tackle these problems, we execute each job as a transaction. A transaction's changes are deferred until commit. At commit time, a transaction commits its changes if and only if: 1) it did not conflict with any other concurrent transaction, and 2) it is reachable under the execution.

4.4.4 Reconstruction Tuning

TM preserves data consistency, but it may cause degraded performance due to successive conflicts. To reduce this, the TM Manager provides feedback to the Builder component to reduce the number of conflicts. We store the commit rate, and the conflicting scenarios in the Knowledge Repository to be used later for further reconstruction. When the commit rate reaches a minimum preconfigured rate, the Builder is invoked. Conflicting traces are combined into a single trace. This requires changes to the control instructions (e.g., branching conditions) to maintain the original execution flow. The newly reconstructed version is recompiled and loaded as a *new class definition* at runtime.

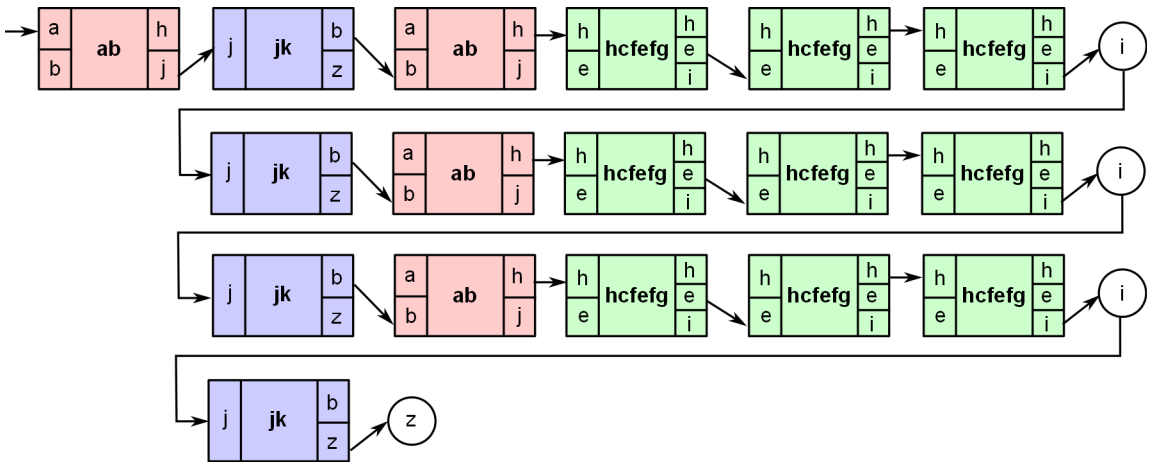


Figure 4.7: 3x3 Matrix Multiplication Execution using Traces Generated by profiling 2x2 Matrix Multiplication

4.4.5 Misprofiling

Profiling depends mainly on the program input. This input may not reflect some runtime aspects of the program flow (e.g., loops limits, biased branches). To illustrate this, we return to the matrix multiplication example in Figure 4.5a. Based on the profiling using 2x2 matrices, we construct the execution graph shown in Figure 4.5b. Now, recall our three traces ab, $hcfefg$, and jk, and assume we need to run this code for matrices 2x3 and 3x2. The Collector will assign jobs to the Executor, but upon the execution of the trace jk, the Executor will find that the code exits after j and needs to execute bs. Hence, it will request the Collector to schedule the job ab, with an entry to basic block b, in the incoming job set. Doing so allows us to extend the flow to cover more iterations. Note that the entry point must be sent to the synthetic method that represents the trace, as it should be able to start from any of its basic blocks (e.g., ab will start from b not a).

In Figure 4.7, traces are represented by blocks with their entries points on the left side, and exits on the right. The figure describes the execution using the traces extracted by profiling 2x2 matrix (See example at Section 4.4.1).

4.5 Implementation

4.5.1 Detecting Real Memory Dependencies

Recall that we use bytecode as the input, and concurrency refactoring is done entirely at the VM level. Compiler optimizations, such as register reductions and variable substitutions, increase the difficulty in detecting memory dependencies at the bytecode-level. For example,

```
y = 1                    y1 = 1
y += 2                   y2 = y1 + 2
x = y / 2                x1 = y2 / 2
```

Figure 4.8: Static Single Assignment form Example

two independent basic blocks in the source code may share the same set of local variables or loop counters in the bytecode. To overcome this problem, we transform the bytecode into the Static Single Assignment form (SSA) [15]. The SSA form guarantees that each local variable has a single static point of definition and is assigned exactly once, which significantly simplifies analysis. Figure 4.8 shows an example of the SSA form.

Using the SSA form, we inspect assignment statements, which reflect memory operations required by the basic block. At the end of each basic block, we generate a call to a *hydra_touch* operation that notifies the VM about the variables that were accessed in that basic block. In the second phase of profiling, we record the execution paths and the memory accessed by those paths. We then package each set of basic blocks in a trace. Traces should not be conflicting and access the same memory objects. However, it is possible to have such conflicts since our analysis uses information from past execution (which could be different from the current execution). We intentionally designed the data dependency algorithm to ignore some questionable data dependencies (e.g., loop index). This gives more opportunities for parallelization since if at run time a questionable dependency occurs, then TM will detect and handle it. Otherwise, such blocks will run in parallel and greater speedup is achieved.

4.5.2 Handing Irrevocable Code

Input and output instructions must be handled as a special case in the reconstruction and parallel execution as they cannot be rolled back. Traces with I/O instructions are therefore marked for special handling. The Collector never schedules such marked traces unless they are reachable – i.e., they cannot be run in parallel with their preceding traces. However, they can be run in parallel with their successor traces. This implicitly ensures that at most one I/O trace executes (i.e., only a single job of this trace runs at a time).

4.5.3 Method Inlining

Method inlining is the insertion of the complete body of a method in every place that it is called. In HydraVM, method calls appear as basic blocks, and in the execution graph, they appear as nodes. Thus, inlining occurs automatically as a side effect of the reconstruction process. This eliminates the time overhead of invoking a method.

Another interesting issue is handling recursive calls. The execution graph for recursion will appear as a repeated sequence of basic blocks (e.g., *abababab*...). Similar to method-

inlining, we merge multiple levels of recursion into a single trace, which reduces the overhead of managing parameters over the heap. Thus, a recursive call under HydraVM will be formed as nested transactions with lower depth than the original recursive code.

4.5.4 ByteSTM

ByteSTM is STM that operates at the bytecode level, which yields the following benefits:

- Significant implementation flexibility in handling memory access at low-level (e.g., registers, thread stack) and for transparently manipulating bytecode instructions for transactional synchronization and recovery;
- Higher performance due to implementing all TM building blocks (e.g., versioning, conflict detection, contention management) at bytecode-level; and
- Easy integration with other modules of HydraVM (Section 4.4)

We modified the Jikes RVM to support TM by adding instructions, *xBegin* and *xCommit*, which are used to start and end a transaction, respectively. Each load and store inside a transaction is done transactionally: loads are recorded in a read signature and stores are sandboxed; stores are stored in a transaction-local storage, called the *write-set*. The address of any variable (accessible at the VM level) is added to the written signature. The read/write signature is represented using a Bloom filter [16] and used to detect read/write or write/write conflicts. This approach is more efficient than comparing transaction read-set and write-set of transactions, but it also increases false negatives. (With the correct signature size, the effect of false positives can be reduced – we do this.)

When a load is called inside a transaction, we first check the write-set to determine if this location has been written to before and if so, the value from the write-set is returned. Otherwise, the value is read from the memory and the address signature is added to the read signature. At commit time, the read signature and write the signature of concurrent transactions are compared, and if there is a conflict, the newer transaction is aborted and restarted again. If the validation shows no conflict, then the write-set is written to memory.

For a VM-level STM, greater optimizations are possible than that for non VM-level STMs (e.g., Deuce [74], DSTM2 [65]). At the VM level, data types do not matter; only their sizes do. This allows us to simplify the data structures used to handle transactions. One through eight-byte data types is handled in the same way. Similarly, all different data addressing is reduced to absolute addressing. Primitives, objects, array elements, and statics are handled differently inside the VM, but they are translated into an absolute address and a specific size in bytes. This simplifies and speeds-up the write-back process, since we only care about writing back some bytes at a specific address. This allows us to work at the field level and at the array element level, which significantly reduces conflicts: if two transactions use the

same object, but each use a different field inside the object, then no conflict occurs (similarly for arrays).

Another optimization is the ability to avoid the VM's garbage collector (GC). GC can reduce STM performance when attempting to free unused objects. Also, dynamically allocating new memory to be used by STM is costly. ByteSTM disables the GC for the memory used for the internal data structures that support STM, we statically allocate memory for STM, handle it without interruption from the GC, and manually recycle it. The new memory is allocated if there is a memory overflow. Note that if a hybrid TM is implemented in Java, then it must be implemented inside the VM. Otherwise, hybrid TM will violate invariants of internal data structures used inside the VM, leading to inconsistencies.

We also inline the STM code inside the load and store instructions and the newly added instructions $xBegin$ and $xCommit$. Thus, there is no overhead in calling the STM procedures in ByteSTM.

Each *job* has an order that represents its logical order in the sequential execution of the original program. To preserve the data consistency between jobs, STM must be modified to support this ordering. Thus, in ByteSTM, when a conflict is detected between two jobs, we abort the one with the higher order. Also, when a block with a higher order tries to commit, we force it to sleep until its order is reached. ByteSTM commits the block if no conflict is detected.

When attempting to commit, each transaction checks its order against the expected order. If they are the same, the transaction proceeds, validates its read-set, commit its write-set, and updates the expected order. Otherwise, it sleeps and waits for its turn. The validation is done by scanning the thread stack and registers, and collecting the accessed objects' addresses. Objects IDs are retrieved from the object copies and used to create a *transaction signature*, which represents the memory addresses accessed by the transaction. Transactional conflicts are detected using the intersection of transaction signatures. After committing, each thread checks if the next thread is waiting for its turn to commit, and if so, that thread is woken up. Thus, ByteSTM keeps track of the expected order and handles commit in a decentralized manner.

4.5.5 Parallelizing Nested Loops

Nested loops introduce a challenge for parallelization, as it is difficult to parallelize both inner and outer loops and imposes complexity to the system design. In HydraVM, we handle nested loops as nested transactions using the closed-nesting model [95]: aborting a parent transaction aborts all its inner transactions, but not vice versa, and changes made by inner transactions become visible to their parents when they commit, but those changes are hidden from outside world till the highest level parent's commit.

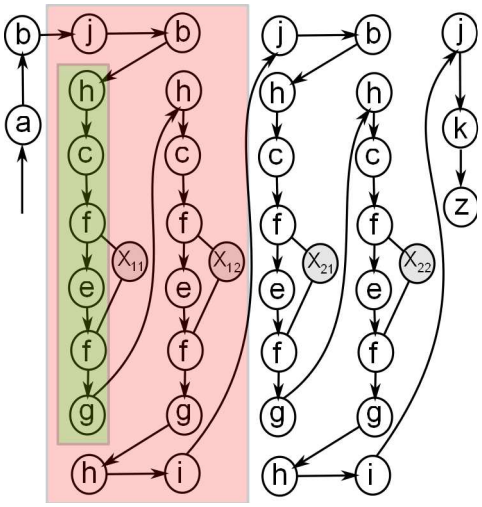


Figure 4.9: Nested Traces

1. Inner transactions share the read-set/write-set of their parent transactions;
2. Changes made by inner transactions become visible to their parent transactions when the inner transactions commit, but they are hidden from the outside world till the commit of the highest level parent;
3. Aborting an inner transaction aborts only its changes (not other sibling transactions, and not its parent transaction);
4. Aborting a parent transaction aborts all its inner transactions;
5. Inner transactions may conflict with each other and also with other, non-parent, higher-level transactions; and
6. Inner transactions are mutually ordered; i.e., their ages are relative to the first inner transaction of their parent. When an inner transaction conflicts with another inner transaction of a different parent, the ages of parent transactions are compared.

The use of closed nesting, instead other models such as linear nesting [95], is twofold:

1. Inner transactions run concurrently; conflict is resolved by aborting the higher age transaction.
2. We leverage the partial rollback of inner-transactions without affecting the parent or sibling transactions.

Although open nesting model [94] allows further increases in concurrency, open nesting contradicts our ordering model. In open nesting, inner transactions are permitted to commit

	Configurations
Processor	AMD Opteron Processor
CPU Cores	8
Clock Speed	800 MHz
L1	64 KB
L2	512 KB
L3	5 MB
Memory	12 GB
OS	Ubuntu 10.04, Linux

Table 4.1: Testbed and platform parameters for HydraVM experiments.

before its parent transaction, and if the parent transaction was aborted it uses a compensating action to revert the changes done by its committed inner transactions. However, in our model allowing inner transactions to commit will violate the ordering rule (lower transactions commit first), and preventing it from commit (i.e., wait until its chronological order) will cancel the idea behind open nesting.

Consider our earlier matrix multiplication example (See Section 4.4). From the execution string, $ab(jb(hcfefg)^2hi)^2jk$, we can create two nested traces: an outer trace $jb(hcfefg)^2hi$, and an inner trace $hcfefg$ (See Figure 4.9). The outer trace runs within a transaction, executing $jbhi$, that invokes a set of inner transactions $hcfefg$ after the execution of the basic block b.

4.6 Experimental Evaluation

Benchmarks. To evaluate HydraVM, we used five applications as benchmarks. These include a matrix multiplication application and four applications from the JOlden benchmark suite [23]: minimum spanning tree (MST), tree add (TreeAdd), traveling salesman (TSP), and bitonic sort (BiSort). The applications are written as sequential applications, though they exhibit data-level parallelism.

Testbed. We conducted our experiments on an 8-core multicore machine. Each core is an 800 MHz AMD Opteron Processor, with 64 KB L1 data cache, 512 KB L2 data cache, and 5 MB L3 data cache. The machine ran Ubuntu Linux.

Evaluation. Table 4.2 shows the result of the Profiler analysis on the benchmarks. The table shows the number of basic blocks, traces, and the average number of instructions per basic block. The lower part of the table shows the number of executed jobs by the Executor, and the maximum level of nesting during the experiments.

Using our techniques, we manage to split the sequential implementation of the benchmarks

Table 4.2: Profiler Analysis on Benchmarks

Benchmark	Matrix	TSP	BiSort	MST	TreeAdd
Avg. Instr. per BB.	4.29	4.2	4.75	3.7	4.1
Basic Blocks	31	77	24	52	10
Traces	3	12	5	3	4
Jobs	1001	1365	1023	12241	8195
Max Nesting	2	5	2	1	3

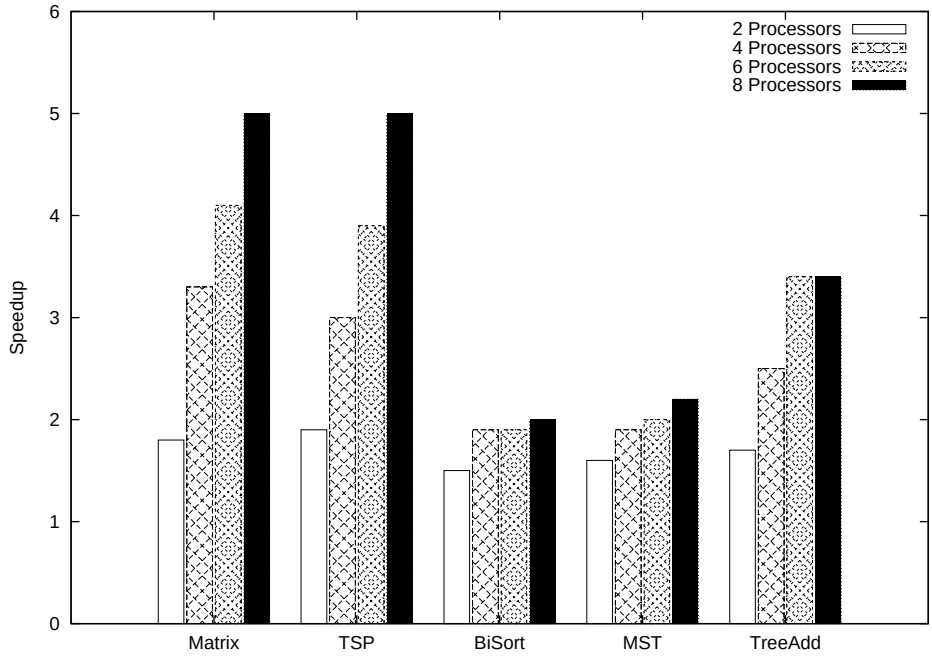


Figure 4.10: HydraVM Speedup

into parallel jobs that exploit the data-level parallelism. Figure 4.10 shows the speedup obtained for different number of processors. For matrix multiplication, HydraVM reconstructs the outer two loops into nested transactions, while the inner-most loop is formed as a trace because of the iteration dependencies. In TSP, BiSort, and TreeAdd, each multiple level of the recursive call is inlined into a single trace. For the MST benchmark, each iteration over the graph adds a new node to the MST, which creates inter-dependencies between iterations. However, updating the costs from the constructed MST and other nodes presents a good parallelization opportunity for HydraVM.

4.7 Discussion

We presented HydraVM, a JVM that automatically refactors concurrency in Java programs at the bytecode level. HydraVM extracts control-level parallelism by reconstructing program as independent traces. Loops, as a special case of traces, is included into the reconstruction procedure which supports data-level parallelism. Although, our proposal targets extracting code traces, most of the applications spend most of time executing loops. Loops have interesting features such as: symmetric code blocks, data level parallelism, recurrences, and induction variables. In the next chapter, we focus on loops as a unit for parallelization.

Online profiling and adaptive compilation provide transparent execution of the programs. Nevertheless, it adds an overhead to the runtime. Such overhead is non-negligible for short life applications, or ones with unpredictable execution paths. Static analysis of the code could be beneficial for providing an initial guess of hot-spot regions of the code, and build dependency relations between code-blocks. Adding a pre-execution static analysis and employing data dependency analysis could enhance the code generation and reduce transactional overhead, we followed this approach in Lerna.

Chapter 5

Lerna

In this chapter, we present Lerna, a system that automatically and transparently detects and extracts parallelism from sequential code. Lerna is cross-platform and independent of the programming language, and does not require the analysis of the application's source code, it simply takes its intermediate representation compiled using *LLVM* [80], the well-known compiler infrastructure, as input and produces ready-to-run parallel code as output[1], thus finding its best fit with (legacy) sequential applications. This approach makes Lerna independent also of the specific hardware used.

Similar to HydraVM, the parallel execution exploits memory transactions to manage concurrent and out-of-order memory accesses. While, HydraVM presents an initial concept of a virtual machine that exploits in-memory transactions to parallelize traces, Lerna focus on parallelizing loops, which usually contains the hotspot sections in many applications. Recall that HydraVM reconstructs the code at runtime through recompilation and reloading class definition, and it is obligated to run the application through the virtual machine. Unlike HydraVM, Lerna does not change the code at runtime through recompilation, which enable us to compile the code to its native representation. Nevertheless, Lerna supports adaptive execution of transformed programs through changing some key perfomance parameters of its runtime library (See Section 5.7). Finally, the extensive profiling phase at HydraVM relies on establishing a relation between basic blocks and their accessed memory addresses, which limits its usage to small size applications. In Lerna, we replaced that with a static alias analysis phase combined with light offline profiling step that complement our static analysis.

Lerna is a complete system, which overcome all the above limitations and embeds system and algorithmic innovations to provide high performance. Our experimental study involves the transformation of 10 sequential applications into parallel. Results showed an average of 2.7× speedup for micro-benchmarks and 2.5× for the macro-benchmarks.

[1] Lerna supports the generation of native executable as output.

5.1 Challenges

Despite the high-level goal showed above, without fine-grain optimizations and innovations, deploying TM-style transactions to blocks of code that are prone to be parallelized leads the application performance to be slower (often much slower) than the sequential, non-instrumented execution. As an example of that, a blind parallelization of a loop (Figure 5.1a) would mean wrapping the whole body of the loop within a transaction (Figure 5.1b). By doing so, we let all transactions conflict with each other on, at least, the increment of the variable c or i[2]. In addition: variables that have been never modified within the loop may be transactionally accessed; the transaction commit order should be the same as the completion order of the iterations if they would have executed sequentially; 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 pay just the overhead of the transactional instrumentation and, as a consequence, providing performance slower than sequential execution.

Lerna does not suffer from the above issues (as showed in Figure 5.1c). 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 tuning. The first include: loop simplification, induction variable reduction, removal of non-transactional work from the context to restore after a conflict, exploitation of the symmetry of executed parallel code, and an optimized in-order commit of transactions. Regarding the latter, Lerna provides an adaptive runtime layer to improve the performance of the parallel execution. This layer is fine-tuned by collecting feedbacks from the actual execution in order to capture the best settings of the key performance parameters that most influence the effectiveness of the parallelization (e.g., number of worker threads, size and number of parallel jobs).

The results are impressive. We evaluated Lerna's performance using a set of 10 applications including micro-benchmarks from the RSTM [2] framework, STAMP [25], a suite of sequential applications designed for evaluating in-memory concurrency controls, and Fluidanimate [96], an application performing physics simulations. The reason we selected them is because they provide (except for Fluidanimate) also a performance upper-bound for Lerna. In fact, they are released with a version that provides synchronization by using manually defined, and optimized, 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 pick of 3.9×), and 2.5× faster considering macro-benchmarks (with a top speedup of one order of magnitude reached with STAMP).

Lerna is the first self-contained, completely automated and transparent system that makes sequential applications parallel. Thanks to an efficient use of transactional blocks, it finds

[2]Libraries that require programmer interaction, as OpenMP, already offer programming primitives to handle loops increments.

```
c = min;
while(i < max){
        i++;
        c = c + 5;
        ... local processing work ...
        if(i < j)
                k = k + c;
}
```

(a) Loop with data dependency

```
c = min;
while(i < max){
    atomic{
       TX_WRITE(i, TX_READ(i) + 1);
       TX_WRITE(c, TX_READ(c) + 5);
       ... local processing work ...
       if(TX_READ(i) < TX_READ(j))
           TX_WRITE(k,
             TX_READ(k) + TX_READ(c));
    }
}
```

(b) Loop with atomic body

```
c = min;
while(i < max){
    i++;
    parallel(i){
        c = min + i*5;
        ... local processing work ...
        atomic{
        if(i < j)
            TX_INCREMENT(k, c);
        }
    }
}
```

(c) Loop with parallelized body

Figure 5.1: Lerna's Loop Transformation: from Sequential to Parallel.

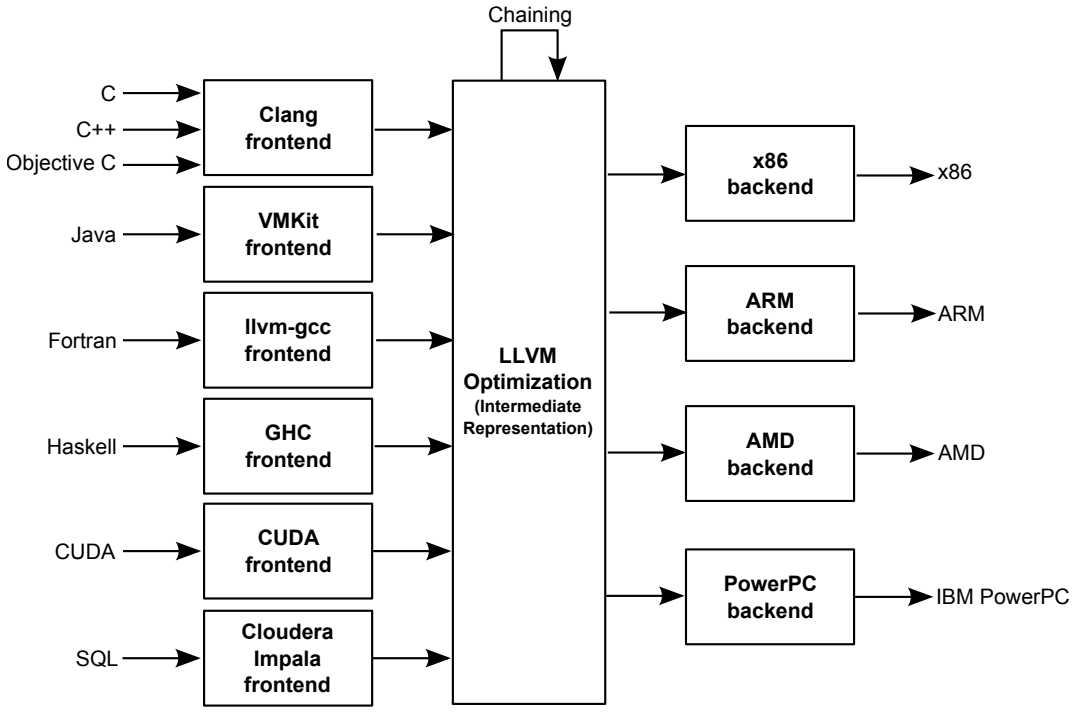


Figure 5.2: LLVM Three Layers Design

its sweet spot with applications involving data sharing but not simply those with partitioned memory access.

5.2 Low-Level Virtual Machine

Low-Level Virtual Machine (LLVM) is a modular and reusable collection of libraries, with well-defined interfaces, that define a complete compiler infrastructure. It represents a middle layer between front-end language-specific compilers, and back-end instruction sets generators. LLVM offers an intermediate representation (IR), bytecode, that can be optimized and transformed into more *efficient* IR. Optimizers can be chained to provide multi-level of optimizations at different levels of scopes (modules, call graph, function, loop, region, and basic block). LLVM supports a language-independent instruction set and data types in the form of Single Static Assignment (SSA).

Such separation of layers help developers to write front-ends to get use of underlying compiler optimizations. At the current stage, LLVM supports most of the widely used languages such as: C, C++, Objective-C, Java, Fortran, Haskell, and Ruby – the full list at [3]. The most common LLVM front-end is Clang which support compiling C, Objective-C and C++ codes. CLange is supported by Apple as a replacement for C/Objective-C compiler in the GCC system. Likewise, several platforms including: x86/x86-64, AMD, ARM, SPARC, MIPS,

and Nvidia, have LLVM back-end generators for its instruction sets – See Figure 5.2. These factors together builds a large community for LLVM as a popular compiler infrastructure. With LLVM, researchers build their optimizations on the intermediate layer using LLVM byte code, and have their optimizations available for large set of languages/platforms.

5.3 General Architecture and Workflow

Lerna is deployed as a container of:

- an *automated software tool* that performs a set of transformations and analysis steps (also called *passes* in accordance with the terminology used by LLVM) that run on the LLVM intermediate representation of the original binary application code. It produces a refactored multi-threaded version of the input program that can run efficiently on multiprocessor architectures;
- 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 (adaptively) the most effective number of worker threads according to the actual setup of the runtime environment, and based on the feedbacks collected from the online execution; *3)* scheduling jobs to threads according to threads' characteristics (e.g., stack size, priority); and *4)* performing memory housekeeping and releasing computational resources.

Figure 5.3 shows the architecture and the workflow of Lerna. Lerna operates at the LLVM intermediate representation of the input program, thus it does not require the application to be written in any specific programming language. However, Lerna's design does not preclude the programmer from providing hints that can be leveraged to make the refactoring process more effective. In this work we provide the fully automated process without considering any programmer intervention, and we discuss how to exploit the prior application knowledge in Section 5.7. 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) later used 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 bytecode (or

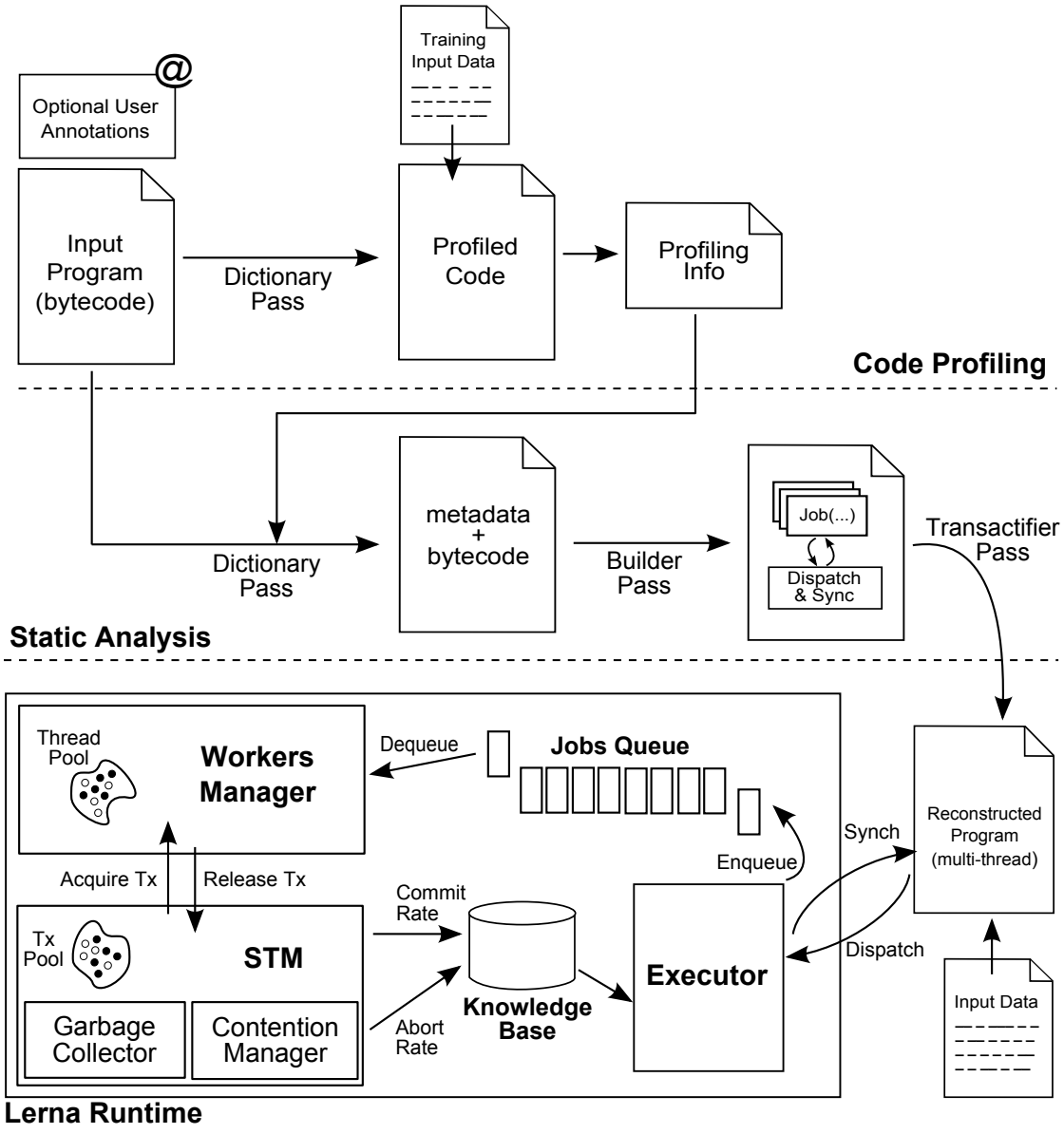


Figure 5.3: Lerna's Architecture and Workflow

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., where the transaction begins and ends).

- *Transactifier Pass*. It applies the alias analysis [35] (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 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.

- *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.

5.4 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 flow (an example of such a graph is shown in Figure 5.5). Basic blocks are easily determined from the bytecode (see Figure 5.4).

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 bytecode 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 bytecode*.

5.5 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 bytecode) is in the static single assignment (SSA) form. With SSA, each variable is defined before it is used, and it is assigned exactly once. Figure 5.4 shows LLVM intermediate representation of the loop Figure 5.1a. The code at Figure 5.4 is in SSA form, and is divided into sets of *basic blocks*. Each basic block starts with an optional label; it is composed of LLVM assembly instructions; and it ends with a branching instruction (*terminator*).

5.5.1 Dictionary Pass

In the dictionary pass, a full bytecode 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:

- It is revocable (e.g., it does not perform input/output operations);
- It does not affect the state of the program; and

```
entry:
  %retval = alloca i32, align 4
  br label %while.cond
while.cond:
; preds = %if.end, %entry
  %0 = load i32* %i, align 4
  %1 = load i32* %max, align 4
  %cmp = icmp slt i32 %0, %1
  br i1 %cmp, label %while.body, label %while.end
while.body:                                       ; preds = %while.cond
  %2 = load i32* %i, align 4
  %inc = add nsw i32 %2, 1
  store i32 %inc, i32* %i, align 4
  call void @_Z19do_local_processingv()
  %3 = load i32* %i, align 4
  %4 = load i32* %j, align 4
  %cmp1 = icmp slt i32 %3, %4
  br i1 %cmp1, label %if.then, label %if.end
if.then:                                          ; preds = %while.body
  %5 = load i32* %k, align 4
  %add = add nsw i32 %5, 1
  store i32 %add, i32* %k, align 4
  br label %if.end
if.end:
; preds = %if.then, %while.body
  br label %while.cond
while.end:                                        ; preds = %while.cond
  %6 = load i32* %retval
  ret i32 %6
```

Figure 5.4: The LLVM Intermediate Representation using SSA form of Figure 5.1a.

- It is thread safe.

A common example of safe calls are random generators, or mathematical basic functions such as trigonometric functions.

5.5.2 Builder Pass

This pass is one of the core steps made by the refactoring process because it takes the code to transform (as 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 thesis 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. In Figure 5.4, the *entry* block is the loop pre-header, while its header is *while.cond*. The loop has one latch (i.e., *if.end*), and a single exit *while.end* (from *while.cond*). The loop body is the set of blocks *while.body*, *if.then* and *if.end*. 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. Examples of natural and simple loops are reported in Figures 5.5 (a) and (b), respectively. In Figure 5.5 (a), the loop header has two types of predecessors, external basic blocks from outside of the loop, and one of the body latches. Putting this loop in its simple form requires adding: *i)* a single pre-header and changing the external predecessors to jump to the pre-header; and *ii)* an intermediate basic block to isolate the second latch from the header.

- *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 one exists, is often an induction variable, and the loop can contain more than one induction variable. The

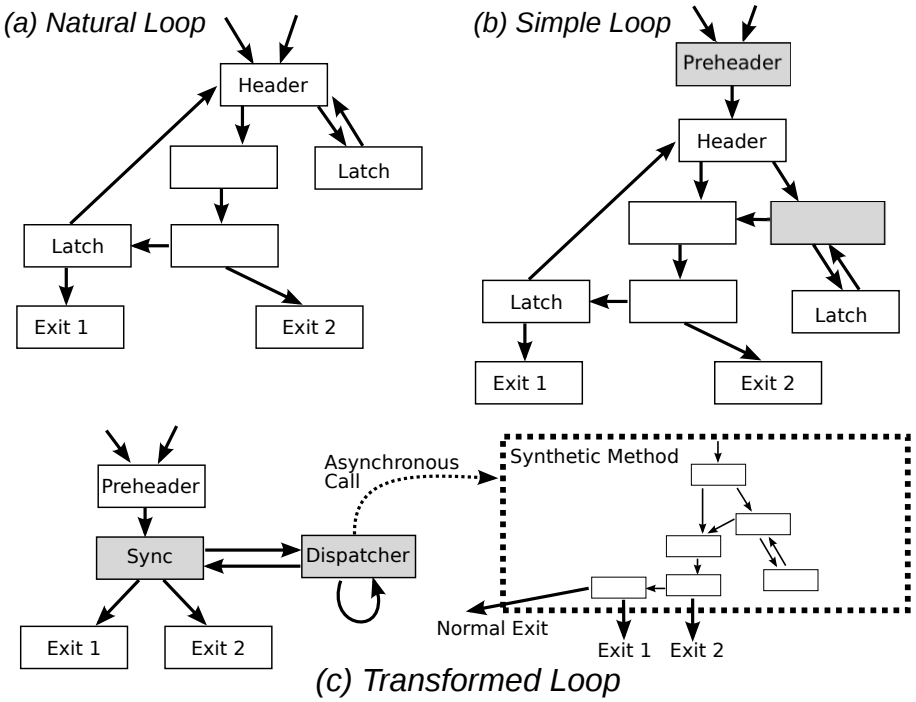


Figure 5.5: Natural, Simple and Transformed Loop

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 to the method.

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, named *lerna_dispatch*, providing it with the address of the

Tx1	Tx2
Y[1] := 0	Y[2] := 0
Z[1] := X[1]	Z[2] := X[2]

(a) Symmetric Transactions

Tx1	Tx2
Y[1] := 0	
	X[1] := Y[1]
Z[1] := X[1]	

(b) Normal Transactions

Figure 5.6: Symmetric vs Normal Transactions

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. Figure 5.5 (c) shows the control flow diagram (CFG) for the loop before and after transformation.

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 way the code is written.

5.5.3 Transactifier Pass

After turning the bytecode 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 for preserving the correctness of the transaction, which iterates over the read-set, is the primary source of TM's overhead. Several hardware prototypes have been proposed to enhance TM read-set management [26, 116]. Alternatively, in our work the 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 [26, 116]. In fact, reading an address transactionally requires: *1)* checking if the address has already been written before (i.e., check the write-set); *2)* adding the address to the read-set; and *3)* returning the address 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 number of resize operations.
- 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. Figure 5.1 shows an example of symmetric transactions. We take advantage of this characteristic by 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, Figure 5.6a, then the load does not need to be performed transactionally. In Figure 5.1, concurrent transactions execute symmetric iterations of the loop. Although j is read within the transactions, we do not have to read it transactionally as it can never be changed by any of the transactions produced from the same loop (thus the only allowed to run concurrently). Note that this technique is specific for parallelizing loops and cannot be applied to the normal transaction processing where all transactions do not necessarily execute the same code as for the symmetric transactions.

In contrast, in Figure 5.6b we show an example of two concurrent non-symmetric transactions, thus executing different transaction bodies. Also in this case each transaction has a load operation that does not alias with other stores but when the two transactions (with different code paths) run concurrently they can produce wrong result. This is because these transactions are not symmetric and thus the read must be done transactionally.

Transactions may contain calls to other functions. As these functions may manipulate memory locations, they must be handled. Whenever possible, we try to inline the called functions; otherwise we create a transactional version of the function called within a transaction. In the latter 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.

5.6 Transactional Execution

Transactional memory algorithms differ in the memory versioning techniques, i.e., undo-log or write-buffer, and in the conflict detection, i.e., lazy or eager. In write-buffer algorithms, transactional loads are recorded in a read-set and stores are and-boxed, which means that each store is not written to the original address but it is kept into a local storage called the write-set until commit. When a load is called inside a transaction, the write-set is first checked to determine if this location has been written before and, if so, the value from the write-set is returned. Otherwise, the value is read from the memory and the address is added to the read-set. At commit time, the read-set is validated to make sure that it is still consistent. If the validation shows no conflict, then the write-set is written back to the shared memory and the changes become visible to all. On the other hand, undo-log algorithms expose memory changes to the main memory right after the transactional write, and they keep the old values in a local log to be restored upon transaction abort.

Contention between transactions occur when two transactions access the same address and one of them is a writer. Eager contention detection is done by associating a lock with each memory address (a lock can cover multiple addresses). A transaction acquires locks on its written addresses at encounter time. Conflicts are detected when a transaction tries to access a locked address. Alternatively, in lazy contention detection, each address has a version record. Transaction stores the version of its read addresses. Succeeded transactions modify the version of their written addresses at commit time. Transaction is aborted when it finds a different version number than the one recorded. Validation of memory addresses accessed transactionally is done either by performing addresses comparison [43], or by checking if the values have changed [38].

Table 5.1 shows different design choices of some known transactional memory implementations.

Our proposal is decoupled from the specific TM implementation used but it requires in-order

		Version	
		Eager	Lazy
Contention	Eager	TinySTM [50], LogTM [92] UTM [6], McRT-STM [115]	LTM [6], TinySTM [50], SwissTM [45]
	Lazy		TL2 [43], TCC [57], NOrec [38]

Table 5.1: Transactional Memory Design Choices

commit. To allow the integration of further TMs, we identified the following requirements needed to support ordering:

Single Committer. At any time only one thread is allowed to commit its transaction(s). This thread is the one executing the transaction with lowest age. While a thread is committing, other threads can proceed by speculatively executing next transactions, or wait until the committed completes. Allowing threads to proceed their execution is risky because it can increase the contention probability (the life of an uncommitted transaction enlarges), so this speculation must be limited by a certain, predefined threshold.

Age-based Contention Management (CM). Algorithms with eager conflict detection (i.e., at encounter time) should favor transactions with lower age (i.e., that encapsulate older iterations), 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. Note that, for value-based validation with eager versioning, it is possible for earlier transactions to wrongly commit after it read from a speculative iteration. To solve this, during the commit phase, the read-set must be compared with the write-sets of completed transactions.

Memory Versioning. For eager versioning, if the implementation uses eager CM, then it prevents speculative iterations from affecting older iteration because they will collide. On the other hand, Lazy CM may cause earlier iterations to read from speculative iterations. However, thanks to the single committer, the former iteration will detect the conflict and both transactions will be aborted. Lazy versioning implementations hide their changes from other transactions, thus no modification is required for this category.

5.6.1 Ordered NOrec

In the current implementation we used NOrec [38] as a TM library. It is an algorithm that offers low memory access overhead with 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. A characteristic of this algorithm is that it permits a single committing writer at a time, which is in general not desirable but it matches the need of Lerna's concurrency control: having a single committer (See Section 5.6). For this reason we decided to rely on

NOrec as the default STM implementation for Lerna because of its small memory footprint and because it matches our ordering conditions (we need a single committer as limitation at NOrec, it is one of the requirement for ordering transactions). Our modified version of NOrec manages which transaction should be the single committer according to the chronological order (i.e., age).

5.6.2 Ordered TinySTM

TinySTM algorithm uses encounter time locking (ETL) and comes with two memory access strategies: write-through and write-back. TinySTM uses timestamps for transactions to ensure a consistent view of memory; we exploit this timestamp to represent the *age* of transactions. Using an aggressive age-based contention manager, transactions can be ordered according to the chronological order of their executing code. With TinySTM, transactions conflict at access time; this allows early detection of conflicting iterations. The choice of write-through or write-back strategy is workload specific; at low-contention write-through provides a fast path execution, while write-back is more suitable for high-contention as it exhibits lower overhead abort procedure.

5.6.3 Irrevocable Transactions

A transaction performs a read-set validation at commit time to preserve correctness. That is needed 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. That is because Tx_n is now the highest priority transaction at this time, so no other transaction can commit its changes to the memory. We do that by flagging Tx_n as an *irrevocable transaction*. Similarly, 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.

This optimization reduces the commit time by just writing the write-set values to the memory.

5.6.4 Transactional Increment

Figure 5.7 illustrates a common situation, which is the *counter*. Loops with counters hamper parallelism because they create data dependencies between iterations, even non-consecutive iterations, which produces a large amount of conflicts. The induction variable substitution

```
for(int i=0;i<100;i++){
    ...
    if(some_condition)
        counter++;
    ...
}
```

(a) Conditional increments

```
while(proceed) {
    ...
    counter++;
    ...
}
```

(b) No induction variable

Figure 5.7: Conditional Counters

cannot produce a closed form (function) of the loop index (if it exists). If a variable is incremented (or decremented) based on any arbitrary condition and its value is used only after the loop completes the whole execution, then it is eligible for the *Transactional Increment* optimization.

In addition to the classical transactional read and write (*tx_read* and *tx_write*), we propose a new transactional primitive, the *transactional increment*, to enable the parallelization of loops with irreducible counters. This type of counter can be detected during our transformations. Within the transactional code, a store S_t is eligible for our optimization if it aliases only with one load L_d, and it writes a value that is based on the return value of L_d. The load, change, and store operations are replaced with a call to *tx_increment*, which receives the address and the value to increment. We propose two ways to implement *tx_increment*:

- Using an atomic increment to the variable, and storing the address to the transaction's meta-data. The atomic operation preserves data consistency; however, it affects the shared memory before the transaction commits. To address this issue, aborted transactions compensate all accessed counters by performing the same increment but with the inverse value.
- By storing the increments into thread-specific meta-data. At the end of each *Sync* operation, threads coordinate with each other to expose the aggregated per-thread increments of the counter. This method is appropriate for floating point variables, which cannot be updated atomically on commodity hardware.

Using this approach, transactions will not conflict on this address, and the correct value of the counter will be in memory after the completion of the loop (See Figure 5.1c).

5.7 Adaptive Runtime

The Adaptive Optimization System (AOS) [8] 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 worker threads) and by dynamically refining sections of code already parallelized statically according to the characteristics of the actual application execution.

Before presenting the optimizations made at runtime, we detail the component responsible for executing jobs (i.e., the Workers Manager in Figure 5.8). Jobs are evenly distributed over workers. Each worker thread keeps a local queue of its slice of dispatched jobs and a circular buffer of transaction descriptors. A worker 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.

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.

5.7.1 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. In our implementation, we use an exponentially increasing batch size. Initially, we dispatch a single job, which covers the common set of loops with zero iterations; if the loops are longer, then we increase the number of dispatched jobs exponentially until reaching a pre-configured threshold. Once a loop is entirely executed, we record the last batch size used so that, if the execution goes back on calling the same loop,

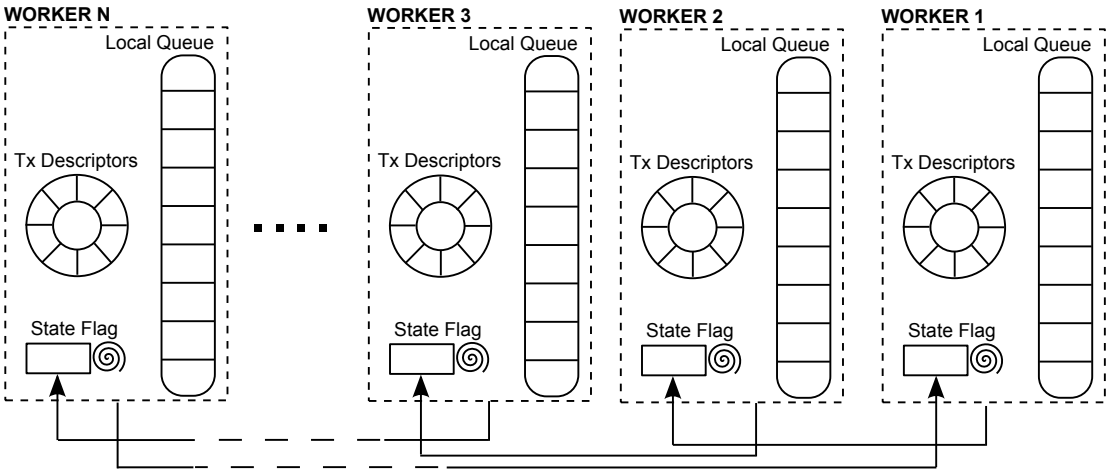


Figure 5.8: Workers Manager

we do not need to perform again the initial tuning.

5.7.2 Jobs Tiling and Partitioning

As explained in Section 5.5.2, 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. We tune tiling by taking into account the number of instructions per job, and commit rate of past executions using the *knowledge base*.

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. As a result, the job is partitioned into non-transactional work, which is now moved out of the transaction scope, and the transaction itself (See Figure 5.1c).

5.7.3 Workers Selection

Figure 5.3 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. The algorithm constructs a window of different worker counts, and iteratively improves it by changing the count of workers.

Using the throughput metric is better than relying on the commit rate. For example, three workers with an average commit rate equal to 50% are better than two workers with 70% average commit rate. In addition to that, parallel execution is subject to other factors such as bus contention, cache hits, and thread overhead.The throughput combines all of these factors, thus giving a global picture about the performance gain.

Finally, in some situations (e.g., high contention or very small transactions) it is better to use a single worker (sequentially). For that reason, if our heuristic decides to use only one worker, then we use the non-transactional version (as a fast path) of the synthetic method to avoid the unnecessary transaction overhead.

5.7.4 Manual Tuning

As stated early, Lerna is completely automated but it still allows the programmer to provide hints about the program to parallelize. In this section we present some of the manual configurations that can be done. These configurations are only applicable if the source code is available.

A *safe call* is a call to an external function defined as a library or a system call; such a call must be stateless and cannot affect the program result if repeated multiple times. Such calls cannot be detected using static analysis, so we rely on the user for defining a list of them. Our framework generates a histogram of calls that represents the number of excluded blocks from our transformation because of this call. Based on this histogram, user can decide which calls are more beneficial to be classified as a safe call.

The alias analysis techniques (see Section 5.5.3) help in detecting dependencies between loads and stores; however, in some situations (as documented in [35]) it produces conservative decisions, which limit the opportunities of 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 such a function 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,

	Configurations
Processor	2× Opteron 6168 processors
CPU Cores	12
Clock Speed	1.9 GHz
L1	128 KB
L2	512 KB
L3	12 MB
Memory	12 GB
OS	Ubuntu 10.04, Linux

Table 5.2: Testbed and platform parameters for Lerna experiments.

hence reducing the overall overhead.

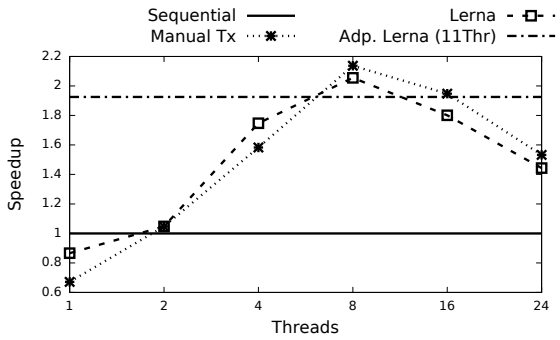
Section 5.4 explains how Lerna detects the eligible code for parallelization through the profiling phase. Alternatively, users can directly inform our *Builder Pass* of their recommendations for applying our analysis. Also, the programmer can exclude some sections of the code from being parallelized for some reason. We support a user-defined *exclude list* for portions of code that will be excluded from any transformation.

5.8 Evaluation

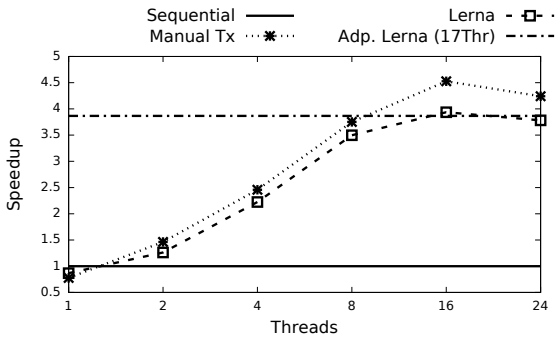
In this section we evaluate Lerna and measure the effect of the key performance parameters (e.g., job size, worker count, tiling) on the overall performance. Our evaluation involves a total of 13 applications grouped into micro-benchmarks and macro-benchmarks: STAMP [25] and PARSEC [96]. The micro-benchmarks allow us to tune the application workload in order to show strengths (and weaknesses) of our automated solution. The applications of the macro-benchmarks show the impact of Lerna in well-known workloads.

We compare the speedup of Lerna over the (original) sequential and the manual, optimized transactional version of the code (not available for PARSEC benchmarks). Note that the latter is not coded by us; it is released along with the benchmark itself, and is made manually 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. As a result, Lerna's performance goal is twofold: providing a substantial speedup over the sequential code, and to be as close as possible to the manual transactional version.

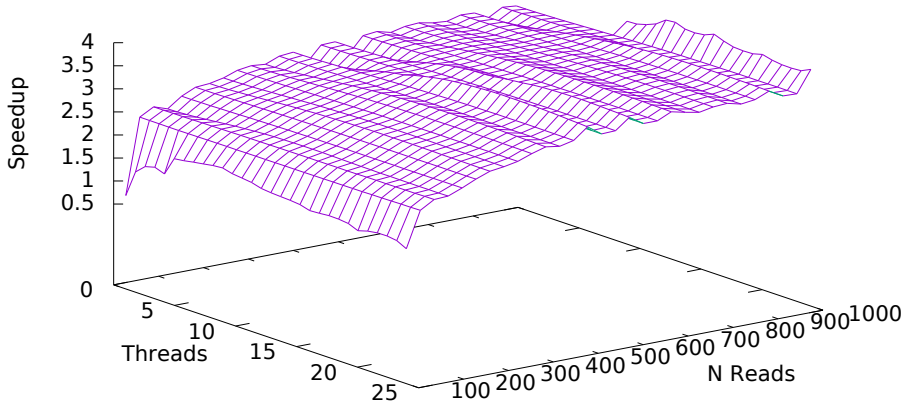
The testbed used in the evaluation consists of an AMD multicore machine equipped with 2 Opteron 6168 processors, each with 12-cores running at 1.9 GHz of clock speed. The total memory available is 12 GB and the cache sizes are 128 KB for the L1, 512 KB for the L2 and 12 MB for the L3. This machine represents well a current commodity hardware. On this machine, the overall refactoring process, from profiling to the generation of the binary, took ∼10s for micro-benchmarks and ∼40s for the macro-benchmarks.



(a) Read100Write1 - Speedup



(b) Read1000Write1 - Speedup



(c) Lerna Speedup

Figure 5.9: ReadNWrite1 Benchmark.

5.8.1 Micro-benchmarks

In our first set of experiments we consider the RSTM micro-benchmarks [2] to evaluate the effect of different workload characteristics, such as the amount of transactional operations per job, the job length, and the read/write ratio, on the overall performance.

We report the speedup over the sequential code by varying the number of threads used. The performance is measured for 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). As a general comment, we observe some small slow-down only when one thread is used; otherwise Lerna is usually very close to the manual transactional version. 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.

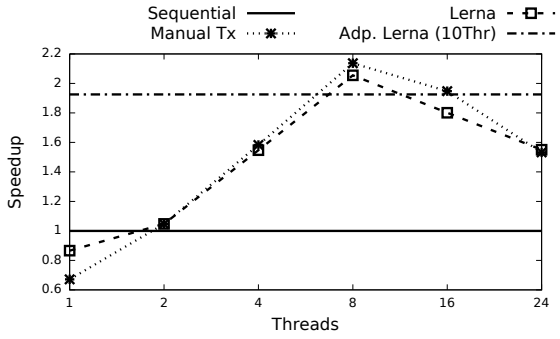
In *ReadNWrite1Bench* (Figure 5.9), transactions read N locations and write 1 location. Given that, the transaction write-set is very small, hence it implies a fast commit of a lazy TM algorithm as ours. The abort rate is low (0% in the experiments), and the transaction length is proportional to the read-set size. Figure 5.9c illustrates how the size of transaction read-set (with a small size write-set) affects the speedup. 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* (Figure 5.10), each transaction reads N locations, and then writes to another N locations. The large transaction write-set introduces a delay at commit time for lazy versioning TM algorithms, and increases the number of 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.

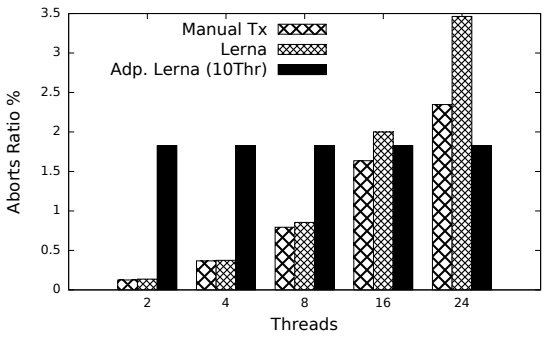
MCASBench performs a multi-word compare and swap, by reading and then writing N consecutive locations. Similarly to ReadWriteN, the write-set is large, but the abort probability is lower than before because each pair of read and write acts on the same location. Figure 5.11 illustrates the impact of increasing workers with long and short transactions. In Figure 5.11e, we see that with increasing the number of operations per transaction, the speedup degrades; besides threads contention, the number of aborts increases with increasing the number of accessed locations (See Figures 5.11b and 5.11d).

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. It worth noting that all micro-benchmarks does not perform many calculations on accessed memory locations, which represents a challenge for a TM-based approach.

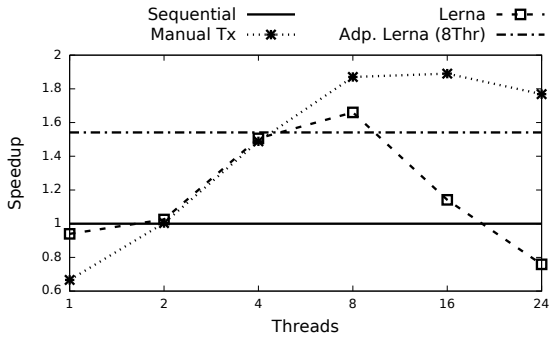
Figure 5.12 summarizes the behavior of the adaptive selection of the number of workers for the three micro-benchmarks and varying the size of the batch. The procedure starts by trying different worker counts within a fixed window (shown here, it is 7), then it picks the best worker according to the calculated throughput. Changing the worker counts shifts the window, thus allowing the technique to learn more and find the most effective settings for the current execution.



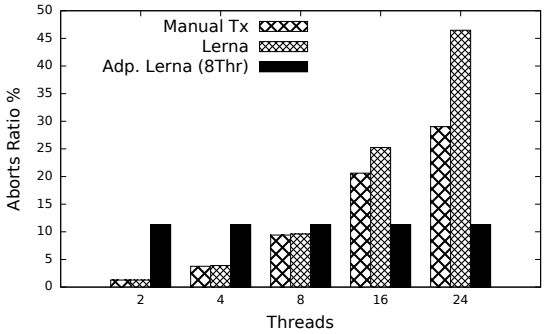
(a) ReadWrite100 - Speedup



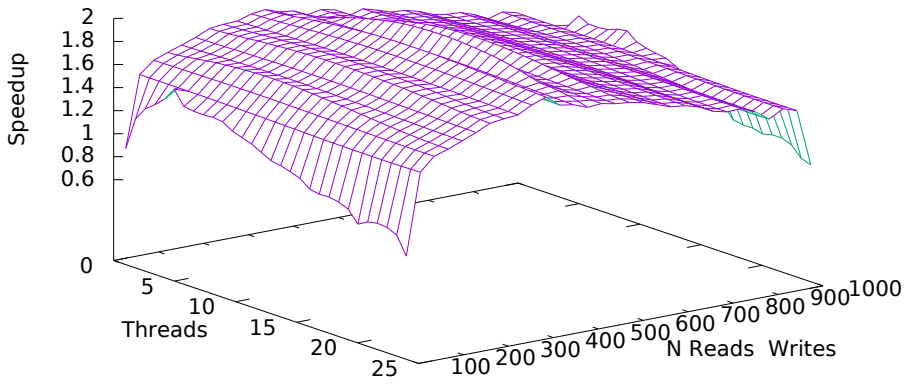
(b) ReadWrite100 - Aborts



(c) ReadWrite1000 - Speedup

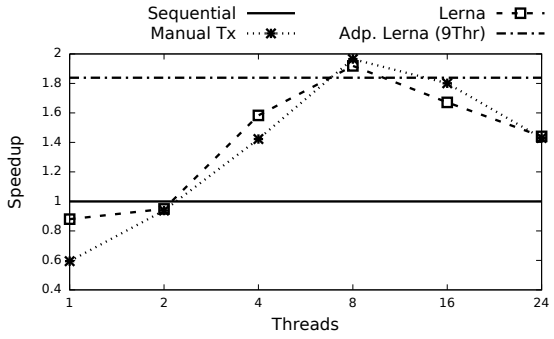


(d) ReadWrite1000 - Aborts

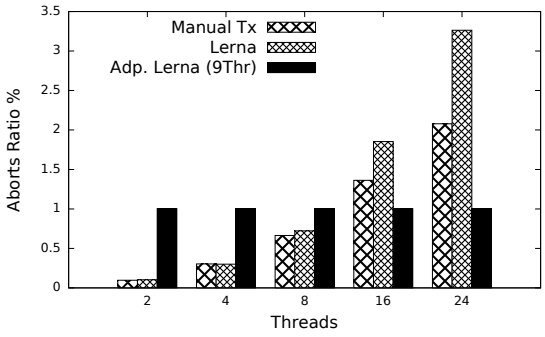


(e) Lerna Speedup

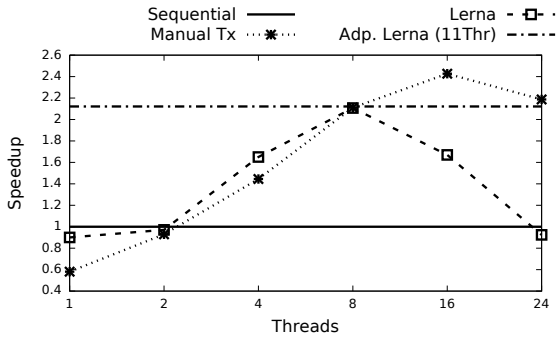
Figure 5.10: ReadWriteN Benchmark.



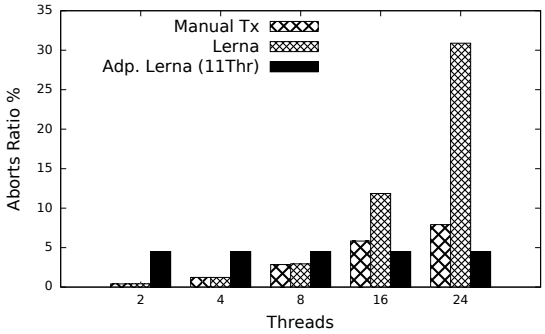
(a) MCAS100 - Speedup



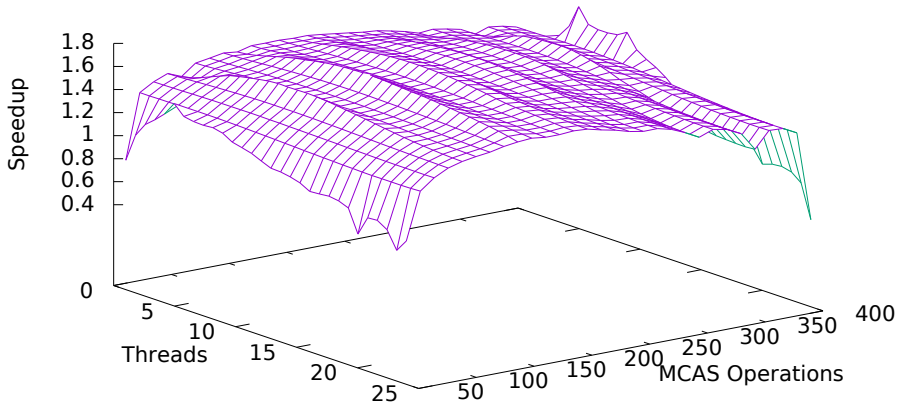
(b) MCAS100 - Aborts



(c) MCAS300 - Speedup



(d) MCAS300 - Aborts



(e) Lerna Speedup

Figure 5.11: MCAS Benchmark.

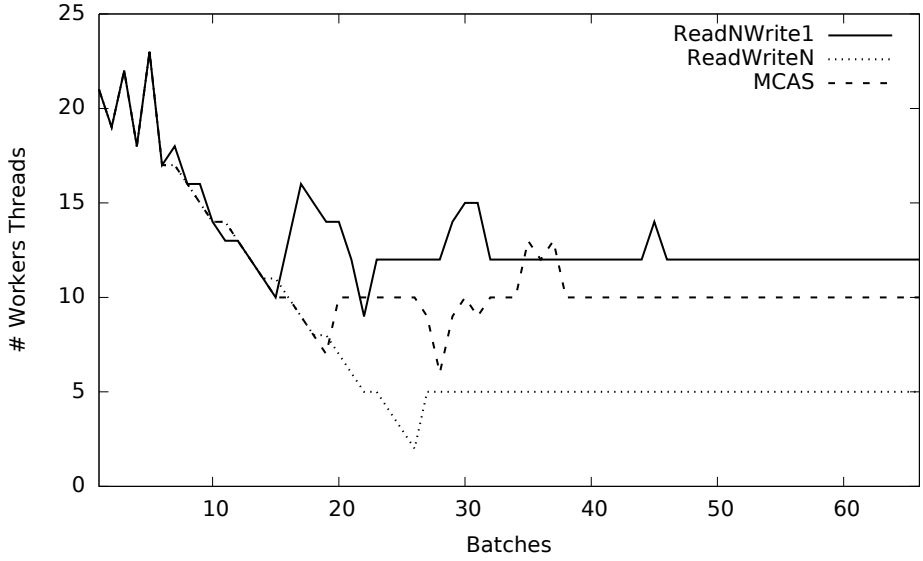


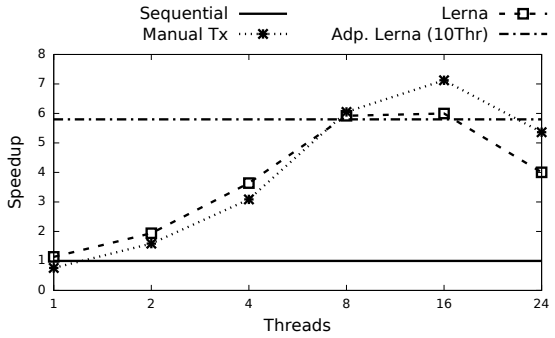
Figure 5.12: Adaptive workers selection

5.8.2 The STAMP Benchmark

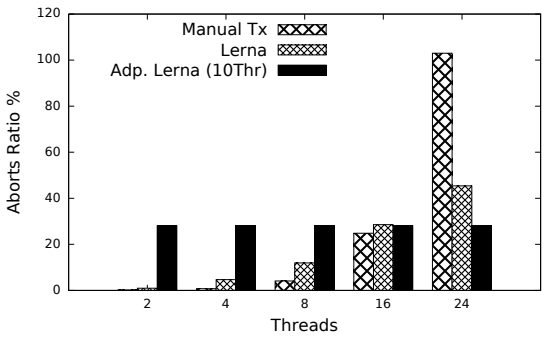
STAMP [25] is a comprehensive benchmark suite with eight applications that cover a variety of domains. Figures 5.13, 5.14 and 5.15 show the speedup of the Lerna's transformed code over sequential code, and against the manually transactified version of the application, which exploits unordered commit. While the main focus of our work is speedup over baseline sequential code, we highlight here the overheads and tradeoff with respect to a handcraft manual transformation aware of the underlying program semantics. Two applications (Yada and Bayes) have been excluded because they expose non-deterministic behaviors, thus their evolution is unpredictable when executed transactionally. Table 5.3 provides a summary of benchmarks and inputs used in the evaluation.

Kmeans, a clustering algorithm, iterates over a set of points and associate them to clusters. The main computation in finding nearest point, while shared data updates occur at the end of each iteration. Using job partitioning, Lerna achieves 6× and 1.6× speedup over the sequential version (See Figures 5.13a - 5.13d). The ordering introduces 25% delay compared to the unordered transactional version.

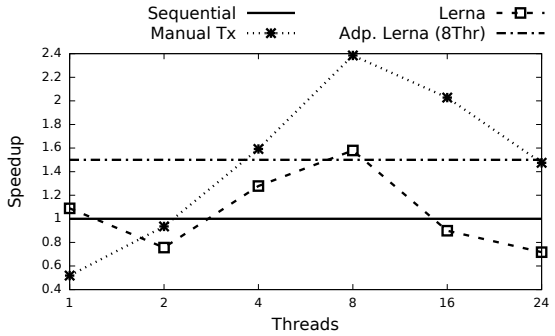
Genome, a gene sequencing program, reconstructs the gene sequence from segments of a larger gene. It uses a shared hash-table to organize the segments, which requires synchronization over its accesses. In Figures 5.13e - 5.13h, Lerna has 3× to 5.5× speedup over sequential. Ordering semantics is not a must for the hash-table insertion, which causes the manual Tx to perform 1.4× to 1.8× faster than Lerna, as transactions commit as soon as they end.



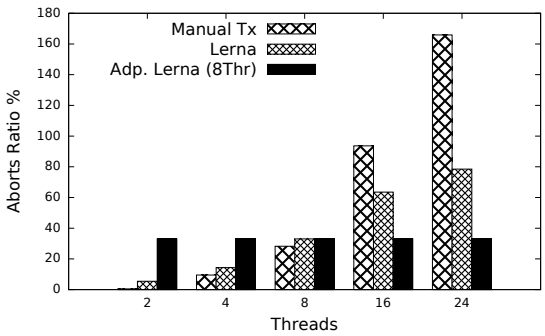
(a) Kmeans, Low Contention - Speedup



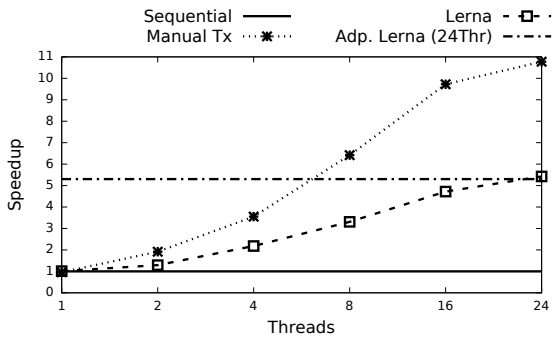
(b) Kmeans, Low Contention - Aborts



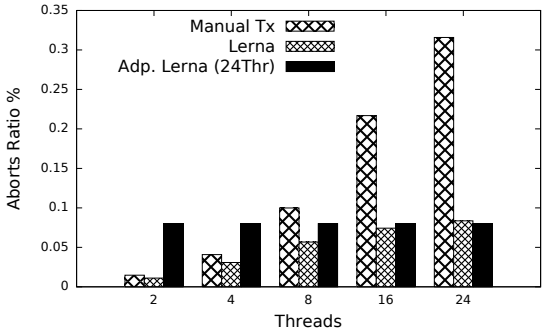
(c) Kmeans, High Contention - Speedup



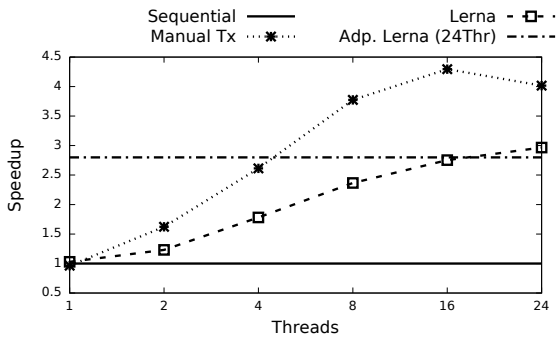
(d) Kmeans, High Contention - Aborts



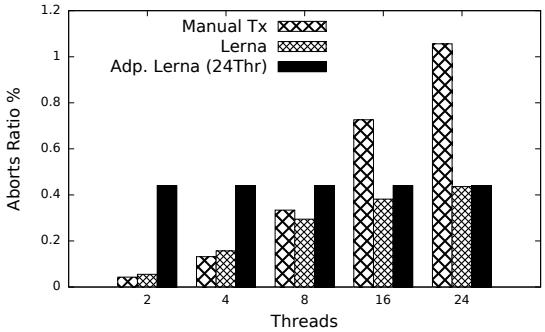
(e) Genome, Low Contention - Speedup



(f) Genome, Low Contention - Aborts



(g) Genome, High Contention - Speedup



(h) Genome, High Contention - Aborts

Figure 5.13: Kmeans and Genome Benchmarks

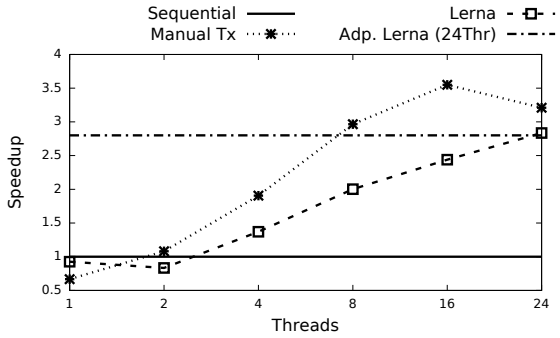
Benchmark	*Configurations*		Description
Kmeans	*Low*	-m60 -n60 -t0.00001 n65536 -d128 -c16	m: *max clusters*, n: *min clusters*
	High	-m20 -n20 -t0.00001 n65536 -d128 -c16	
Genome	*Low*	-g16384 -s64 -n86777216	n: *number of segments*
	High	-g16384 -s64 -n16777216	
Vacation	*Low*	-n30 -q90 -u100 -r1048576 -t4194304	n: *queries*, q: *relations queried ratio*
	High	-n50 -q60 -u100 -r1048576 -t4194304	
SSCA2	*Low*	-s20 -i0.1 -u0.1 -l3 -p3	s: *problem scale*, i: *inter cliques ratio*,
	High	-s19 -i0.5 -u0.5 -l3 -p3	u: *unidirectional ratio*
Labyrinth	*Low*	-x128 -y128 -z3 -n128	x, y, z: *maze dimensions*, n: *exits*
	High	-x32 -y32 -z3 -n96	
Intruder	*Low*	-a10 -l128 -n262144 -s1	l: *max number of packets per stream*
	High	-a10 -l12 -n262144 -s1	

Table 5.3: Input configurations for STAMP benchmarks.

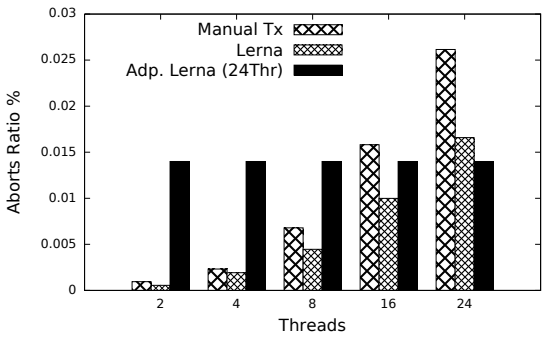
Vacation is a travel reservation system using an in-memory database. The workload consists of clients reservation. This application emulated an OLTP workload. Lerna improves the performance by 2.8× faster than the sequential system, and it is very close to the manual (See Figures 5.14a - 5.14d). Vacation transactions do not inhibit a lot of aborts as it accesses relatively large amount of data.

SCAA2 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 1.4×, while dropping the in-order commit allows up to 3.9× (See Figures 5.14e - 5.14h).

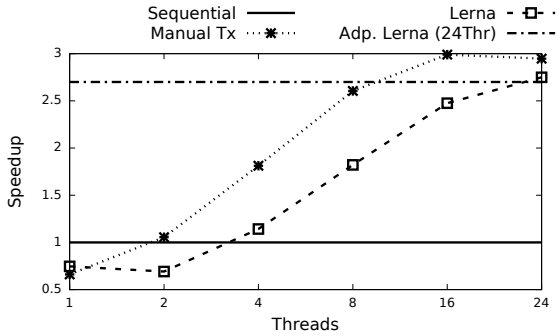
Lerna exhibits no speedup using *Labyrinth* and *Intruder* (See Figure 5.15) because they use an internal shared queue for storing the processed elements and they access it at the beginning of each iteration to dispatch them for the execution (i.e., a single contention point). While our jobs execute as a single transaction, the manual transactional version, creates multiple transactions per iteration. The first iteration, handle just the queue synchronization, while other iterations do the processing. Jobs partitioning will not help in this situation because the shared access occur at the beginning of the iteration. Assume splitting each job into two transactions, the ordering between jobs will prevent the first transaction at higher iterations from commit. However, we can see that this technique can help to parallelize two concurrent iterations at most; the first transaction at the higher index iteration can access the shared queue right after the corresponding lower index iteration commits.



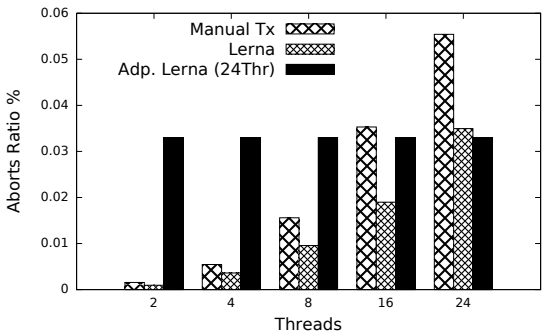
(a) Vacation, Low Contention - Speedup



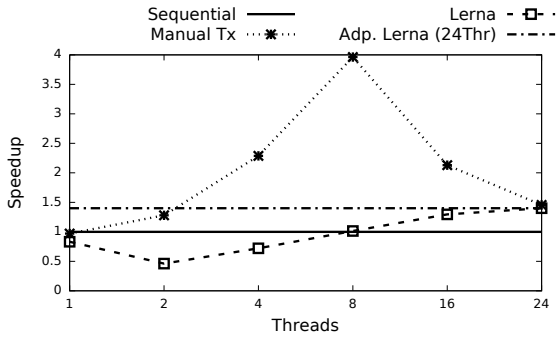
(b) Vacation, Low Contention - Aborts



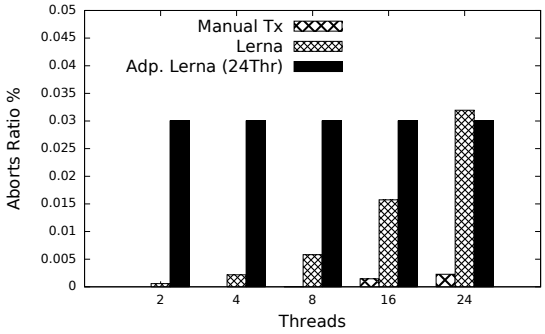
(c) Vacation, High Contention - Speedup



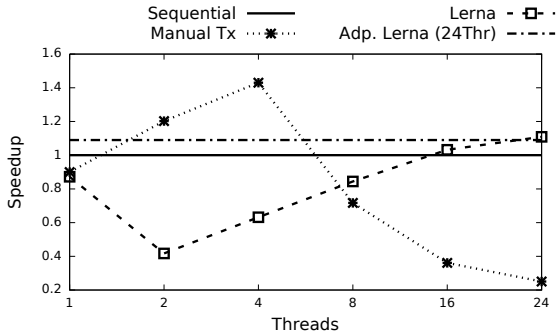
(d) Vacation, High Contention - Aborts



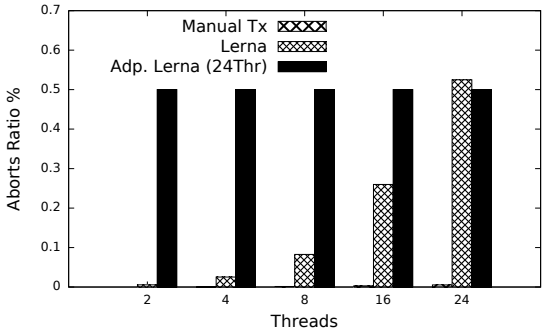
(e) SCAA2, Low Contention - Speedup



(f) SCAA2, Low Contention - Aborts

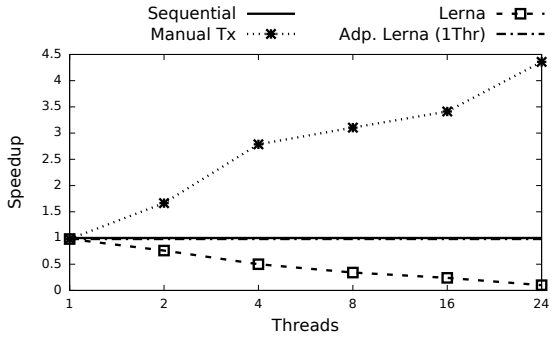


(g) SCAA2, High Contention - Speedup

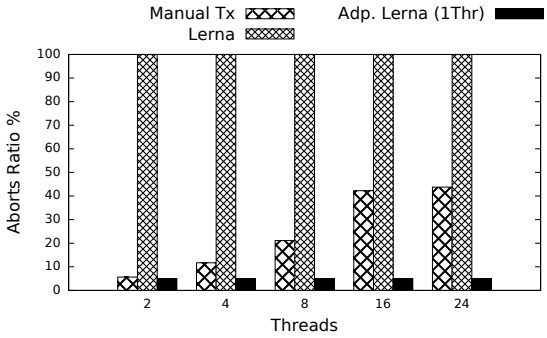


(h) SCAA2, High Contention - Aborts

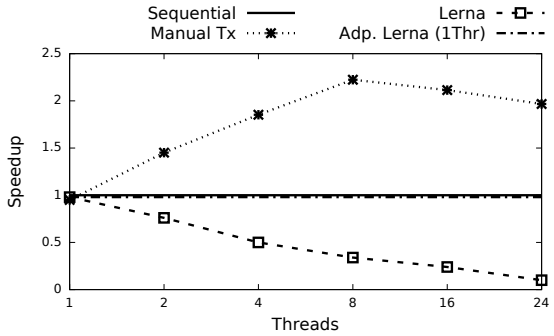
Figure 5.14: Vacation and SSCA2 Benchmarks



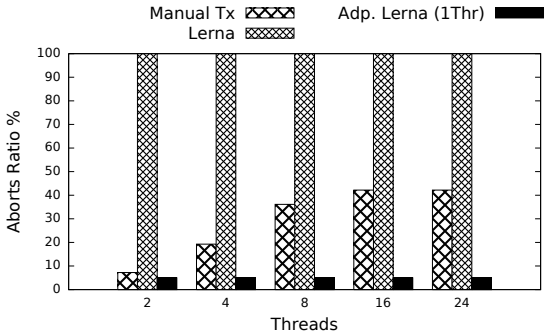
(a) Labyrinth, Low Contention - Speedup



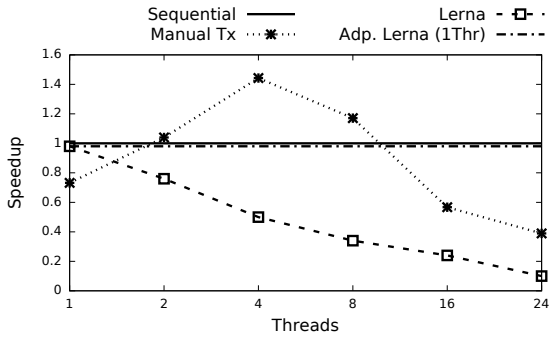
(b) Labyrinth, Low Contention - Aborts



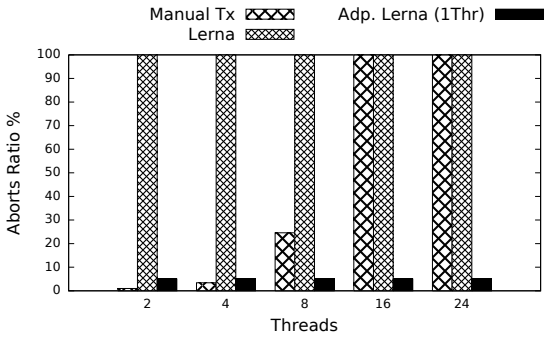
(c) Labyrinth, High Contention - Speedup



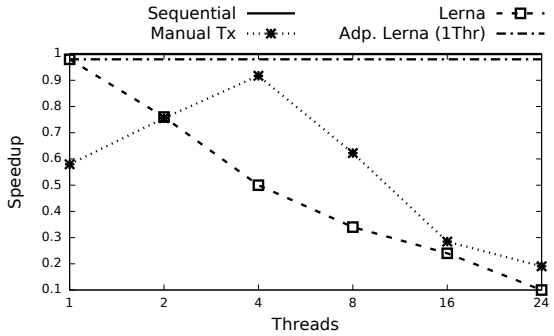
(d) Labyrinth, High Contention - Aborts



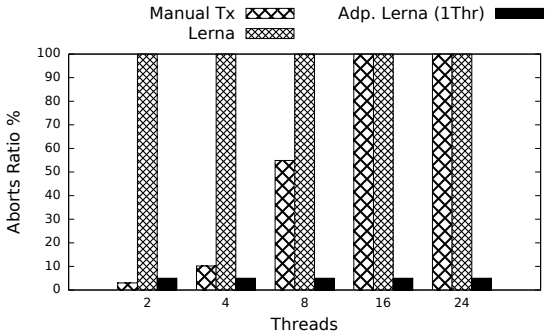
(e) Intruder, Low Contention - Speedup



(f) Intruder, Low Contention - Aborts



(g) Intruder, High Contention - Speedup



(h) Intruder, High Contention - Aborts

Figure 5.15: Labyrinth and Intruder Benchmarks

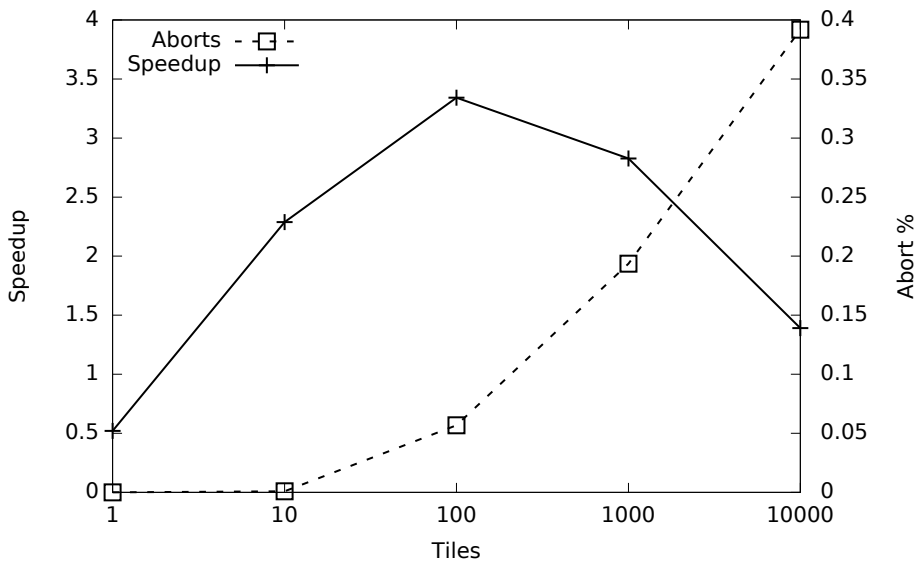
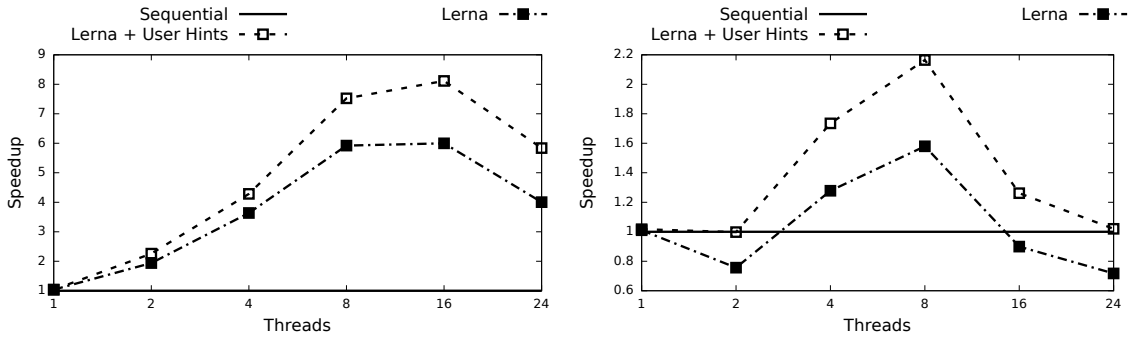


Figure 5.16: Effect of Tiling on abort and speedup using 8 workers and Genome.

As explained in Section 5.7.2, selecting the number of jobs per each transaction (jobs tiling) is crucial for performance. Figure 5.16 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 any avoidable overhead. An evidence of this is reported in Figure 5.17 where we show the speedup of Kmeans against different numbers of worker threads using two variants of the transformed



(a) Low Contention (b) High Contention

Figure 5.17: Kmeans performance with user intervention

Benchmark	*Configurations*	Description
Fluidanimate	-i in_500K	i: *input file with 500K particles data*
Swaptions	-ns 100000 -sm 1 -nt 1	ns: *number of swaptions*
Blackscholes	-i in_10M	i: *input file with 10 millions equations data*
Ferret	top=50 depth=5	top: *top K*, depth: *depth*

Table 5.4: Input configurations for PARSEC benchmarks.

code using Lerna: the first is the normal automatic transformation, and the second leverages user's hints about memory locations that can be accessed safely (i.e., non-transactionally). The figure shows that Lerna's transformed code outperforms even the manual transactional code.

5.8.3 The PARSEC Benchmark

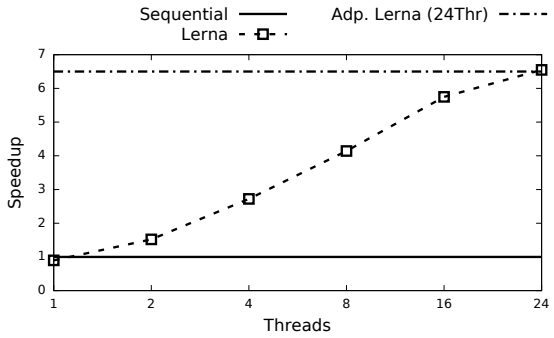
PARSEC [14] is a benchmark suite for shared memory chip-multiprocessors architectures. We evaluate Lerna performance using a subset of these benchmarks which cover different aspects of our implementation. Table 5.4 provides a summary of benchmarks and inputs used in the evaluation.

The Black-Scholes equation [73] is a differential equation that describes how, under a certain set of assumptions, the value of an option changes as the price of the underlying asset changes. This benchmark calculates Black-Scholes equation for input values and produces the results. The iterations are relatively short; which causes producing a lot of jobs in Lerna's transformed program. However, jobs can be tiled (See Section 5.7.2) where each group of iterations execute within a single job. Another approach is to add an unrolling pass earlier to our transformation. Figure 5.18c shows the speedup with different values for loop unrollings.

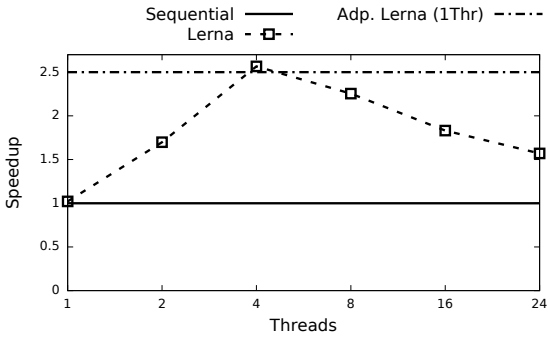
Swaptions benchmark contains routines to compute various security prices using Heath-Jarrow-Morton (HJM) [62] framework. Swaptions employs Monte Calro (MC) simulation to compute prices. Figure 5.18b shows Lerna speedup over the sequential code.

Fluidanimate [96] is a known 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 (See Figure 5.18d).

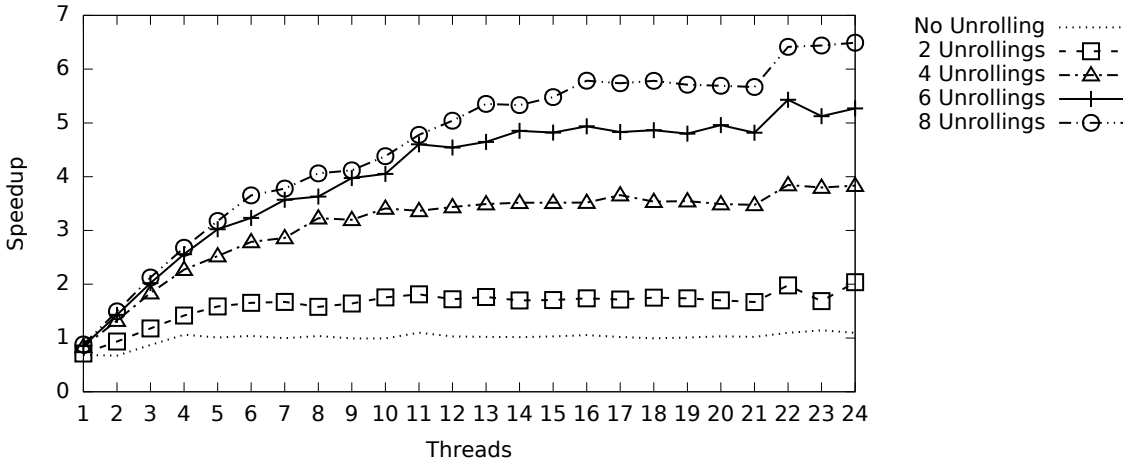
Ferret is a toolkit which is used for content-based similarity search. The benchmark workload is a set of queries for image similarity search. Similar to *Labyrinth* and *Intruder*, Ferret uses a shared queue to process its queries; which represents a single contention point and prevents any speedup with Lerna (See Figure 5.18e).



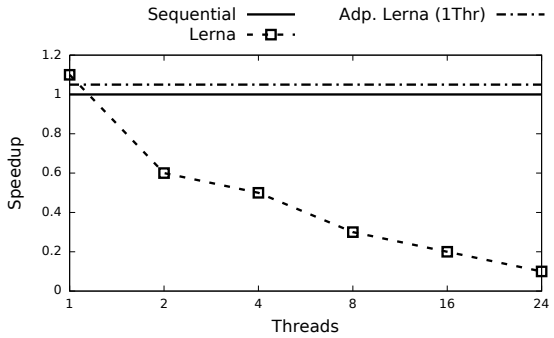
(a) Blackscholes



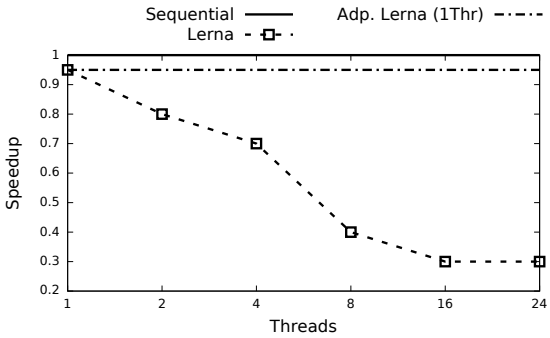
(b) Swaptions



(c) Effect of Loop Unrolling on speedup using Blackscholes



(d) Fluidanimate



(e) Ferret

Figure 5.18: PARSEC Benchmarks

5.9 Discussion

Lerna can be applied to all applications other than the used benchmarks. Here we discuss the lesson learnt from our evaluation in order to provide an intuition about which are the (negative) cases where Lerna's parallelization refactoring is less effective.

Lerna extracts parallelism when possible. 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 automatically break due to the lack of "semantic" knowledge. Examples of that include complex data structure operations. Examples of that include *Labyrinth*, *Intruder* and *Ferret* as explained before, or data-level conflicts as in *Fluidanimate*. We also looked into SPEC [63] applications, and we found that most of them use data structures iterators.

The primary factors of overhead are: ordering transactions, contention on accessing shared data (e.g., implied constraint by underlying bus-based architecture), and aborting conflicting transactions because of true data or control dependencies, or false conflicts.

In addition, Lerna becomes less effective when: there are loops with few iterations (e.g., Fluidanimate) 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 (e.g., Labyrinth); and workload is heavily unbalanced across iterations. Anyway, in all the above cases, at worst, the code produced by Lerna performs as the original.

Chapter 6

Conclusions & Post-Preliminary Work

The vast majority of the applications and algorithms are not designed to fulfill the new trend of multi-processor chips design, which creates a gap between the available commodity hardware and the running software. This gap is expected to continue for years with the burdens of developing and maintaining parallel programs. This dissertation proposal aims at extracting coarse-grained parallelization from the sequential code. We exploited TM as an optimistic concurrency technique for supporting safe memory access and introduced algorithmic modifications for preserving program chronological order. TM is known with its execution overhead, which could be comparable to locking overhead, but in comparison to sequential code it could outweigh any performance gain. We tackled this issue in several ways such as: employing static analysis, fast-path sequential execution, transactional pooling, transactional increments, and irrevocable transactions optimizations.

We presented HydraVM, a JVM that automatically refactors concurrency in Java programs at the bytecode-level. Our basic idea is to reconstruct the code in a way that exhibits data-level and execution-flow parallelism. STM was exploited as memory guards that preserve consistency and program order. Our experiments show that HydraVM achieves speedup between 2×-5× on a set of benchmark applications.

We presented Lerna, the first completely automated system that combines a software tool and a runtime library to extract parallelism from sequential applications without any programmer intervention. Lerna leverages software transactions to solve conflicts due to data sharing of the produced parallel code, thus preserving the original application semantics. Using Lerna is finally possible ignoring the application logic and exploiting the cheap hardware parallelism using a blind parallelization. Lerna showed promising results with multiple benchmarks with average 2.7× speedup over the original code.

6.1 Proposed Post-Prelim Work

In the previous chapters, we leveraged the key idea of employing TM as an optimistic concurrency strategy where code blocks run as transactions and they commit according to the program chronological order. Conflict is handled by aborting (and re-executing) the transaction which run code with the later chronological order.

As a post-preliminary works, we propose two optimizations for enhancing the TM performance and increasing processors utilization. Our first optimization aims to relax the transaction commit ordering and instead modifying the underlying TM algorithm to recover from inconsistent memory state. Secondly, we try to minimize the costly abort procedure by *checkpointing* the work done within a conflict transaction. That way, an aborted transaction does not need to retry all work done from the beginning.

6.1.1 Analysing Ordered Commit

Assuming T_i and T_j running code blocks i and j respectively, where $i < j$ in the program chronological order (age). Although that T_j may finish executing its code before T_i, it must wait for T_i to commit first, before T_j can start committing its changes. This strategy force thread executing higher age transactions to either: i) *stall*, or ii) *freeze* completed transaction, and moving to higher age transactions which increases transaction lifetime (hence, the conflict probability). In both cases, the overall utilization is negatively affected either by wasting processing cycles in the stall or through re-excuting aborted transactions (as a result of increasing conflict probability).

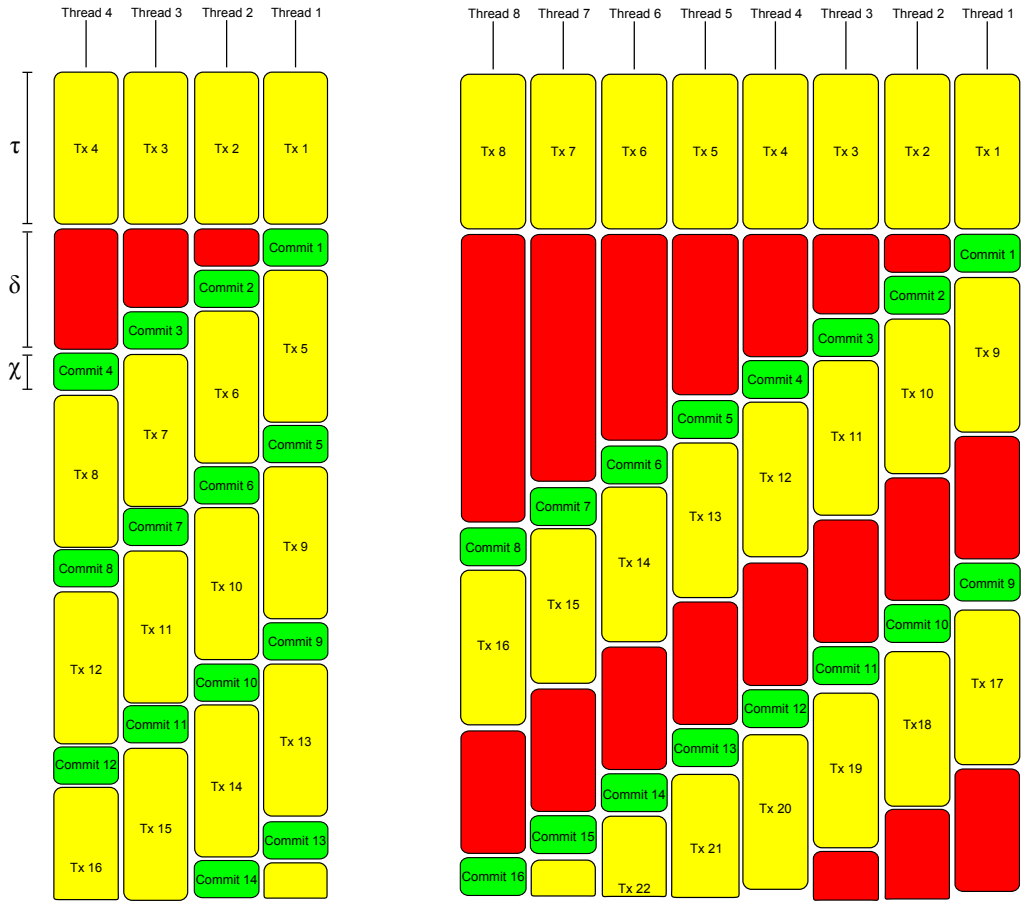
Let τ_x be the time required for T_x to execute code block x, where x is the transaction *age* according to program chronological order. T_x requires time of χ_x to expose (commit) its changes to outer world (including the time for retry executing transaction, if commit fails), and to notify its successor transaction with the commit completion.

Assuming T_x is the last committed transaction, and transactions T_{x+1} to T_{x+n} are running in parallel on n processor. Given two transactions T_{x+i} and T_{x+j}, where $i < j$, if T_{x+j} completed execution before T_{x+i} finish commit, so T_{x+j} will have to wait for $\delta_{i,j}$ time.

The following formula shows the total wait time for N transactions distributed equally over the n processor.

$$\Sigma_\delta = (N - n) * ((n - 1) * \chi - \tau) + \chi * n * (n - 1)/2$$

The first n transactions start at the same time but commit in order, causing the delay represented by the second term in the equation. Transactions $n+1$ till N have to wait all its predecessors to completes commit, however, there could be an overlap between transaction



(a) 4 Threads (b) 8 Threads

Figure 6.1: The Execution of Ordered Transactions over 4 and 8 threads

active and the commit of its predecessors. This (possible) delay is formed in the first term of the formula.

Figure 6.1 shows the execution of ordered transactions running over 4 and 8 threads, where $\chi = 0.25\ \tau$. For the sake of brevity, in this example we assume that: 1) all transaction executing code with same size, 2) there is no conflicting transaction. With 4 threads, the delay occurs only at the beginning (the second term of the above equation), while 8 threads suffer from continuous delay along the execution. Using the above formula we can see that up to 5 threads, there is no continuous delay during the execution.

The total delay exhibits quadratic growth with increasing the number of threads, and is affected by the ratio of commit time, χ, to execution time, τ. Figure 6.2 shows the total delay with increasing number of threads, and for different ratios of χ and τ. Notice that χ can be longer than τ when the transaction is aborted during the commit (e.g., as a result of read-set validation), so it will be re-executed.

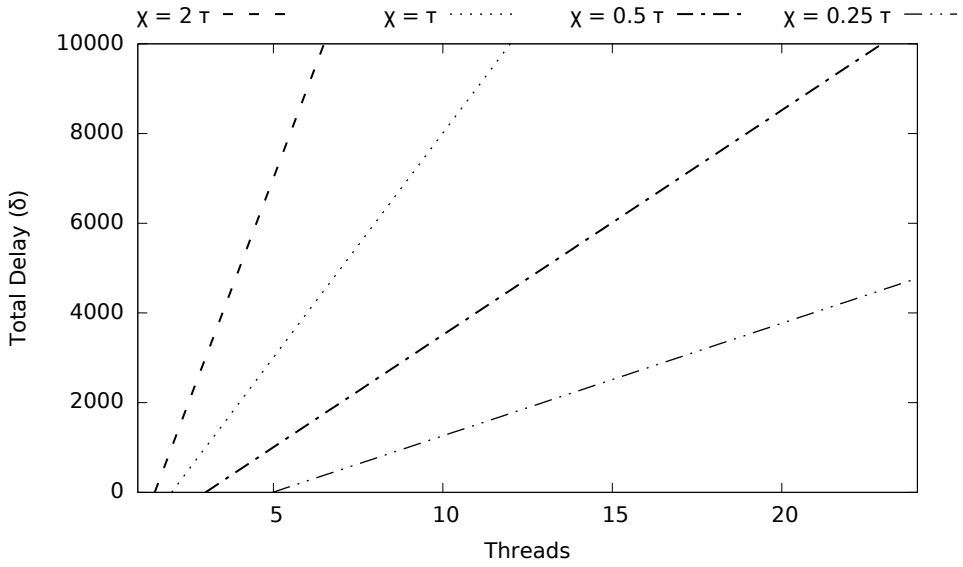


Figure 6.2: Total delay time (δ) of executing 1000 transactions with increasing number of threads. The unit of y axis in number of τ

From the previous discussion, we see that ordering transactions commit greatly affects the overall utilization and could nullify any targeted gain from adding more threads. In the following section, we propose some changes to transactional memory algorithms to avoid ordering delay.

Towards Out-Of-Order Transactional Execution

In the context of speculative code execution using memory transactions, transactions are aborted if: 1) transaction is unreachable (e.g., change in the program control flow), or 2) transaction conflict with earlier transactions. If we allowed transactions to commit independently from its chronological order, then we will need to handle the following situations:

- For unreachable transactions, there must be a way to rollback its changes
- Aborted transactions must be able to do cascade aborting to any other transaction that had read a variable in the aborted transaction write-set
- Lower age transactions must be able to conflict with (and abort) committed higher age transactions

To support these situations, we will need to keep some transaction meta-data even after it commit. This meta-data should help us to detect conflict or abort the committed transactions, however, once we are sure that the transaction become reachable (i.e., all its predecessor transactions get committed successfully) then we can release this meta-data.

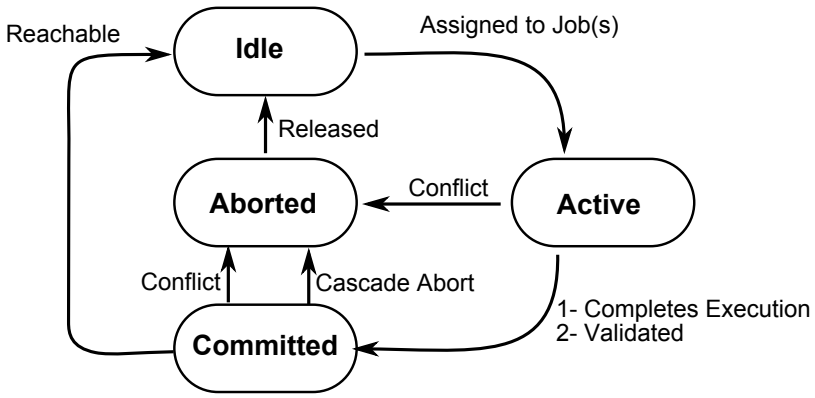


Figure 6.3: Transactional States

Figure 6.3 shows the proposed transactional states: *idle*, *active*, *committed*, and *aborted*, with its transition conditions. Transaction descriptor is *idle* till being assigned to a code block to execute. The transaction is *active* as long as it is executing the code block. Upon conflict with another concurrent transaction with lower *age*, the transaction is *aborted*. If the transaction completed all code execution, it does the validation step and commit its changes moving to the *committed* state. Transaction keeps its read-set and write-set in the committed state. Unlike the classical transactional state, *committed* transaction may move to the *aborted* state due to conflict with lower age transactions, or if it had read address from another *aborted* transaction (cascade abort).

To summarize, the required changes to TM algorithm to support out-of-order commit of parallel code is:

- Keep read and write sets, including any acquired locks, for the committed transaction, and release them when all predecessor transactions get committed.
- Support cascade abort between multiple transactions in the committed states that shares read and write sets addresses. Taking into account that cycles can exist when an active transaction with lower age read value from committed transaction, then tries to update a value written by the committed transaction, In this case, both transactions will be aborted.

6.1.2 Transaction Checkpointing

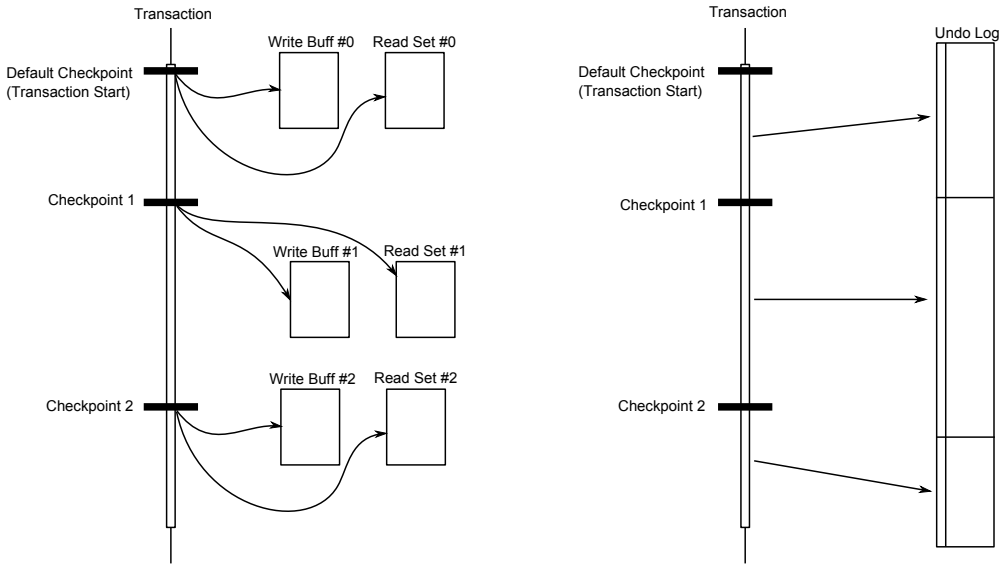
Transactions are defined as a unit of work; *all or none*. This assumption mandates executing the whole transaction body upon conflicts, even if the transaction has executed correctly for a subset of its lifetime. In the context of parallelization, restarting transactions could prevent any possible speedup; especially with preserving order and executing balanced transactions with equal processing time (e.g., similar loop iterations). In order to tackle this problem, we propose *transaction checkpointing* as a technique that creates multiple checkpoints at some execution points. Using checkpoints, a transaction acts as if it saved the system state at each checkpoint. A conflicting transaction can select which is the best checkpoint wherein the execution was valid and retry execution from it.

Inserting checkpoints can be done using different techniques:

- *Static Checkpointing*. Insert new checkpoint at equal portions of the transaction. This is done statically at the code.
- *Dependency Checkpointing*. Alias analysis and memory dependency analysis provides a best-effort guess for memory addresses aliasing. A common situation is the *may alias* result, which indicates a possible conflict. Placing a checkpoint before *may alias* accesses could improve the performance; a transaction will continue execution if *no alias*, and will retry execution right before the invalid access at *exact alias* situations.
- *Periodic Checkpointing*. During runtime, we checkpoint the work done at certain times (e.g., after doing a considerable amount of work).
- *Last Conflicting Address*. In our model, transactions executing symmetric code (i.e., iteration). Conflicting transactions usually continue conflicting at successive retrials. A conflicting address represents a guess for possible transactions collisions, and could be used as a hint for placing a checkpoint (i.e., before accessing this address).

The performance gain from using any of these proposed techniques is application and workload dependent. Additionally, checkpointing introduces an overhead for handling the checkpoints; unless the work done by the transaction is large enough to outweigh this overhead, it is not recommended. Checkpointing is useful when a transaction finishes its processing by updating a shared data structure (e.g., Fluidanimate see Section 5.8.3); with checkpointing, the aborted transaction can jump back till the checkpoint before the shared access and retry execution.

In order to support checkpointing a transaction must be *partially* aborted. As mentioned before, TM algorithms use either eager or lazy versioning through the usage of a *write-buffer* or an *undo log*. For eager TM, the undo log need to store a meta-data about the checkpoint (i.e., when it occurs). Whenever transaction wants to partially rollback, the undo-log is used to undo changes until the last recorded checkpoint in the log (See Figure 6.4b). With



(a) Write-buffer (b) Undo Log

Figure 6.4: Checkpointing implementation with write-buffer and undo-logs

lazy TM implementations, the write-buffer must be split according to the checkpoints. Each checkpoint is associated with a separate write-buffer stores its changes (See Figure 6.4a). Upon conflict, transaction discard write-buffers for checkpoints exist after the conflicting location. Drawbacks of this approach are: read operations needs to check multiple write-sets, and write-buffers should not be overlapped. Another alternative approach is to consider check checkpoint as a nested transaction, and employs closed-nesting techniques for handling partial rollback. However, supporting closed-nesting introduces a considerable overhead to the TM performance and complicates the system design.

Bibliography

[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] The LLVM Compiler Infrastructure. `http://llvm.org`.

[4] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the 2006 Conference on Programming language design and implementation*, pages 26–37, Jun 2006.

[5] M. Agarwal, K. Malik, K. M. Woley, S. S. Stone, and M. I. Frank. Exploiting postdominance for speculative parallelization. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 295–305. IEEE, 2007.

[6] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Washington, DC, USA, 2005. IEEE Computer Society.

[7] T. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 1(1):6 –16, Jan. 1990.

[8] M. Arnold, S. Fink, D. Grove, M. Hind, and P. 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.

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

[10] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *ACM SIGPLAN Notices*, volume 35, pages 1–12. ACM, 2000.

[11] J. Barreto, A. Dragojevic, P. Ferreira, R. Filipe, and R. 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.

[12] L. Baugh, N. Neelakantam, and C. Zilles. Using hardware memory protection to build a high-performance, strongly atomic hybrid transactional memory. In *In Proceedings of the 35th 8 International Symposium on Computer Architecture*, 2008.

[13] A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreading. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '02, pages 99–108, New York, NY, USA, 2002. ACM.

[14] C. Bienia, S. Kumar, J. P. Singh, and K. 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.

[15] G. Bilardi and K. Pingali. Algorithms for computing the static single assignment form. *J. ACM*, 50(3):375–425, 2003.

[16] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13:422–426, July 1970.

[17] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, and T. Lawrence. Parallel programming with polaris. *Computer*, 29(12):78–82, 1996.

[18] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. *SIGARCH Comput. Archit. News*, 35(2):24–34, 2007.

[19] P. Boulet, A. Darte, G.-A. Silber, and F. Vivien. Loop parallelization algorithms: from parallelism extraction to code generation. *Parallel Comput.*, 24:421–444, May 1998.

[20] B. Bradel and T. Abdelrahman. Automatic trace-based parallelization of java programs. In *Parallel Processing, 2007. ICPP 2007. International Conference on*, page 26, sept. 2007.

[21] B. J. Bradel and T. S. Abdelrahman. The use of hardware transactional memory for the trace-based parallelization of recursive java programs. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 101–110, New York, NY, USA, 2009. ACM.

[22] R. L. H. C. C. M. Bratin Saha, Ali-Reza Adl-Tabatabai and B. Hertzberg. McRT-STM: a high performance software transactional memorysystem for a multi-core runtime. In *PPOPP*, pages 187–197, 2006.

[23] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, PACT '01, pages 280–291, Washington, DC, USA, 2001. IEEE Computer Society.

[24] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le. Robust architectural support for transactional memory in the power architecture. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 225–236. ACM, 2013.

[25] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[26] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, Jun 2007.

[27] B. Carlstrom, J. Chung, H. Chafi, A. McDonald, C. Minh, L. Hammond, C. Kozyrakis, and K. Olukotun. Executing Java programs with transactional memory. *Science of Computer Programming*, 63(2):111–129, 2006.

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

[29] B. Chan and T. Abdelrahman. Run-time support for the automatic parallelization of Java programs. *The Journal of Supercomputing*, 28(1):91–117, 2004.

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

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

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

[33] P. Chen, M. Hung, Y. Hwang, R. Ju, and J. Lee. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. In *ACM SIGPLAN Notices*, volume 38, pages 25–36. ACM, 2003.

[34] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. *ACM SIGPLAN Notices*, 34(10):1–19, 1999.

[35] R. A. Chowdhury, P. Djeu, B. Cahoon, J. H. Burrill, and K. S. McKinley. The limits of alias analysis for scalar optimizations. In *Compiler Construction*, pages 24–38. Springer, 2004.

[36] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Proc. of 1986 Int'l Conf. on Parallel Processing*, 1986.

[37] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

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

[39] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 336–346, New York, NY, USA, 2006. ACM.

[40] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *ACM SIGPLAN Notices*, volume 29, pages 230–241. ACM, 1994.

[41] M. DeVuyst, D. M. Tullsen, and S. W. 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.

[42] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco, W. Mesard, M. Moir, K. Moore, and D. Nussbaum. Applications of the adaptive transactional memory test platform. In *Transact 2008 workshop*, 2008.

[43] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *In Proc. of the 20th Intl. Symp. on Distributed Computing*, 2006.

[44] Dice, D. and Shavit, N. What Really Makes Transactions Faster? In *Proc. of the 1st TRANSACT 2006 workshop*, 2006.

[45] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *ACM Sigplan Notices*, volume 44, pages 155–165. ACM, 2009.

[46] Z. Du, C. Lim, X. Li, C. Yang, Q. Zhao, and T. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. *ACM SIGPLAN Notices*, 39(6):71–81, 2004.

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

[48] R. Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006.

[49] K. A. Faigin, S. A. Weatherford, J. P. Hoeflinger, D. A. Padua, and P. M. Petersen. The polaris internal representation. *International Journal of Parallel Programming*, 22(5):553–586, 1994.

[50] P. Felber, C. Fetzer, and T. Riegel. 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 '08, pages 237–246, New York, NY, USA, 2008. ACM.

[51] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor. Kremlin: rethinking and rebooting gprof for the multicore age. In *ACM SIGPLAN Notices*, volume 46, pages 458–469. ACM, 2011.

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

[53] R. Guerraoui and M. Kapalka. Opacity: A Correctness Condition for Transactional Memory. Technical report, EPFL, 2007.

[54] M. Gupta, S. Mukhopadhyay, and N. Sinha. Automatic parallelization of recursive procedures. *International Journal of Parallel Programming*, 28(6):537–562, 2000.

[55] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, and E. Bu. Maximizing multiprocessor performance with the suif compiler. *Computer*, 29(12):84–89, 1996.

[56] L. Hammond, M. Willey, and K. Olukotun. *Data speculation support for a chip multiprocessor*, volume 32. ACM, 1998.

[57] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *in Proc. of ISCA*, page 102, 2004.

[58] T. Harris and K. Fraser. Language support for lightweight transactions. *ACM SIGPLAN Notices*, (38), 2003.

[59] T. Harris, J. Larus, and R. Rajwar. Transactional Memory, 2nd edition. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, 2010.

[60] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[61] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[62] D. Heath, R. Jarrow, and A. 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.

[63] J. L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[64] M. Herlihy. The art of multiprocessor programming. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 1–2, New York, NY, USA, 2006. ACM.

[65] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. volume 41, pages 253–262, New York, NY, USA, October 2006. ACM.

[66] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer. Software transactional memory for dynamic-sized data structures. In *In Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing*, pages 92–101. ACM Press, 2003.

[67] M. Herlihy, J. E. B. Moss, J. Eliot, and B. Moss. Transactional memory: Architectural support for lock-free data structures. In *in Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.

[68] S. S. Huang, A. Hormati, D. F. Bacon, and R. M. Rabbah. Liquid metal: Object-oriented programming across the hardware/software boundary. In J. 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.

[69] G. C. Hunt, M. M. Michael, S. Parthasarathy, and M. L. Scott. An efficient algorithm for concurrent priority queue heaps. *Inf. Process. Lett.*, 60:151–157, November 1996.

[70] W. M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *The Journal of Supercomputing*, 7:229–248, 1993. 10.1007/BF01205185.

[71] T. Johnson. Characterizing the performance of algorithms for lock-free objects. *Computers, IEEE Transactions on*, 44(10):1194 –1207, Oct. 1995.

[72] D. Kanter. Analysis of haswells transactional memory. *Real World Technologies (Febuary 15, 2012)*, 2012.

[73] N. Karjanto, B. Yermukanova, and L. Zhexembay. Black-scholes equation. *arXiv preprint arXiv:1504.03074*, 2015.

[74] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with Java STM. In *Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*, 2010.

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

[76] V. P. Krothapalli and P. Sadayappan. An approach to synchronization for parallel computing. In *Proceedings of the 2nd international conference on Supercomputing*, pages 573–581. ACM, 1988.

[77] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 209–220, New York, NY, USA, 2006. ACM.

[78] M. Lam and M. Rinard. Coarse-grain parallel programming in jade. In *ACM SIGPLAN Notices*, volume 26, pages 94–105. ACM, 1991.

[79] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan and Claypool, 2006.

[80] C. Lattner and V. 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.

[81] Z. Li, A. Jannesari, and F. Wolf. Discovery of potential parallelism in sequential programs. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 1004–1013. IEEE, 2013.

[82] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. 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.

[83] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. 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.

[84] M. G. Main. Detecting leftmost maximal periodicities. *Discrete Appl. Math.*, 25:145–153, September 1989.

[85] J. Mankin, D. Kaeli, and J. Ardini. Software transactional memory for multicore embedded systems. *SIGPLAN Not.*, 44(7):90–98, 2009.

[86] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. S. III, and M. L. Scott. Lowering the overhead of nonblocking software transactional memory. *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, June 2006.

[87] M. Mehrara, J. Hao, P.-C. Hsu, and S. 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.

[88] M. Mehrara, J. Hao, P.-C. Hsu, and S. 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.

[89] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[90] M. Moir. Practical implementations of non-blocking synchronization primitives. In *In Proc. of 16th PODC*, pages 219–228, 1997.

[91] K. E. Moore. Thread-level transactional memory. In *Wisconsin Industrial Affiliates Meeting*. Oct 2004. Wisconsin Industrial Affiliates Meeting.

[92] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *In Proc. 12th Annual International Symposium on High Performance Computer Architecture*, 2006.

[93] J. E. B. Moss. Open nested transactions: Semantics and support. In *In Workshop on Memory Performance Issues,*, 2005.

[94] J. E. B. Moss, N. D. Griffeth, and M. H. Graham. Abstraction in recovery management. In *ACM SIGMOD Record*, volume 15, pages 72–83. ACM, 1986.

[95] J. E. B. Moss and A. L. Hosking. Nested transactional memory: model and architecture sketches. *Sci. Comput. Program.*, 63:186–201, December 2006.

[96] M. Müller, D. Charypar, and M. 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.

[97] S. C. Müller, G. Alonso, A. Amara, and A. 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, Oct. 2014. USENIX Association.

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

[99] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. Daedalus: toward composable multimedia mp-soc design. In *Proceedings of the 45th annual Design Automation Conference*, pages 574–579. ACM, 2008.

[100] C. Nvidia. Compute unified device architecture programming guide. 2007.

[101] C. D. Polychronopoulos. Compiler optimizations for enhancing parallelism and their impact on architecture design. *Computers, IEEE Transactions on*, 37(8):991–1004, 1988.

[102] M. K. Prabhu and K. Olukotun. Using thread-level speculation to simplify manual parallelization. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '03, pages 1–12, New York, NY, USA, 2003. ACM.

[103] C. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *ACM Sigplan Notices*, volume 40, pages 269–279. ACM, 2005.

[104] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 65–76, New York, NY, USA, 2010. ACM.

[105] A. Raman, H. Kim, T. R. Mason, T. B. Jablin, and D. I. August. Speculative parallelization using software multi-threaded transactions. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 65–76. ACM, 2010.

[106] R. Ramaseshan and F. 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.

[107] L. Rauchwerger and D. Padua. The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization. *SIGPLAN Not.*, 30:218–232, June 1995.

[108] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *DISC 2006*, volume 4167 of *Lecture Notes in Computer Science*, pages 284–298. Springer, Sep 2006.

[109] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In S. Dolev, editor, *Distributed Computing*, Lecture Notes in Computer Science, pages 284–298. Springer Berlin / Heidelberg, 2006.

[110] T. Riegel, C. Fetzer, H. Sturzrehm, and P. Felber. From causal to z-linearizable transactional memory. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 340–341, New York, NY, USA, 2007. ACM.

[111] C. J. Rossbach, Y. Yu, J. Currey, J. Martin, and D. Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In M. Kaminsky and M. Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 49–68. ACM, 2013.

[112] E. Rotenberg and J. Smith. Control independence in trace processors. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 4–15. IEEE Computer Society, 1999.

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

[114] G. Rünger and M. Schwind. Parallelization strategies for mixed regular-irregular applications on multicore-systems. In *Advanced Parallel Processing Technologies*, pages 375–388. Springer, 2009.

[115] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06*, pages 187–197, Mar 2006.

[116] B. Saha, A.-R. Adl-Tabatabai, and Q. 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.

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

[118] W. N. Scherer III and M. L. Scott. Contention management in dynamic software transactional memory. In *PODC '04: Proceedings of Workshop on Concurrency and Synchronization in Java Programs.*, NL, Canada, 2004. ACM.

[119] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[120] M. Snir. *MPI–the Complete Reference: The MPI core*, volume 1. MIT press, 1998.

[121] G. S. Sohi, S. E. Breach, and T. Vijaykumar. Multiscalar processors. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 414–425. ACM, 1995.

[122] M. Spear, K. Kelsey, T. Bai, L. Dalessandro, M. Scott, C. Ding, and P. Wu. Fastpath speculative parallelization. *Languages and Compilers for Parallel Computing*, pages 338–352, 2010.

[123] J. Steffan and T. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *High-Performance Computer Architecture, 1998. Proceedings., 1998 Fourth International Symposium on*, pages 2–13. IEEE, 1998.

[124] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. *A scalable approach to thread-level speculation*, volume 28. ACM, 2000.

[125] K. Streit, C. Hammacher, A. Zeller, and S. Hack. Sambamba: runtime adaptive parallel execution. In *Proceedings of the 3rd International Workshop on Adaptive Self-Tuning Computing Systems*, page 7. ACM, 2013.

[126] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 356–369. IEEE, 2007.

[127] J. Tsai and P. Yew. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *Parallel Architectures and Compilation Techniques, 1996., Proceedings of the 1996 Conference on*, pages 35–46. IEEE, 1996.

[128] N. A. Vachharajani. *Intelligent speculation for pipelined multithreading*. PhD thesis, Princeton, NJ, USA, 2008. AAI3338698.

[129] H. Vandierendonck, S. Rul, and K. 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.

[130] C. von Praun, R. Bordawekar, and C. 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.

[131] P. Wu, A. Kejariwal, and C. Caşcaval. Compiler-driven dependence profiling to guide program parallelization. *Languages and Compilers for Parallel Computing*, pages 232–248, 2008.

[132] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[133] C. Zilles and G. Sohi. Master/slave speculative parallelization. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 35, pages 85–96, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[134] H. P. Zima, H.-J. Bast, and M. Gerndt. Superb: A tool for semi-automatic mimd/simd parallelization. *Parallel computing*, 6(1):1–18, 1988.