The HydraVM Project

[Technical Report]

Mohamed M. Saad
ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
msaad@vt.edu

Mohamed Mohamedin
ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
mohamedin@vt.edu

Binoy Ravindran
ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
binoy@vt.edu

ABSTRACT

We present a virtual machine prototype, called HydraVM, 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, and exploits software transactional memory to manage concurrent and out-of-order memory accesses. We describe HydraVM's architecture and implementation, and report on experimental studies using the JOlden benchmark, which shows up to 5× speed up.

Categories and Subject Descriptors

D.1.3 [**Software**]: Concurrent Programming; H.2.4 [**Systems**]: Transaction processing

General Terms

Design, Performance, Parallel Programming

Keywords

Software Transactional Memory (STM), Legacy Systems, Automatic Parallelization

1. INTRODUCTION

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 significant when legacy codebases run into several million LOC and are not significantly concurrent (often intentionally designed to be sequential to reduce development costs, while exploiting Moore's law of single-core chips). Manual exposition of concurrency is largely non-scalable, due to the significant difficulty in exposing concurrency and ensuring correctness of the converted code. Additionally, in some instances, sources are not available due to proprietary reasons, intellectual property issues (of integrated third-party software), and organizational boundaries. This motivates the need for *automated concurrency refactoring* techniques and tools.

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) [8, 23, 37, 19].

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.

Parallelization using thread-level speculation (TLS) hardware has been extensively studied, most of which largely focus on loops [36, 42, 22, 30, 41, 15, 16, 34, 43]. Automatic and semi-automatic parallelization without TLS hardware have also been explored [28, 37, 19, 17, 14].

Transactional memory (TM) has recently emerged as a powerful concurrency control abstraction [29]. With TM, code that read/write shared memory objects is organized as *transactions*, which 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 lock-based synchronization [38] and is composable. Multiprocessor TM has been proposed in hardware (HTM), in software (STM), and in hardware/software combination.

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 [10, 11, 13, 20], which use HTM to handle dependencies. [35] parallelizes loops with dependencies using thread pipelines, wherein multiple parallel thread pipelines run concurrently. [32] parallelizes loops by running them as

```
1  for(i=0; i<k; i++){
2     ...                                SUPERBLOCK As
3  }
4  if(i>50){
5     for(j=0; j<100; j++){
6        ...                             SUPERBLOCK Bs
7     }
8  }
```

Figure 1: Superblock example.

transactions, with STM preserving the program order. [40] parallelizes loops by running a non-speculative "lead" thread, while other threads run other iterations speculatively, with STM managing dependencies.

In this paper, we exploit STM for automated concurrency refactoring. Our basic idea is to optimistically split code (at the bytecode level) into parallel semi-independent sections, called *superblocks* [25]. For each superblock, we create a synthetic method that contains the code for the superblock and receives variables accessed by the superblock as parameters, and returns the exit point of the superblock. This synthetic method is executed in a separate thread, and is run as a memory transaction, while relying on STM to detect and resolve memory conflicts (between the superblocks).

Thus, each transaction has its own memory that it accesses or modifies. When the transaction is invoked, a copy of all variables is made and is sent to the method. Upon successful completion of the transaction, this copy is then merged back with the master memory version. In short, our memory model is lazy-commit with write-buffer implementation. To distinguish between multiple copies of an object, an identifier is added to the header of an object, which is unique in all copies of the object. We define a successful execution of an invoked superblock as when 1) it does not cause a memory conflict with another superblock with an older chronological order, and 2) it is reachable in a future execution of the program. The approach thus guarantees safe access to shared memory.

We build these techniques into a virtual machine (VM) called, HydraVM, by extending the Jikes RVM [3] and modifying its baseline compiler.

Figure 1 shows a simple example of executing superblocks. Assume that the two loops were detected as candidate superblocks. HydraVM will package every 10 iterations of the first loop as superblocks A_1, A_2, ..., A_n, and similarly for the second loop as superblocks B_1, ..., B_m. These superblocks will then be executed in parallel on different cores. However, some of the superblocks may conflict (e.g., due to variable dependencies between iterations), or may not be reachable (e.g., due to control flow statements). Thus, we abort and retry conflicting blocks, and abort non-reachable ones.

This approach is different than loop parallelization [9], because a superblock can span loops and method calls (see section 3.4). The main concern in constructing a superblock is in determining the portion of memory that it may access, and the number of instructions that it may contain.

As mentioned before, we need an STM implementation to handle potential memory conflicts. We thus develop *ByteSTM* (Section 3.5), which is an STM implementation at the virtual machine-level, which yields the following benefits: 1) 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; 2) Higher performance due to implementing all TM building blocks (e.g., versioning, conflict detection, contention management) at bytecode-level; and 3) Easy integration with other modules of HydraVM.

In contrast to other STM implementations, in ByteSTM, at commit time, a transaction scans the thread stack and registers and collects the addresses of the accessed objects. Objects identifiers 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.

To preserve the program order, each transaction must wait until its preceding code in the original program has been executed to commit. Toward this, 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 [12]. Our results reveal speedup of up to 5×.

Our work is different from past STM-based parallelization works in that we consider entire programs (not just loops such as [32, 40]), and automatically identify parallel sections (i.e., superblocks) by compile and run-time program analysis techniques, which are then executed as transactions. Additionally, our work targets arbitrary programs (not just recursive such as [11]), is entirely software-based (unlike [11]), and do not require program source code.

HydraVM is publicly available at `www.hydravm.org`.

The rest of the paper is organized as follows. In Section 2, we describe HydraVM's design and underlying mechanisms. We report on our experimental studies in Section 4. In Section 5, we overview past and related efforts and contrast them with HydraVM. We conclude in Section 6.

2. OVERVIEW

Adaptive Optimization System (AOS) [3] is a general virtual machine architecture that allows online feedback-directed optimizations. In HydraVM, we extend the AOS architecture to enable parallelization of input programs, and dynamically refine parallelized sections based on execution. Figure 2 shows the architecture of HydraVM which contains six components:

- Profiler: performs static analysis and adds additional instructions to monitor data access and execution flow

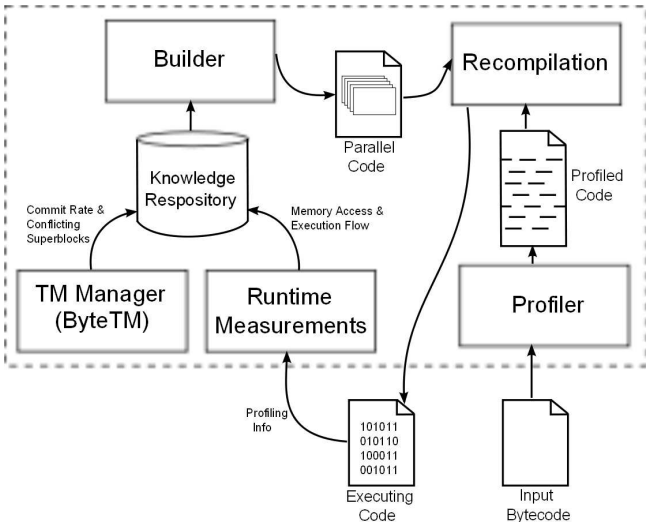

Figure 2: HydraVM Architecture

at run-time.

- Inspector: monitors program execution at run-time and produces profiling data.
- Recompilation: recompiles bytecode into machine code and reloads classes definitions at run-time.
- Knowledge Repository: a store for profiling data.
- Builder: uses profiling data to reconstruct the program as multi-threaded code, and tunes execution according to data access conflicts.
- TM Manager: does transactional concurrency control to guarantee safe memory and preserves program execution order.

HydraVM works in three phases. The first three phases focus 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 superblocks, which can be executed in parallel. New version of the code is generated and is compiled by the *Recompilation* component. The *TM Manager* manages memory access of the execution of the parallel version, and organizes transaction 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 superblocks 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.

```
1 for (Integer i = 0; i < DIMx; i++) {
2   for (Integer j = 0; j < DIMx; j++) {
3     for (Integer k = 0; k < DIMy; k++) {
4       X[i][j] += A[i][k] * B[k][j];
5     }
6   }
7 }
```

Figure 3: Matrix Multiplication Example

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

In the next subsections, we describe each of HydraVM's components.

2.1 Bytecode Profiling

First, HydraVM accepts program bytecode and converts it to architecture-specific machine code. 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. Thus, any program can be represented by a graph in which nodes represent basic blocks and edges represent the program control flow – i.e., an execution graph (see Figure 4). 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. To collect this information, we modify Jikes RVM's baseline compiler to insert additional instructions (in the program bytecode) at the edges of the basic blocks (e.g., branching, conditional, return statements) that detect whenever a basic block is reached. Additionally, we insert instructions into the bytecode to 1) statically detect the set of variables accessed by the basic blocks, and 2) mark basic blocks with input/output operations, as they need special handling in program reconstruction. This code modification doesn't affect the behavior of the original program. We call this version of the modified program, *profiled bytecode*.

2.2 Superblock 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. The benefits of execution graph are multifold: 1) Hot-spot portions of the code can be identified by examining the hot paths of the graph, 2) static data dependencies between blocks can be determined, and 3) parallel execution patterns of the program can be identified.

To determine superblocks, we use a string factorization technique: we represent each basic block by a character that acts like an unique identifier for that block. Now, an execution

of a program can be represented as a string. For example, Figure 3 shows a matrix multiplication code snippet. An execution of this code for a 2x2 matrix can be represented as the string $abjbhcfefghcfefghijbhcfefghcfefghijk$. We factorize this string into its basic components or "factors", using a variant of Main's string factorization algorithm [31], which is described in Algorithm 1.

Algorithm 1 Factorize(String x)

```
1: Compute the s-factorization x= u_1 . . . u_k of the input
   string x.
2: for all h in 2 .. k do
3:    L = 2*|u_h-1|+|u_h|
4:    t_h = substring of x with length L and which immedi-
      ately precedes u_h.
5: end for
6: S = new(Stack)
7: for all h in 2 .. k do
8:    S.push(all maximal periodicities in t_h u_h, which start
      in t_h and end in u_h).
9: end for
10: periodicities = new(List)
11: while !S.isEmpty() do
12:    m = S.pop()
13:    m_periodicities = Factorize(m)
14:    if m_periodicities.isEmpty() then
15:       periodicities.push(m) {minimal periodicity}
16:    else
17:       periodicities.push(m_periodicities)
18:    end if
19: end while
20: return periodicities
```

Periodicities are non empty strings of the form $p^m q$, where $m \geq 2$. In Algorithm 1, we start by finding the periodicities with the maximal length (lines 1-9). We then reapply the same method recursively to find the minimal ones (lines 10-20).

This 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 a set of nested calls, where each call execute a group of basic blocks, which becomes a *superblock*.

Thus, we divide the code, optimistically, into independent parts called superblocks that represent subsets of the execution graph. Each superblock doesn't overlap with other superblocks 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 superblock 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 superblock, leaving the other path outside the superblock. On the other hand, in unbiased branches, both paths may be included in the superblock. Therefore, a superblock has multiple exits, according to the program control flow during its execution. A superblock also has multiple entries, since a jump or a branch instruction may target one of the basic blocks that constructs it. The parallelizer module orchestrates the construction of superblocks and distributes them over parallel threads. However, this may potentially lead to out-of-order execution of the code, which we address through STM concurrency control (see Section 2.4). I/O instructions are excluded from superblocks, as changing their execution order affects the program semantics.

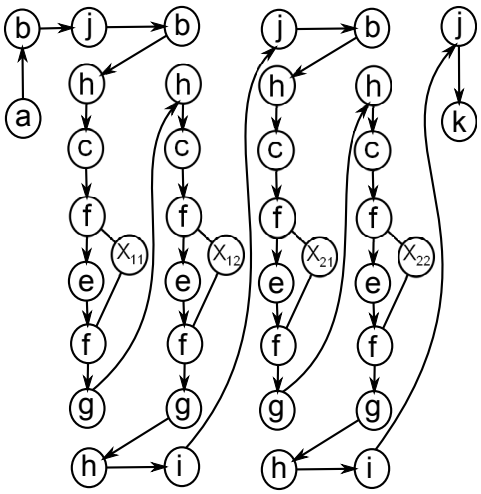

Figure 4: Matrix Multiplication Execution Graph

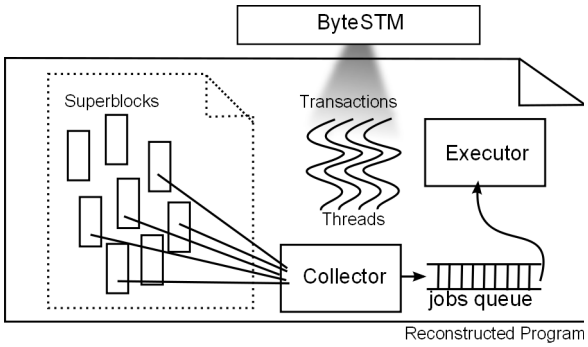

Figure 5: Program Reconstruction as a Producer-Consumer pattern

2.3 Code Reconstruction

Upon detection of candidate superblocks 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 tasks. 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 superblocks: the *Collector* has access to the generated superblocks and uses them as jobs, while the *Executor* executes the superblocks by assigning them to a pool of core threads.

Figure 5 shows the overall pattern of the generated program. Under this pattern, we utilize the available cores by executing the superblocks in parallel. However, doing so requires handling of several issues such as:

- Threads may finish in out of original execution order.

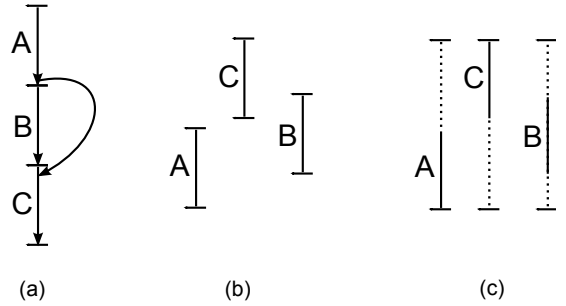

Figure 6: Parallel execution pitfalls: (a) normal sequential execution, (b) possible parallel execution scenario, and (c) transactional memory execution.

- The execution flow may change at run-time causing some of the assigned superblocks to be skipped from the correct execution.
- Due to the differences between 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 3.2).

To tackle these problems, we execute each thread 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.

2.4 TM Managed Parallelization

We now describe how to preserve data consistency and program order. To ensure data consistency, we use STM. Memory access violations are detected and resolved by STM through transactional conflict detection, abort, roll-back, and retry. Program order is maintained by deferring the commit of transactions that complete early till their valid execution time.

Consider the example in Figure 6, where three superblocks 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. Superblock A can have B or C as its successor, and that cannot be determined until run-time. According to the parallel execution shown in Figure 6(b), thread 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. 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 the parallel execution scenario in Figure 6(b), this will not be detected as a transactional conflict (T_C finishes before T_A). To handle this scenario, we extend the life time of transactions to the earliest transaction starting time. When a transaction must wait for its predecessor to commit, its life time is extended till the end of its predecessor. Figure 6(c) shows the execution from the transactional memory perspective.

```
1    y = 1
2    y += 2
3    x = y

1    y1 = 1
2    y2 = y1 + 2
3    x1 = y2
```

Figure 7: Static Single Assignment form Example

2.5 Reconstruction Tuning

Transactional memory preserves data consistency. However, 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 superblocks are merged together as a single superblock with the required 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 run-time.

3. IMPLEMENTATION

3.1 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, 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) [6]. The SSA form guarantees that each local variable has a single static point of definition, which significantly simplifies analysis. Figure 7 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 for a *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 during those paths. We then package each set of basic blocks in a superblock. Superblocks 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.

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, if a questionable dependency occurs,

the STM will detect and handle it. Otherwise, such blocks will run in parallel and greater speedup is achieved.

3.2 Misprofiling

We rely on our analysis on online profiling for detecting execution flow, which mainly depends on the input in the profiling phase. This input may not reflect some run-time aspects of the program flow (e.g., loops limits, biased branches). To illustrate this, we return to the matrix multiplication example in Figure 3. Based on the profiling using 2x2 matrices, we construct the execution graph shown in Figure 4. Now, assume that we have the following superblocks ab, $jbhi$, $hcfefg$ and jk, and 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 superblock jk, the Executor will find that the code exits after j and needs to execute bhi. Hence, it will request the Collector to schedule the job $jbhi$ in the incoming job set. Doing so allows us to extend the flow to cover more iterations. Note that the entry point must be send to the synthetic method that represents the superblock, as it should be able to start from any of its basic blocks (e.g., $jbhi$ will start from b not j, as j already executed before).

3.3 Handing Irrevocable Code

Input and output instructions must be handled as a special case in the reconstruction and parallel execution as they are irrevocable – i.e., they cannot be rolled back. Superblocks with I/O instructions are therefore marked for special handling. The Collector never schedules such marked superblocks unless they are reachable – i.e., they cannot be run in parallel with their preceding superblocks. However, they can be run in parallel with their successor superblocks. This implicitly ensures that at most one I/O superblock executes.

3.4 Method Inlining

Method inlining is the insertion of the complete body of a method at 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 has significant impact on performance, as it 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., $abababab\ldots$). Similar to method-inlining, we merge multiple levels of recursion into a single superblock, 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.

3.5 ByteSTM

ByteSTM is an STM that operates at the bytecode level and is integrated into the VM. We modified the Jikes RVM to support TM by adding new instructions, $xBegin$ and $xCommit$. $xBegin$ is used to start a transaction, while $xCommit$ is used to end the transaction.

Each load and store inside a transaction is done transactionally – i.e., loads are recorded in a read signature and stores are sand-boxed. In more detail, each store is not written to the original variable, but instead, it is stored in a transaction-local storage, called the write set. Since we are working at the low level of the VM, the address of any variable can be accessed. Thus, we add the address to the write signature. The read/write signature is represented using a Bloom filter [7]. Using a signature to detect read/write or write/write conflicts between transactions is significantly less expensive than comparing the read set and write set of transactions, but it also increases false negatives.[1]

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 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 and the changes become visible to the outside world.

For a VM-level STM, greater optimizations are possible than that for non VM-level STMs (e.g., Deuce [27], DSTM2 [24]). At this low level, data types do not matter. All we care about is the size of a data type, which allows us to simply the data structures used to handle transactions. One through eight-byte data types are handled in the same way. In the same manner, 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 transactional 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. Working at the VM level allows us to disable the GC for the memory used for the internal data structures that support STM. In ByteSTM, we statically allocate memory for STM, handle it without interruption from the GC, and manually recycle it. 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.

When used to preserve the data consistency between su-

[1]With the correct signature size, the effect of false positives can be reduced – we do this.

perblocks, STM must be modified to support ordering of the superblocks. Each superblock has an order that represents its logical order in the sequential execution of the original program. This order must be maintained. Thus, in ByteSTM, when a conflict is detected between two superblocks, 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 (in time). ByteSTM then allows it to commit 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 and updates the expected order. Otherwise, it sleeps and waits for its turn. 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.

3.6 Parallelizing Nested Loops

Nested loops are generally difficult for a parallelization engine, as it is difficult to parallelize both inner and outer loops at run-time. In HydraVM, we handle nested loops as nested transactions using the closed-nesting model [33], which is summarized using the following rules:

- Inner transactions share the readset/writeset of their parent transactions.
- Inner transactions may conflict with each other and also with other, non-parent, higher-level transactions.
- Aborting a parent transaction aborts all its inner transactions.
- 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.

Consider our earlier matrix multiplication example. We have an outer transaction $jbhi$, which invokes a set of inner transactions $hcfefg$ after the execution of the basic block b.

4. 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 [12]: 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 a multicore machine with 8 cores, each of which is an 800 MHz AMD Opteron Processor, with 64 KB L1 data cache, 512 KB L2 data cache, and 5 MB L3 data cache, and running Ubuntu Linux.

Evaluation. Table 1 shows the result of the Profiler analysis on the benchmarks. The table shows the number of basic blocks, superblocks, 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.

Table 1: Benchmark breakdown

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

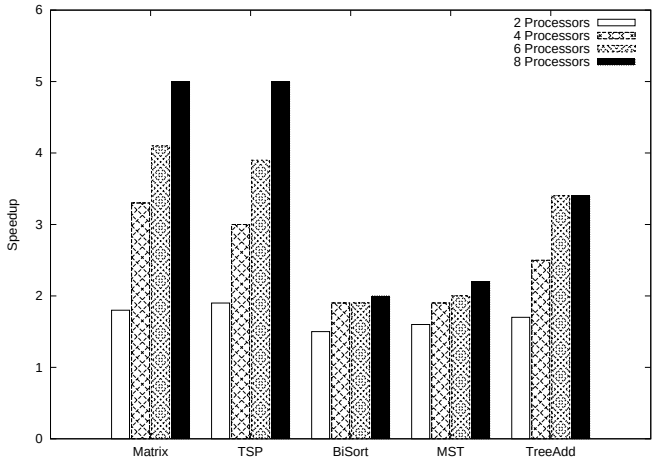

Figure 8: HydraVM Speedup

Using our techniques, we manage to split the sequential implementation of the benchmarks into parallel jobs that exploit data-level parallelism. Figure 8 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 into a superblock because of the iteration dependencies. In TSP, BiSort and TreeAdd, each multiple level of recursive call is inlined into a single superblock. 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 our engine.

5. PAST AND RELATED WORK

Past efforts on parallelizing sequential programs can be broadly classified into *speculative* and *non-speculative* techniques. Non-speculative techniques are usually compiler-based. Most works in this category exploit loop-level parallelism. Different techniques differ on the type of data dependency that they handle (e.g., static arrays, dynamically allocated arrays, pointers, etc.). For example, in [8, 23], the compiler finds loops with no cross-iteration dependencies and executes each iteration in a thread. A simple data analysis of arrays and scalars is used to determine such loops. Privatization is applied to arrays to eliminate some data dependency. In [37], recursive divide and conquer algorithms are split to execute in multiple threads. Data dependency in dynamically allocated arrays are also analyzed. In [19], data dependency between pointer aliases are analyzed. [18] presents a parallel compiler, editor, and debugger. The editor analyzes

a program and suggests possible transformations to increase parallelism to the programmer. [1] presents four optimization techniques to parallelize code. [39] presents a parallel translator system that converts sequential Fortran code into a parallel one.

Without speculation, the extracted threads must have no dependencies, which is generally difficult through static analysis alone at compile time. Some recent research in non-speculative parallelization delay data dependency analysis to run-time. The idea is to collect some information at compile time and use it to detect data dependencies at run-time (e.g., [14]).

Speculative techniques can be broadly classified based on factors including 1) the program constructs that are used to extract threads (e.g., loops, subroutines, traces, branch targets, etc.), 2) whether the technique is implemented in hardware, software, or depends on thread-level speculation (TLS) hardware, 3) whether the source code is required or not, and 4) whether the technique is online, offline, or both. Of course, this classification is not mutually exclusive.

Parallelization using TLS hardware has been extensively studied. Again, most such efforts focus on loop parallelization. In [36], DOALL loops without proven dependencies are parallelized using TLS hardware. Loops with dependencies are parallelized using thread pipelining and a new hardware design in [42]. [22] parallelizes loops based on a cost-driven data analysis and running them on TLS hardware. [30] parallelizes loops and subroutines on TLS hardware. [2] uses immediate post-dominators of conditional branches to parallelize on TLS hardware. [41] is based on loading data early using thread-level data speculation. Also, a new hardware addition is used. [15] uses a hardware tracer to dynamically extract speculative thread loops from a sequential program. The extracted threads run on TLS hardware.

[16, 34, 43, 22] analyze loop data dependencies at compile time and parallelize using TLS hardware. [41] parallelizes code using program profilers and TLS hardware. [15] presents TLS hardware to speculatively parallelize sequential programs. [30] uses heuristics to exploit parallelism from loops, subroutines, and continuations, and uses TLS hardware. Automatic loop parallelization using TLS hardware is also explored in [26].

Automatic and semi-automatic parallelization without TLS hardware have also been studied. [28] manually annotates concurrent and synchronized code blocks in C programs and then uses those annotations for run-time parallelization. [37] does compile-time analysis to exploit parallelism in array-based, divide-and-conquer programs. [19] analyzes symbolic access paths for interprocedural may-alias analysis, toward exploiting parallelism. [17] presents escape analysis for Java programs for determining object lifetimes toward enhancing concurrency. Automatic parallelization of pointer-based, recursive Java programs is presented in [14], where method dependencies are statically analyzed and used at run-time for parallel execution.

Trace-based automatic/semi-automatic parallelization is studied in [10, 11, 13, 20]. (Traces were invented in [4, 5] as part of HP's Dynamo optimizer, which optimizes native program binary at run-time using a trace cache.) [13] and [20] manually parallelize Java programs and use HTM to handle dependencies. A Trace Collection System is developed in [10, 11]. This work uses traces for parallelization of recursive programs with data-level parallelism, and HTM is used to handle dependencies.

[35] parallelizes loops with dependencies using thread pipelines, wherein multiple parallel thread pipelines run concurrently. [32] presents an all software technique to parallelize loops. The work uses a specially designed STM, called STMLite, to handle conflicts between threads and for ordering committing transactions. [40] proposes an all software technique to parallelize loops. A "lead" thread works non-speculatively and other threads run other iterations speculatively. STM is used to manage dependencies.

Our work is different from all these works in that, we propose STM-based parallelization, which is 1) entirely software-based, 2) is program source-independent, and 3) adaptive at run-time according to execution changes. Perhaps, the closest to our proposed work is [11]. Our work differs from [11] in the following ways. First, we propose STM for concurrency control, which doesn't need any hardware transactional support. Second, [11] is restricted to recursive programs, whereas we allow arbitrary programs. Third, [11] does not automatically infer transactions; rather, entire work performed in tasks (of traces) is packaged as transactions. In contrast, we identify superblocks by compile and run-time program analysis techniques, which are then executed as transactions.

Similar to our work, [32] uses STM for handling thread conflicts and uses a Transaction Commit Manager (TCM) to preserve program order. However, the work focuses on parallelizing loops; not an entire program. [21] introduces Behavior Oriented Parallelization (BOP), which parallelizes programs through profiling and uses the concept of Possibly Parallel Regions, which are code segments manually annotated for parallelization. In contrast, HydraVM is fully automated.

6. CONCLUSIONS

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.

7. REFERENCES

[1] V. Adve, G. Jin, J. Mellor-Crummey, and Q. Yi. High performance fortran compilation techniques for parallelizing scientific codes. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '98, pages 1–23, Washington, DC, USA, 1998. IEEE Computer Society.

[2] M. Agarwal, K. Malik, K. Woley, S. Stone, and M. Frank. Exploiting postdominance for speculative parallelization. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th*

International Symposium on, pages 295–305. IEEE, 2007.

[3] 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.

[4] V. Bala, E. Duesterwald, and S. Banerjia. Transparent dynamic optimization: The design and implementation of dynamo. Technical report, Technical Report HPL-1999-78, Hewlett-Packard Laboratories, 1999.

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

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

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

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

[18] K. Cooper, M. Hall, R. Hood, K. Kennedy, K. McKinley, J. Mellor-Crummey, L. Torczon, and S. Warren. The parascope parallel programming environment. *Proceedings of the IEEE*, 81(2):244 –263, feb 1993.

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

[20] 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.

[21] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. *SIGPLAN Not.*, 42:223–234, June 2007.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] T. Johnson, R. Eigenmann, and T. Vijaykumar. Speculative thread decomposition through empirical optimization. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 205–214. ACM, 2007.

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

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

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

[30] 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.

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

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.

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

[38] 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.

[39] V. Sarkar. The ptran parallel programming system. *Parallel Functional Programming Languages and Compilers*, pages 309–391, 1991.

[40] 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.

[41] 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.

[42] 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.

[43] 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.