HydraVM: Extracting Parallelism from Legacy Sequential Code Using STM

Mohamed M. Saad, Mohamed Mohamedin, and Binoy Ravindran
ECE Dept., Virginia Tech, Blacksburg, VA 24061, USA
{msaad,mohamedin,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. Our experimental studies show up to 5× speedup on the JOlden benchmark.

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 for such codebases. In some instances, sources are not available due to proprietary reasons, intellectual property issues (of integrated third-party software), and organizational boundaries. This motivates techniques and tools for *automated concurrency refactoring*.

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) [4, 13, 16, 27].

Speculative techniques can be broadly classified based on 1) what program constructs they use to extract threads (e.g., loops, subroutines), 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 [10, 11, 15, 20, 24, 26, 31–33]. Automatic and semi-automatic parallelization without TLS hardware have also been explored [9, 12, 13, 18, 27].

Transactional memory (TM) has recently emerged as a powerful concurrency control abstraction [19]. 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 [29] 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 [5, 6, 8, 14], which use HTM to handle dependencies. [25] parallelizes loops with dependencies using thread pipelines, wherein multiple parallel thread pipelines run concurrently. [22] parallelizes loops by running them as transactions, with STM preserving the program order. [30] 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* [17]. 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.

We build these techniques into a virtual machine (VM) called, HydraVM, by extending the Jikes RVM [1] and modifying its baseline compiler.

To handle potential memory conflicts, we develop ByteSTM, which is a VM-level STM implementation, 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 (Section 3.4). 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 [7]. 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 [22, 30]), 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 [6]), is entirely software-based (unlike [6]), and do not require program source code.

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

2. Overview

Adaptive Optimization System (AOS) [1] is a general VM 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 1 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 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 execution order.

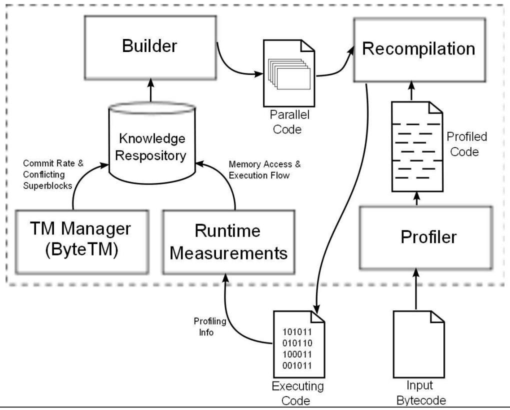

Figure 1. HydraVM Architecture

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.

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.

We now 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

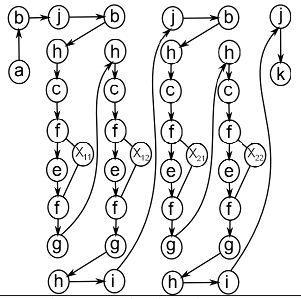

Figure 2. Matrix Multiplication Execution Graph

```
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];
```

Figure 3. Matrix Multiplication Example

which nodes represent basic blocks and edges represent the program control flow – i.e., an execution graph (see Figure 2). 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 graph's hot paths, 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: each basic block is represented by a character that acts like an unique ID 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 using a variant of Main's algorithm [21] that we have developed (our variant is described in [28]). 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 a set of nested calls, where each call execute a group of basic blocks, which becomes a *superblock*.

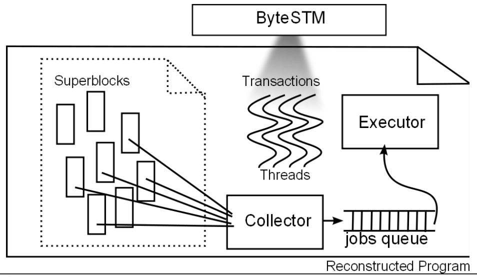

Figure 4. Program Reconstruction as a Producer-Consumer Pattern

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. I/O instructions are excluded from superblocks, as changing their execution order affects the program semantics, and they are irrevocable (i.e., at transaction aborts).

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 4 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.
- 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.

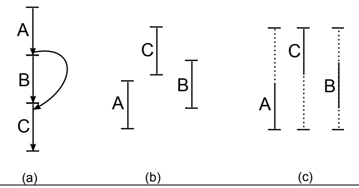

Figure 5. Parallel execution pitfalls: (a) normal sequential execution, (b) possible parallel execution scenario, and (c) TM execution.

2.4 TM Managed Parallelization

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 5, 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 in Figure 5(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. 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 5(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 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 5(c) shows the execution from the TM perspective.

2.5 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 superblocks are combined into a single superblock. 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 run-time.

3. Implementation

3.1 Detecting Real Memory Dependencies

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

Figure 6. Static Single Assignment form Example

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) [2]. The SSA form guarantees that each local variable has a single static point of definition, which significantly simplifies analysis. Figure 6 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. 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 runtime 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 2. 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 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 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.

3.4 ByteSTM

ByteSTM is an STM that operates at the bytecode level and is integrated into HydraVM. 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 sand-boxed; 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 write signature. The read/write signature is represented using a Bloom filter [3] and used to detect read/write or write/write conflicts.

Each superblock has an order that represents its logical order in the sequential execution of the original program. To preserve the data consistency between superblocks, STM must be modified to support this ordering. 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. 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 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.

3.5 Parallelizing Nested Loops

Nested loops are generally difficult for parallelization, as it is difficult to parallelize both inner and outer loops. In HydraVM, we handle nested loops as nested transactions using the closed-nesting model [23]: 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.

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 [7]: 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.

Table 1. 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
Superblocks	3	12	5	3	4
Jobs	1001	1365	1023	12241	8195
Max Nesting	2	5	2	1	3

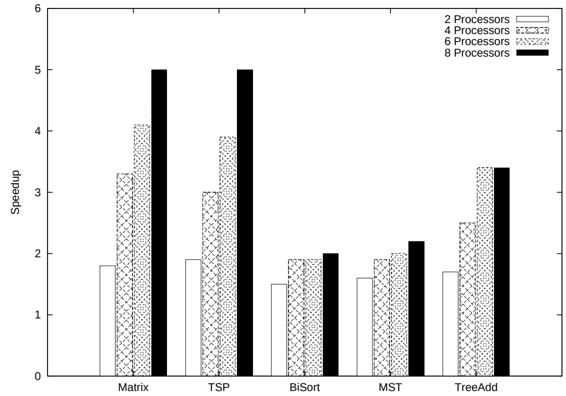

Figure 7. HydraVM Speedup

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 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.

Figure 7 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 interdependencies between iterations. However, updating the costs from the constructed MST and other nodes presents a good parallelization opportunity for HydraVM.

5. 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.

References

[1] 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. ISBN 1-58113-200-X. doi: http://doi.acm.org/10.1145/353171.353175. URL `http://doi.acm.org/10.1145/353171.353175`.

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

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13:422–426, July 1970. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/362686.362692. URL `http://doi.acm.org/10.1145/362686.362692`.

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

[5] 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. doi: 10.1109/ICPP.2007.21.

[6] 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. ISBN 978-1-60558-598-7. doi: http://doi.acm.org/10.1145/1596655.1596671. URL `http://doi.acm.org/10.1145/1596655.1596671`.

[7] 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. ISBN 0-7695-1363-8. URL `http://dl.acm.org/citation.cfm?id=645988.674177`.

[8] 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.

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

[10] 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.

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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. ISSN 0920-8542. URL `http://dx.doi.org/10.1007/BF01205185`. 10.1007/BF01205185.

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

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

[20] 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.

[21] M. G. Main. Detecting leftmost maximal periodicities. *Discrete Appl. Math.*, 25:145–153, September 1989. ISSN 0166-218X. doi: 10.1016/0166-218X(89)90051-6. URL `http://dl.acm.org/citation.cfm?id=82349.82359`.

[22] 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. ISBN 978-1-60558-392-1. doi: http://doi.acm.org/10.1145/1542476.1542495. URL `http://doi.acm.org/10.1145/1542476.1542495`.

[23] J. E. B. Moss and A. L. Hosking. Nested transactional memory: model and architecture sketches. *Sci. Comput. Program.*, 63:186–201, December 2006. ISSN 0167-6423. doi: 10.1016/j.scico.2006.05.010. URL `http://portal.acm.org/citation.cfm?id=1228561.1228567`.

[24] 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.

[25] 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. ISBN 978-1-60558-839-1. doi: http://doi.acm.org/10.1145/1736020.1736030. URL `http://doi.acm.org/10.1145/1736020.1736030`.

[26] 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. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/223428.207148. URL `http://doi.acm.org/10.1145/223428.207148`.

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

[28] M. M. Saad, M. Mohamedin, and B. Ravindran. HydraVM Project : Technical Report. Technical report, ECE Dept., Virginia Tech, January 2012. URL `http://www.hydravm.org/hydra/wiki/Publications`.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.