M-TAGs: Multi-Threaded Access Groups Analysis

Aditya Venkataraman
adityav@cs.wisc.edu
University of Wisconsin-Madison

Naveen Neelakandan
neelakandan@cs.wisc.edu
University of Wisconsin-Madison

Abstract

Emerging parallel-programming frameworks simplify multi-core programming, but require the programmer to provide hints about program behavior such as per-thread memory access regions, producer-consumer relationships etc. In this work, we present M-TAGs: a static analyzer that can identify per-thread memory read-/write-sets for multi-threaded C++ programs.

M-TAGs recognizes the usage of standard APIs in the new parallel-programming models for spawning new threads as well as identifying and synchronizing threads. It assumes statically determinable per-thread identifers and exploits this knowledge for important operations such as address computations, CFG pruning, etc. It evaluates the program in a Strided Interval Abstract Domain, which emulates popular stencil-/chunking-based work distribution paradigms. Lastly, M-TAGs keeps track of each thread's read-/write-set and enumerates write-write and write-read conflicts across all threads.

We demonstrate our approach's efficacy using a prototype implementation and highlight its advantages over routine thread-insensitive approaches using dedicated microbenchmarks as well as a benchmark from the PARSEC suite.

1. Introduction

Multi-core computing systems are ubiquitous today. From minute Internet-of-Things to high-end servers, processors equipped with multiple cores have exposed an enormous potential for improving performance, concurrency and scalabilty of applications. These systems typically support a shared memory abstraction and are commonly programmed using multi-threaded programs. However, multi-threaded programming is considered to be very challenging [8], due to data races, deadlocks, unpredictable thread interleavings, etc. Hence, such programs are highly vulnerable to hidden and non-reproducible bugs. Static analysis of multi-threaded programs can be highly effective at detecting potential data races, however it is difficult to realize for a broad range of multi-threaded programs [11]. A few common difficulties include:

- A single memory access instruction can access different locations in memory depending on the thread executing the instruction.
- Performing a flow-sensitive analysis of multi-threaded programs is computationally infeasible due to an unbounded number of possible thread interleavings.
- A flow-insensitive approach is computationally tractable, but rapidly loses precision

An emerging trend in parallel programming is the growing popularity of frameworks that assist parallel programming and execution, for e.g., Cilk [4], Cilk++ [1], Intel Thread Building Blocks [10], Java's Fork-Join Framework [7], Microsoft Task Parallel Library [9], StackThreads/MP [14] and Parakram [5]. These frameworks shield the programmer from the complexities of parallel execution such as synchronization. Instead, the programmer writes a simpler program and annotates it with hints about program behavior, such as per-thread/per-task memory access regions, producer-consumer relationships, etc. The hints are utilized by a runtime-engine to execute the program in a performant manner. The resulting performance is highly dependent on the quality of the hints provided; for e.g. highly conservative estimates on shared memory regions across threads could lead to excessive and unnecessary synchronization. Manual inputs for such performance-critical hints are likely to be conservative and potentially inaccurate in highly complex parallel programs. In this work we recognize the potential for static analysis to replace manual hints to parallel programming frameworks, particularly the estimation of per-thread memory access regions in multi-threaded programs. To overcome the challenges of traditional flow-insensitive techniques, we recognize that parallel programming frameworks allow only a few standard APIs for common operations such as thread spawn, thread kill, thread synchronization and thread identification. By recognizing these APIs and gleaning information from them, our analysis hopes to achieve thread-sensitive multi-threaded analysis. We hypothesize

that thread-sensitivity will simplify the analysis and improve the precision of the results.

Contributions The main contributions of this paper are:

- *We propose a thread-sensitive analysis of multi-threaded programs that exploits thread identification information to compute over-approximate memory access regions of each thread.*
- *We interpret the parallel program in a Strided Interval Abstract Domain that elegantly emulates stencil-/chunking-based work distribution paradigms.*
- *We implement a prototype of our analyzer on llvm.v.3.8 and demonstrate its efficacy for a range of microbenchmarks and a benchmark from the PARSEC parallel programming suite.*

Paper Organization We first give an overview of our approach and motivate it using simple examples (§2) . Subsequently, we describe our abstract domain and present a subset of its transformers (§3). We then present the analyzer's algorithm in three phases (§4) and describe its working using two case studies (§5). We present a few results for comparing M-TAGs against a non-thread sensitive technique (§6). Lastly, we summarize the related works (§7) and conclude (§8).

2. M-TAGs: Overview

Consider the example in Figure 1. Let us assume that two threads, T1 and T2, are concurrently manipulating two globally defined arrays `A` and `B`. The regions of the arrays accessed by T1 and T2 are shown in green and red respectively. As evident from the code, the arrays are being chunked into discrete blocks and assigned to different threads using the `tid` variable. However, a thread-insensitive static analyzer would report that the pointer `p` could point to any part of array `A` and `B`. If a static analyzer could be sensitive to the `tid` variable and the restricted values it could contain, then the analysis would be more precise.
Similarly consider the example in Figure 2. Let us assume that the pictured code snippet is part of a function being concurrently executed by multiple threads. However, based on `tid`, `Code A` will be executed by a single thread only. This presents an opportunity to perform single-threaded analysis on `Code A` and avoid polluting the analysis of `Code B`. As evident from these motivating examples, there is potential for a thread-sensitive analysis to simplify and improve the memory access analysis of multi-threaded programs. M-TAGs has the following key attributes:

2.1. Thread-sensitive Analysis

M-TAGs assumes a statically determinable thread-identifier for each thread. This could be realized by annotating the tid variable in the source code or exploiting existing APIs of parallel programming frameworks whose arguments necessarily include a tid variable. M-TAGs will assign unique values to this variable for different threads to, hopefully, simplify address calculations, CFG pruning for each thread, etc. M-TAGs also assumes that the work-functions for each thread are statically determinable. This is reasonable as thread spawns are done using standard APIs that require a work-function pointer as an argument.

2.2. Abstract Interpretation

M-TAGs computes the per-thread read-/write-sets by statically evaluating the program in a novel Strided Interval Abstract Domain. This domain will be defined in section (§3). This domain was desirable since it emulates strided array/matrix accesses.

2.3. Conflict Detection

Through abstract interpretation, M-TAGs computes an over-approximate set of read-/write-addresses for each thread. A memory location is said to be conflicted if it is accessed by the same or different instruction executed by different threads, and one of the accesses is a write. By post-processing the read-/write-sets of all threads, M-TAGs detects and enumerates conflicting memory locations.

3. Strided Interval Abstract Domain

The Strided Interval Abstract Domain represents a set of named regions of the address space. It is a 4-tuple, $dom : \rho(Name \times \mathbb{N} \times \mathbb{N} \times \mathbb{N})$ consisting of the following:
Name - A unique string to identify the region
Start - An inclusive lower bound
End - An exclusive upper bound
Stride - A step interval
If n denotes the number of threads in the program, then each virtual register in the LLVM IR is represented in the abstract domain as $reg : id \rightarrow dom^n$. Similarly, abstract memory is represented by $mem : addr \rightarrow dom$. As the virtual registers are private to each thread, we shall always perform strong updates on them. However, system memory is shared by all

```
int i, j, A[Max], B[Max];
int *p;
j = tid * CHUNK_SIZE;
for ( i=0; i<CHUNK_SIZE;
i++) {
  if (flag) p = &A[j];
  else p = &B[j];
  p[i] = p[i]*p[i];
  if (cond) flag = true;
}
```

Figure 1: Multi-threaded array-manipulation with no-conflicting accesses

```
if ( tid == REDUCER_TID )
{
  <Code A>
}
else {
  <Code B>
}
```

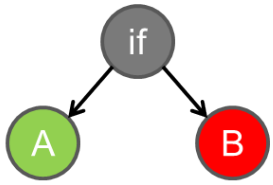

Figure 2: Differentiated worker threads based on thread id

threads. Hence, we shall perform weak updates in parallel regions of code and strong updates in single-threaded regions.

Below are two examples of elements in the concrete domain being represented in the abstract domain. The first example is that of a constant value while the second example denotes a set of values in the concrete domain.

$$x = 25 \xrightarrow{abs} (x,\ 25,\ 26,\ 1)$$

$$x = [2,\ 4,\ 6] \xrightarrow{abs} (x,\ 2,\ 7,\ 2)$$

Elements in the Strided Interval Abstract Domain are composable for a wide range of operations. As an example, consider the *Addition* and *LShift* as defined on the abstract domain as shown below. $X(l_1,\ u_1,\ s_1)$ and $Y(1_2,\ u_2,\ s_2)$ are elements of the abstract domain.

$$X\ ADD^{\#}\ Y = (l_1 + l_2,\ u_1 + u_2,\ GCD(s_1 + s_2))$$

$$X\ LSHIFT^{\#}\ Y = (l_1 << l_2,\ u_1 << u_2,\ GCD(l_1 + s_1) << u_2)$$

A complete list of the transformations can be found at Sen and Srikant [13].

4. M-TAGs: Algorithm

The M-TAGs analyzer takes as input a multi-threaded C++ program which uses standard APIs (for eg. pthreads) for thread operations such as spawning. At the end of the analysis, it outputs a set of over-approximate representations of the read-/write-regions of each thread in the application. It also computes any read-write and write-write conflicts across all threads.

The analyzer operates in three phases, as described below:

4.1. Pre-Processing Phase

The pre-processing phase takes a multi-threaded C++ program as input and prepares the program for convenient analysis. This phase has three passes:

- First we compile the C++ program into LLVM's SSA-based intermediate representation (IR) using LLVM assembler. LLVM IR is a strongly typed language with a simple type-system. It abstracts away underlying machine details such as the calling convention of the ISA. The SSA-based IR offers an infinite set of virtual registers, where each register will be assigned to exactly once. This simplifies our analysis as each arithmetic instruction can be associated with a single virtual register.
- Next, we apply LLVM's in-built mem2reg optimization on the IR. This optimization promotes references to thread local memory to virtual register references, pruning the IR. The remaining `load`, `store` instructions in the IR will pertain to global memory and pass-by-reference parameters.
- Lastly, we apply a self-written Inliner pass on the pruned IR. The Inliner pass will inline all functions calls made by the `main()` function into `main()` itself, except the `pthread_create` function call. Similarly, we will inline all calls made by each thread's start function into the start function itself. The various thread start functions can be detected from the arguments to `pthread_create`

call. As a result of inlining, M-TAGs need not consider inter-procedural data-flows during the analysis.

4.2. Analysis

First we describe the important data-structures used by the algorithm and then describe the algorithm itself.

4.2.1. Data Structures M-TAGs uses the following conceptual data-structures for its analysis:

Constraint Maps M-TAGs operates on the Control-Flow Graph (CFG) of the program. As shown in Figure 3, at each incoming edge of the CFG, we maintain an input constraints map that captures all constraints incoming from predecessor nodes in the CFG. Constraints are represented as a mapping from abstract instruction identifiers to a set of feasible abstract values for that instruction's destination operand. Similarly, at each outgoing edge of the CFG, we maintain an output constraints map, which might be different from the input constraints map if new constraints are being generated within the CFG node.

TID bitmaps Apart from the constraint maps, at each incoming and outcoming edge of the CFG, we also maintain thread-identification (TID) bitmaps. These bitmaps contain a single bit for each thread in the application. If the bit is 1, then we infer that the corresponding CFG node will be executed by that thread. The TID bitmaps allow us to prune CFG nodes that will not be executed by any thread or some threads. Since the TID bitmaps are propagated from parent to child nodes, we effectively propagate the information to all concerned nodes.

Abstract Memory Map For the main memory, we maintain an Abstract Memory Map which is a mapping from an abstract address to a set of abstract values potentially found in that memory location. In multi-threaded regions of the application, we will perform weak updates on Abstract Memory Map.

Instruction Operation Information For each instruction in the application, we conceptually maintain a vector of abstract domain values. The vector has one entry for each thread executing that instruction and captures the abstract representation of the instruction's operands and operation. We perform our abstract evaluation using this information.

4.3. Algorithm

An overview of the algorithm is presented below:

$prog \leftarrow [main(), thread\_worker\_func1()...]$
$func \leftarrow next(prog)$
repeat
  for all *BasicBlock BB in* $func$ **do**
    for *Instr i in BB* **do** $Evaluate(i)$
    end for
  end for
until fixpoint

At the end of Inlining phase, the multi-threaded program has been decomposed into a list of high-level functions, including `main()`. For each function, the analyzer will visit each basic block in its CFG. At each basic block, the analysis will check for any incoming constraints from predecessor nodes, and propagate the constraints if any. Within each basic block, the analyzer will examine each instruction and evaluate the instruction in the abstract domain, as represented by the $Evaluate$ function. This function conceptually performs the following actions:

- Convert each operand into its abstract equivalent.
- Check if any operand has any constraints in the incoming constraints map for the current basic block. Apply the constraints if any.
- Evaluate the instruction in the abstract domain by expressing the instruction's operation as a function of the abstract domain's transformers. For example, an ADD opcode is expressed as a addition operation in the abstract domain.
- If any new constraints are generated by the instruction, add them to the outgoing constraints map.
- If the instruction is a load/store, then add the load-/store address to each thread's read-/write-set.

This process is repeated till we reach a fix-point on all instructions in all basic blocks of each function.

4.3.1. Propagating constraints Constraints on values of variables need to be propagated through the CFG. At each incoming edge of a CFG node, the incoming constraints will be the union of all constraints from all predecessor nodes in the CFG. At each outgoing edge of a CFG node, the outgoing constraints will be the intersection of all incoming constraints with any newly generated constraints within the CFG node.

4.4. Post-Processing

At the end of the Analysis phase, we have sets of abstract read-/write-addresses for each thread. In the post-processing phase, we enumerate these addresses

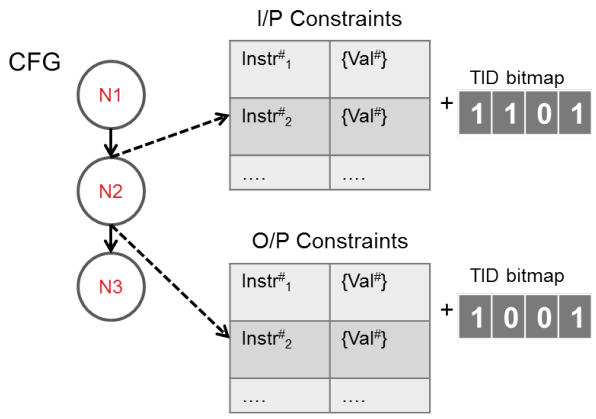

Figure 3: I/p and O/p Constraint Maps on CFG nodes

and check for intersections between read-write and write-write sets of different threads. The computed conflicts and collected read-/write-sets are provided as output to the programmer. We envision that this information could be consumed by a parallel runtime engine to identify data races and automatically insert locks around conflicting accesses.

5. Case-Studies

We now look at a couple of simple programs as case-studies and describe the behavior of our algorithm on these programs.

5.1. Case-Study 1: Disjoint Access Patterns

The following C++ code shows a toy worker function.

Listing 1: Disjoint Access Patterns

```
#define NUMTHREADS 2
int A[NUMTHREADS]
void* worker(int tid)
{
    for(int i=0; i<1; ++i)
        A[tid]=0;
}
```

Each thread assigns the value 0 to the index of *A* corresponding to its *tid* value. For simplicity we assume that the *tid* variable is directly passed as an argument to the worker function, even though in an actual program, the semantics of this operation would be different. Figure 4 shows the translation of the C++ source into the LLVM IR after our pre-processing passes. Note that, we have annotated each instruction with an instruction number. Figure 5 shows the output of our analysis for each instruction in the IR after the first iteration. No explicit result is calculated for branch instructions, instead the constraints are propagated as shown. At the *store* instruction, the additions to the write set of each thread are also shown.

There are 3 main points to note in Figure 5.

- At instruction 3, the conditional *br* instruction computes constraints on the value of the *%i1* virtual register. Specifically, on the true path, *%i1* < 1 and on the false path, *%i1* >= 1. The true and false constraints computed are passed to the respective successor basic blocks, *for.body* and *for.end*
- At instruction 4, we detect the *tid* variable in the *sext* instruction and assign a unique value to it in each thread. Thus, both threads access different regions of memory at the *gep* instruction that follows. Subsequently, the write sets of thread 0 and thread 1 are also correctly identified to be distinct.
- At instruction 8, the *add* instruction causes a strong update to the value of *%i* virtual register as this represents a variable that is private to each thread.

5.2. Case-Study 2: Weak Updates on Memory

The above example showed a case where our analyzer was able to precisely determine the write-/read-sets of each thread. However, we have come across certain cases where the over-approximations computed by our analyzer are too imprecise to be useful. The following C++ program shows an example of such a case where a thread initially writes the value 0 to the index of the global array *A* corresponding to its *tid* value. The thread then increments the value at that index.

Listing 2: Weak Updates on Memory

```
#define NUMTHREADS 2
int A[NUMTHREADS]
void* worker(int tid)
{
        A[tid]=0;
        A[tid]++;
}
```

The corresponding LLVM IR is shown in Figure 6. Figure 5 shows the output, during the first and second iterations of our analysis at each instruction in the IR. The values read from memory at the *load* instruction are highlighted in red.

From the C++ source code, it is evident that at the end of the program, the value at *A[tid]* is 1. However, as we perform weak updates in multi-threaded regions, the value read at the *load* instruction in iter-

```
entry:
  0. br label %for.cond

for.cond:preds = %for.inc, %entry
  1. %i.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
  2. %cmp = icmp slt i32 %i.0, 1
  3. br i1 %cmp, label %for.body, label %for.end

for.body:preds = %for.cond
  4. %idxprom = sext i32 %tid to i64
  5. %arrayidx = getelementptr inbounds [2 x i32],
               [2 x i32]* @A, i64 0, i64 %idxprom
  6. store i32 0, i32* %arrayidx, align 4
  7. br label %for.inc

for.inc:preds = %for.body
  8. %inc = add nsw i32 %i.0, 1
  9. br label %for.cond

for.end: = %for.cond
  10. ret i32 undef
```

Figure 4: LLVM IR for case-study 1

Instruction	Iteration 0	
	Thread 0	Thread 1
0	-	-
1	[0]	[0]
2	[0]	[0]
3	Propagate T (*%i1* < 1) F (*%i1* >= 1)	Propagate T (*%i1* < 1) F (*%i1* >= 1)
4	[0]	[1]
5	A[0]	A[4]
6	Add A[0] to WS of thread 0	Add A[4] to WS of thread 1
7	Propagate *%i1* < 1	Propagate *%i1* < 1
8	[1]	[1]
9	Propagate *%i1* < 1	Propagate *%i1* < 1
10	-	-

Figure 5: Output of M-TAGS at each instruction in IR of case-study 1

ation 2 is seen to be either 0 or 1. The following *inc* thus determines a value of either 1 or 2 as the result of the operation. As this is different from the result of the corresponding *inc* operation in iteration 0, we do not hit a fixed point and the analyzer proceeds to the next iteration of the analysis. The number of iterations of the analyzer is thus potentially unbounded and so the value loaded/stored into *A[tid]* is eventually determined to be the **TOP** of our abstract domain. In the future, we could implement some form of loop recognition as part of our LLVM pre-processing phase and avoid this problem.

6. Results

We evaluated the utility of our analyzer across 6 different workloads. These include 5 microbenchmarks and 1 benchmark from the PARSEC benchmark suite [2].

```
entry:
 0. %idxprom = sext i32 %tid to i64
 1. %arrayidx = getelementptr inbounds [2 x i32],
              [2 x i32]* @A, i64 0, i64 %idxprom
 2. store i32 0, i32* %arrayidx, align 4
 3. %idxprom1 = sext i32 %tid to i64
 4. %arrayidx2 = getelementptr inbounds [2 x i32],
              [2 x i32]* @A, i64 0, i64 %idxprom1
 5. %0 = load i32, i32* %arrayidx2, align 4
 6. %inc = add nsw i32 %0, 1
 7. store i32 %inc, i32* %arrayidx2, align 4
 8. ret i32 undef
```

Figure 6: LLVM IR for case-study 2

Instruction	Iteration 0		Iteration 1	
	Thread 0	Thread 1	Thread 0	Thread 1
0	[0]	[1]	[0]	[1]
1	A[0]	A[4]	A[0]	A[4]
2	Add A[0] to WS of thread 0	Add A[4] to WS of thread 1	Add A[0] to WS of thread 0	Add A[4] to WS of thread 1
3	[0]	[1]	[0]	[1]
4	A[0]	A[4]	A[0]	A[4]
5	[0] Add A[0] to RS of thread 0	[0] Add A[4] to RS of thread 1	[0, 1] Add A[0] to RS of thread 0	[0, 1] Add A[4] to RS of thread 1
6	[1]	[1]	[1, 2]	[1, 2]
7	Add A[0] to WS of thread 0	Add A[4] to WS of thread 1	Add A[0] to WS of thread 0	Add A[0] to WS of thread 0
8	-	-	-	-

Figure 7: Output of M-TAGS at each instruction in IR of case-study 2

Figure 8 shows the memory access pattern of our first benchmark - Array manipulation with no overlap. Multiple threads access a global array without any overlap in their memory access regions. Our second workload is Array manipulation with partial overlap where multiple threads access a global array, but with partial overlap in the read-/write-sets of threads accessing adjacent chunks as shown in Figure 9. Workloads 3 and 4 are similar to workloads 1 and 2 respectively, but the global data structure is a 2-dimensional matrix instead of an array, as shown in Figure 10 and Figure 11.

The final microbenchmark is an example of differentiated worker-threads where there is a single reader and multiple writers to a global array. All threads execute the same worker function, but depending on the *tid* variable, the reader and the writers execute different portions of the worker function. A thread-sensitive analysis can detect this behavior and restrict the analysis for the reader thread.

In addition, we also considered Black-Scholes from the PARSEC benchmark suite. The Black-Scholes application computes prices for a set of options by solving a partial differentiation equation for each option. Typically, each thread operates on some subset of the options. But there is no overlap in the work performed by each thread and so there are no memory access conflicts between the different

Figure 8: Array manipulation with no overlap

Figure 9: Array manipulation with partial overlap

threads.

The workloads mentioned above were analyzed using a prototype implementation of M-TAGs on LLVM v3.8 [6]. The read-/write-sets calcuated by M-TAGs were verified to be over-approximate representations of each benchmarks. The number of read-write and write-write conflicts detected across all threads by M-TAGs and a thread-insensitive approach is shown in Table 1. We can see that for each benchmark, the thread-insensitive approach considers the entire data-structure to be operated on by all threads in the system. On the other hand, M-TAGs correctly detects cases where there is no overlap across threads; and where there is some partial overlap, M-TAGs computes an over-approximate set of conflicts. For Black-Scholes, the thread-insensitive approach does not complete within a maximum time-bound, while M-TAGs arrives at the correct result that there are no conflicts across different worker threads. Hence, if the results from M-TAGs are consumed by a parallel execution runtime engine, it can avoid unnecessary synchronization, potentially improving the performance of the parallel program.

7. Related Works

Extensive research has been carried out in utilizing static analysis for the purpose of detecting bugs in multi-threaded programs or extracting useful information on the runtime behavior of a parallel computation. RacerX [3] uses flow-sensitive, interprocedural analysis to infer a wide range of information on the execution of a multithreaded program such as which operations are carried out in mutually exclusive regions and which code contexts are

Figure 10: Matrix manipulation with no overlap

Figure 11: Matrix manipulation with partial overlap

multithreaded. von Praun and Gross [15] consider the problem of tracking thread memory accesses in an object-oriented domain. They introduce an abstraction called the Object Use Graph (OUG), which statically captures accesses from different threads to objects. This is used to define a partial order of access events relevant to each runtime object. The success of these works show the potential of static analysis in helping programmers reason about the behavior of multi-threaded programs. However, the aforementioned analyses compute more detailed information about the runtime behavior of a program that what we are interested. The hints provided to a parallel programming framework need not be as comprehensive as the runtime-engine ensures the correctness of program execution. Thus, our work has focused on producing simpler hints, such as the per-thread read-/write sets which are less compute-intensive to determine. Eraser [12] is one of the most influential works in the literature for techniques to ensure the correctness of program. Eraser uses binary rewriting to instrument the executable to monitor shared-memory references and detect data races dynamically. As a result, this causes a huge slowdown in applications by a factor of 10x to 30x. However, the authors note that static analysis could greatly reduce this overhead by detecting potentially dangerous shared accesses and only instrumenting these.

8. Conclusion

Parallel programming has long been confined to only expert programmers. With the rise in multi-core processors, it is becoming increasingly important to help average programmers bridge the skill gap so as to better exploit the parallel computing power of today's machines. Parallel programming frameworks that hide some of the complexity of parallel programming are a step in this direction. In this work, we have built a thread-sensitive static analysis tool which further assists programmers who utilize such frameworks. For each thread in the program, our tool computes an over-approximation of the thread's read and write sets in an abstract domain. It also identifies conflicts in the memory access regions of the different threads. The output of our tool can be fed as hints to a parallel programming framework. Furthermore, using dedicated micro-benchmarks, and a real-world benchmark from the PARSEC suite, we have shown the utility of our tool in comparison to a thread-insensitive analyzer. In future, we hope to

Benchmark	Thread-Insensitive Analysis	M-TAGs
Array Manipulation (no overlap)	256 (length of array)	0
Array Manipulation (partial overlap)	256 (length of array)	3
Differentiated Worker Threads	256 (length of array)	Wr-Wr = 0, Wr-Rd = Wr-set of each writer
Matrix Manipulation (no overlap)	32 (Whole matrix)	0
Matrix Manipulation (partial overlap)	32 (Whole matrix)	24
Black-Scholes	Does not complete	0

Table 1: Number of read-write and write-write conflicts detected

take into account other parallel programming constructs such as thread barriers to further refine the precision of our analysis. We hope this work paves the way for more tools that cater to emerging parallel programming frameworks.

9. Acknowledgements

We would like to thank Prof. Thomas Reps of the University of Wisconsin-Madison for his guidance and feedback throughout the project. We also thank Rathijit Sen and Gagan Gupta for providing the abstract domain library and related infrastructure.

References

[1] C. Arts, "Cilk++."

[2] 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*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 72–81.

[3] D. Engler and K. Ashcraft, "Racerx: Effective, static detection of race conditions and deadlocks," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 237–252.

[4] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in *ACM Sigplan Notices*, vol. 33, no. 5. ACM, 1998, pp. 212–223.

[5] G. Gupta and G. S. Sohi, "Semantically ordered, parallel execution of multiprocessor programs," *Reason*, vol. 3, no. F4, p. F5.

[6] 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*. IEEE, 2004, pp. 75–86.

[7] D. Lea, "A java fork/join framework," in *Proceedings of the ACM 2000 conference on Java Grande*. ACM, 2000, pp. 36–43.

[8] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, 2006.

[9] D. Leijen, W. Schulte, and S. Burckhardt, "The design of a task parallel library," in *Acm Sigplan Notices*, vol. 44, no. 10. ACM, 2009, pp. 227–242.

[10] C. Pheatt, "Intel® threading building blocks," *Journal of Computing Sciences in Colleges*, vol. 23, no. 4, pp. 298–298, 2008.

[11] R. Rugina and M. C. Rinard, "Pointer analysis for structured parallel programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 25, no. 1, pp. 70–116, 2003.

[12] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, Nov. 1997.

[13] R. Sen and Y. Srikant, "Executable analysis using abstract interpretation with circular linear progressions," in *Formal Methods and Models for Codesign, 2007. MEMOCODE 2007. 5th IEEE/ACM International Conference on*, May 2007, pp. 39–48.

[14] K. Taura, K. Tabata, and A. Yonezawa, *Stackthreads/mp: integrating futures into calling standards*. ACM, 1999, vol. 34, no. 8.

[15] C. von Praun and T. R. Gross, "Static conflict analysis for multi-threaded object-oriented programs," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI '03. New York, NY, USA: ACM, 2003, pp. 115–128.