

Verification of Concurrent Programs Using Hybrid Concrete-Symbolic Interpretation

Emily Tucker and Louis-Noël Pouchet

Colorado State University

{Emily.Tucker,Louis-Noel.Pouchet}@colostate.edu

Abstract. Source-level transformations of programs are fundamental to achieve high-performance: complex loop transformations, including loop tiling and parallelization, must typically be applied by a user or an automated tool. However this process is error-prone, especially when combined with transformations of the data layout, code structure and statements themselves.

In this work, we present an approach to prove the equivalence between a function and its candidate optimized version which is mostly agnostic to the schedule and storage implemented. It can prove the equivalence between a sequential function and its parallelized version, under practical restrictions.

Keywords: Verification · Program Equivalence · Concurrent Programs.

1 Introduction

Optimizing compilers must provide a semantically equivalent implementation to the input program being compiled, while optimizing the program description to efficiently map it onto a target hardware. Typically, *parallelism* should be exposed, be it at the instruction level, using SIMD vectorization, or using coarser-grain multi-thread parallelization. This parallelism may be automatically exposed by the compiler, or be assisted by the programmer’s rewriting and annotations of the program to specify how parallelism may be implemented. An upside is the potential performance of parallel programs versus their sequential implementation. A downside is the risk for the compilers and humans alike to be buggy: such parallelizing transformations may break the original program dependences, be incorrect due to possible race conditions or deadlocks, or simply not following the sequential program semantics due to mismatches between the operations computed in the original and transformed programs.

In this work, we present a verification approach to prove the correctness of a set of parallelizing transformations. It is based on a hybrid concrete-symbolic interpreter, to verify the correctness of a pair of programs at the source level [26]. These programs, expressed in C semantics, may contain OpenMP directives for parallelization. Our tool can prove the absence of race conditions and deadlocks in each of them, as well as prove the semantic equivalence between them: that is, for any input, they both produce the exact same output, necessarily.

The problem of determining the absence of races or deadlocks in parallel programs has been studied from multiple angles ranging from static analyses e.g. [34,5,38,7], dynamic analyses [3,8] including intercepting the OpenMP runtime [15], symbolic analyses [32,30,26], as well as using Coq-formalized proofs [11]. Program equivalence itself has been studied from these angles, including for affine programs [36,4], and by symbolic execution [28,32,17] and concrete-symbolic interpretation [26]. We develop an approach which relies on concrete interpretation of the control-flow instructions, therefore limited to a class of programs: those with *statically interpretable control-flow* (SICF) [26]. In turn, this enables a program equivalence approach that has linear time and space complexity with respect to the number of operations *executed* by the program, in a manner fully independent from the syntax used, schedule of operations, and storage implemented [26]. Extending to support parallel programs incurs a low-overhead additional complexity, and can prove for sequentially consistent programs the absence of parallelization errors, and full equivalence between a parallel and a sequential program. We make the following contributions:

- We outline how our hybrid concrete-symbolic interpreter proceeds to verify the correctness of some forms of concurrent programs.
- We introduce a translation of several OpenMP constructs into this framework, enabling the detection of race conditions for a class of OpenMP programs.
- We present experimental results demonstrating the ability of our system to catch concurrency bugs in programs generated by an auto-parallelizing compiler, and prove correct the parallelization implemented otherwise.

2 Background and Overview

We first summarize key aspects of the CDAG-based verification approach [26], before developing the ideas for the verification of a class of concurrent programs.

2.1 Hybrid Concrete-Symbolic Verification for SICF programs

Our approach is based on *concrete* interpretation of selected instructions in the input program, while treating any other operation as *symbolic* and building a symbolic representation for these based on Computation Directed Acyclic Graph (CDAG) [18,13,24]. Specifically, any conditional branch instruction, and the operations transitively involved in computing the value of these condition(s), as well as operations involved in address calculations, need to be concretely interpretable from the input program. Any other operation can be symbolic: the concrete value of their operands need not be known at interpretation time. This leads to a class of statically interpretable control-flow programs we support: those where all branches to be taken by the program (i.e., the control-flow) when executing on the target machine can be discovered by concrete interpretation of the relevant operations from the input program.

We remark a relation with Static Control-Flow Programs (SCoP) [14], known as affine programs [22,27]: for these programs all branches to be taken can be

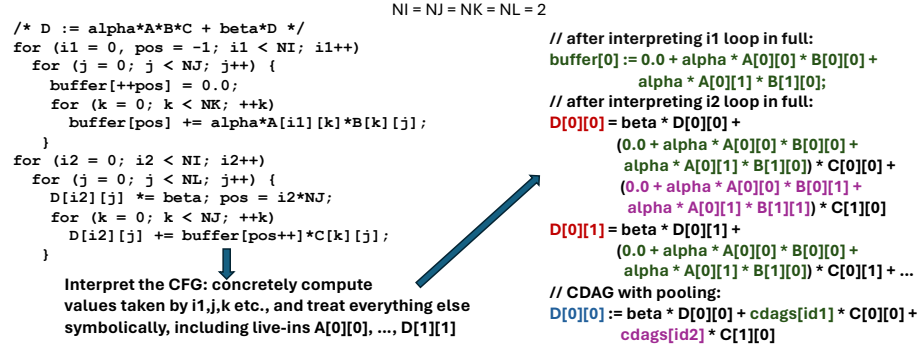
exactly characterized by *static analysis* and modeled using affine relations. SICF programs do not require any affine structure, do not depend on how loops and control-flow are implemented, but does require to know the actual problem size (e.g., loop bound) of SCoPs. Parametric loop bounds are not supported as we require the concrete values for each conditional branch to be taken by the interpreter. All SCoPs with known values for the parameters are SICF.

Another important restriction of our approach is the requirement to map each memory cell that is live-in/live-out to a unique name, to be used to reason on the memory values of both programs. We therefore typically target the verification of a pair of functions f_{orig}, f_{opt} (themselves possibly calling other functions which are also interpreted) such that f_{opt} is an optimized version of f_{orig} , meant to replace it in a larger program. They would therefore inherit the same *execution context* necessarily, making the verification possible [12].

Illustrative Example We present below a simple example of our CDAG-based concrete-symbolic interpretation. The interpreter is equipped with a *concrete evaluator* which implements exactly the same concrete semantics as the target machine, making simplification of expressions by concrete interpretation valid. Every operation that can be concretely interpreted is; otherwise, the operation is promoted to a symbolic CDAG representation. CDAGs are schedule-independent and storage-independent, and represent the ordered set of operations that produce a value as a function of only live-in symbolic values and constants [18,26].

Equivalence is achieved by showing that for the same live-out memory cells of both programs, the CDAGs constructed for them for f_{orig} and f_{opt} are semantically equivalent. In its simplest form, they can be strictly identical (making this check linear time), or possibly some semantics-preserving rewrites of the CDAGs may be needed first, e.g. to support associativity and commutativity [26].

CDAGs are used to reason on I/O lower bounds, given they are agnostic to scheduling and storage implemented. They represent the fundamental nature of the computation, not how it is implemented [18]. However as seen below they can grow exponentially: the values used for the second matrix-multiply are themselves a CDAG. To control complexity and build a representation during interpretation that is linear in space wrt. the operations executed, we deploy memory pooling of distinct (sub-)CDAGs, and use pointers to them. For multiple references to the same memory cell, only a pointer to its CDAG is duplicated.



```

/* D := alpha*A*B*C + beta*D */
for (i1 = 0, pos = -1; i1 < NI; i1++)
  for (j = 0; j < NJ; j++) {
    buffer[++pos] = 0.0;
    for (k = 0; k < NK; ++k)
      buffer[pos] += alpha*A[i1][k]*B[k][j];
  }
for (i2 = 0; i2 < NI; i2++)
  for (j = 0; j < NL; j++) {
    D[i2][j] *= beta; pos = i2*NJ;
    for (k = 0; k < NJ; ++k)
      D[i2][j] += buffer[pos++] * C[k][j];
  }

```

Interpret the CFG: concretely compute values taken by i1,j,k etc., and treat everything else symbolically, including live-ins A[0][0], ..., D[1][1]

```

// after interpreting i1 loop in full:
buffer[0] := 0.0 + alpha * A[0][0] * B[0][0] +
           alpha * A[0][1] * B[1][0];
// after interpreting i2 loop in full:
D[0][0] = beta * D[0][0] +
(0.0 + alpha * A[0][0] * B[0][0] +
 alpha * A[0][1] * B[1][0]) * C[0][0] +
(0.0 + alpha * A[0][0] * B[0][1] +
 alpha * A[0][1] * B[1][1]) * C[1][0]
D[0][1] = beta * D[0][1] +
(0.0 + alpha * A[0][0] * B[0][0] +
 alpha * A[0][1] * B[1][0]) * C[0][1] + ...
// CDAG with pooling:
D[0][0] := beta * D[0][0] + cdags[id1] * C[0][0] +
cdags[id2] * C[1][0]

```

2.2 Extending to Support Concurrent Programs

Prior work generalized this sequential verification approach to handle limited forms of concurrency: specifically program regions executing concurrently, synchronized using blocking FIFOs [26]. This support enables the verification of source-to-source transformations for high-performance accelerator designs targeting High-Level Synthesis toolchains, enabling for instance the verification of correctness of a systolic array implemented over 140k LoCs, using a 64x64 array of concurrent units [37] (8k Processing Elements in total, interconnected using 16k FIFOs) in about 15 minutes, using a single CPU core [26].

We build on this approach to enable a preliminary support of some OpenMP parallelization of C programs, as described in Sec. 3. In a nutshell, the verification of concurrent programs works as follows. (1) The interpreter is extended with support for concurrent regions, which can be scheduled for interpretation akin to a multiprogramming approach in operating systems, switching between concurrent regions ready to execute. It implements interrupts (e.g., when a blocking FIFO is not ready) and regions can be ready to execute, executing, waiting (on FIFO readiness) or terminated. (2) It maintains virtual timestamps for the operations interpreted, capturing the earliest (virtual) time at which blocks of operations can execute, and associate each shared memory accesses with such timestamps. (3) It maintains histories of the (shared) memory accesses, enabling a check for non-determinism in case of accesses to the same memory location at the same virtual timestamp: if two possible valid concurrent schedules lead to a different memory state, the interpreter aborts. We use these features to develop a verification approach for a set of C+OpenMP parallel programs.

3 Verifying a Set of Concurrent Programs

We now detail our approach to verifying the subset of concurrent programs supported in this work. Specifically, we target the verification of a pair of functions f_{orig}, f_{opt} such that one is a substitute for the other in a larger program, both expressed as source-level C programs, possibly containing a subset of OpenMP pragmas as described in Sec. 4. If successful, the verification approach proves that both programs have the same semantics in that they both compute the same result, if given the same inputs; and that for any synchronization-preserving concurrent schedule following the OpenMP pragmas specification, no race condition nor deadlock can occur. As shown in Sec. 5, this enables the verification of the correctness of source-to-source parallelizing compilers such as Pluto and PoCC which rely on inserting `#pragma omp parallel for` around parallel affine loops, including whether variables are properly privatized, whether complex tiled wavefront parallelization was properly implemented, etc.

3.1 Detecting Non-Determinism

A key aspect of our approach is to significantly limit the form of memory consistency we support. During interpretation, if two operations (from two different

logical threads) can occur at the same timestamp, and they both access the same shared memory location, then neither of these two operations can be a write. Otherwise the result may be non-deterministic: depending on which of the two operations in practice would execute first, the state of memory may be different. We perform such detection for both concrete and symbolic memory locations.

If two accesses (one being a write) to the same shared memory location need to be performed, we require a synchronization (such as, but not limited to, a barrier) between them, forcing sequentially consistent behavior for these two accesses. In practice, the virtual timestamp we maintain to determine which operations may execute at the same time typically changes only when a synchronization is interpreted, as it ensures every operation depending on it is executed after operations leading to it.

3.2 Verified Properties for Concurrent Programs

To verify a class of OpenMP programs, we assign *every* block of code that may execute in parallel (as per the OpenMP pragmas) to a distinct *logical* thread, as discussed in Sec. 4. For example, an `omp for` loop with 1000 iterations can be represented with 1000 logical threads. Therefore all possible serializations of these logical threads (e.g., different OpenMP schedules and their mapping to physical threads) will only reduce the amount of parallelism considered, making the analysis conservative but safe.

To model synchronizations, we rely on our own API to handle interrupts/wait/resume for logical threads. Note we assume every operation within a thread executes serially, in the *source program* order. This facilitates debugging at the source level, however it does not model the effect of compiler optimizations in the binary eventually executed, which may include reordering of operations.

Definition 1 (Virtual timestamp and synchronization). *Given two operations o_1, o_2 executed by two logical threads l_1, l_2 . A virtual timestamp $t(o)$ is assigned to every operation, so that $t(o_1) < t(o_2)$ iff there exists a point-to-point synchronization operation $o_s : l_1 \rightarrow l_2$, and o_1 executes before o_s in l_1 's sequential order, and o_2 executes after o_s in l_2 's.*

This relation amounts to the happens-before relation [15], that is, the necessity for all operations that execute serially before a point-to-point synchronization to all be terminated before the start of operations executing after this synchronization. Consequently, operations may have the same virtual timestamp, if they are not (transitively) ordered via synchronizations. Such operations may therefore execute at the same time, and we check whether they can produce a race by tracking the timestamp of operations accessing all non-thread-local data. We operate with a zero-latency model, that is we assume all operations execute in zero cycles, and only synchronizations can force a (partial) order between operations in different logical threads.

We detect the existence of a possible race condition conservatively, in this zero-latency model, by determining if there is any memory location accessible by two or more logical threads at the same time.

Definition 2 (Read-write conflict). *Given two operations o_1, o_2 executed by two logical threads l_1, l_2 such that $t(o_1) = t(o_2)$, which both access the same shared (non-local) memory location, and one of these accesses is a write. Then the operations expose a read-write conflict.*

It is also possible in general to create a deadlock situation, where point-to-point synchronizations are incorrectly implemented leading to a blocking state, that is, one or more logical thread cannot terminate.

Definition 3 (Deadlock). *Given a logical thread l in a blocking/interrupted state. If there is no other logical thread that can become active by interpretation, and which can modify the semaphore(s) on which l is blocked, then l is deadlocked.*

For our system to output a conclusive analysis of a program, it must reach termination without error. Without loss of generality, we assume every program is encapsulated in a function, and a single exit point exists for this function.

Property 1 (Termination). If the concrete interpretation of the control-flow and dataflow addressing reaches the main function’s exit point, no read-write conflict was detected, and no deadlock is detected, then the interpretation terminated.

Note we implement an inelegant yet practical approach to handle non-terminating programs (e.g. infinite loops): a counter of the number of operations interpreted. If a maximum limit threshold is reached, the interpretation aborts.

We conclude with the equivalence between a pair of programs f_{orig}, f_{opt} .

Property 2 (Equivalence between programs). Given f_{orig}, f_{opt} two functions such that one is a substitute for the other in a main program, and all their arguments are non-aliasing and referencing all destinations of side-effects. If their interpretation terminates, and for every live-out memory location referenced in their arguments, the concrete values or CDAGs produced for the same location are semantically equivalent, then the two functions are equivalent.

A proof of an equivalent version of this property is outlined in [26].

3.3 High-Level Verification Procedure

We summarize the verification process as follows. f_{orig}, f_{opt} are each independently interpreted, computing for each a final memory state M_f . For each function f , the interpreter proceeds as described in prior work [26], interpreting the code for a thread until termination, or interruption due to a blocking synchronization primitive. A scheduler context-switches to the next thread in the queue in case of interrupt, until there is no more interrupted nor active thread in the queue (otherwise a deadlock is occurring). During interpretation, we associate a thread-specific timestamp to every non-local memory element read/written by each thread, itself only increased when synchronizations are interpreted. If a location has been written by a thread, no other thread can read-write it at the same timestamp, otherwise a read-write conflict is produced. We conservatively

assume pointers to data initialized prior to the start of a concurrent region are shared between threads in this region. Pointer arithmetic besides thread-local declarations must lead to addressing these shared pointed locations, otherwise the interpreter aborts. As we operate at the source-level, scalar and array variables locally declared within a thread, and their scope of liveness, is trivially detected. Consequently, every variable that is not accessible at the end of the function is deleted from M_f . We do not support external functions (e.g., `libc`).

If interpretation terminated, we obtained $M_{f_{orig}}$ and $M_{f_{opt}}$ two memory states made only of variables accessible when the function terminates: live-in and live-out data [26]. As we restrict to programs where a unique name can be built for every memory cell, (e.g., $A[42][51]$, *foobar*, etc.), requiring a lack of aliasing on the main function parameters, we simply proceed by checking for every such name that the memory values computed (be they concrete, or symbolic CDAGs) are semantically equivalent, e.g. $M_{f_{orig}}(A[42][51]) \equiv M_{f_{opt}}(A[42][51])$. Note we support a variety of rewrite rules on CDAGs prior to checking equivalence, e.g. to normalize associative/commutative reductions [26].

4 OpenMP

4.1 API For Concurrent Programs

We now briefly present our API for interpreting concurrent programs, and the translation of some OpenMP constructs with it.

At its core, only three API functions are needed. First we declare blocks of code to execute concurrently within a parallel region `regionid`, using the `register_concurrent(regionid,block)` for each such block. When interpretation of the code block assigned to a thread reaches the end of its control-flow, the thread is terminated. Note there is an infinite number of logical threads available. Second, we declare a point-to-point synchronization with a semaphore, using `set_semaphore_value(regionid,semid,value)` and the associated blocking function `wait_until_semaphore_value(regionid,semid,value)`. These calls are inserted in the program as regular C function calls, and different threads may read/write the same semaphores. They form a sufficient flexible set of constructs (fork/join, and point-to-point synchronization) to emulate the OpenMP constructs we target in this work.

4.2 OpenMP Constructs

#pragma omp parallel A parallel OpenMP region, outside of any `for` loop, requires knowledge of the concrete number of threads to be verified. The code block dominated by the pragma is recorded for parallel execution, with one call to `register_concurrent(regionid,block)` per thread. Then, the block in the main program is replaced by a pair of calls to be executed by the main/-master thread: `concurrent_region_start(regionid)` and the associated join `concurrent_region_end(regionid)`, which emulates the fork/join OpenMP

model. When interpreting the **start** call, the interpreter creates the requested threads, and executes them in order, context-switching to the next thread only if the current thread has terminated, or is interrupted due to a blocking call (e.g., waiting on a semaphore to reach a particular value). The **end** call acts as a global barrier for the region, except the threads are deleted, this only once all have reached the end of their control-flow. Otherwise a deadlock is detected.

#pragma omp for By design the interpreter requires loop bounds to be concretely interpretable, from the statically-interpretable control-flow requirement. Consequently, the loop bound expressions are known and the code blocks for the concurrent region can be seamlessly updated with the set of iterations to be run by each thread, the code **block** becoming the loop body. We support the **static** schedule, as well as a conservative mode where the number of logical threads is dynamically increased to assign one loop iteration per thread. **private** and related clauses are translated to equivalent local variable declarations and initializations explicitly by pre-processing. A **barrier** is automatically inserted at the loop exit, unless the **nowait** clause is specified.

#pragma omp section OpenMP **sections** are translated similarly to parallel regions, with the proper code blocks registered for concurrent execution.

#pragma omp barrier Finally, a barrier is implemented via semaphores, similar to the point-to-point synchronization described above. Technically, we use a single collective synchronization API `wait_on_semaphores(regionid,sem_array)` interpreted by all threads, with each thread writing its own semaphore when it reached the API call, and interpretation the next instruction after the barrier (if any) being implemented only when all threads wrote their semaphore, to avoid ordering issues.

4.3 Extensions and Ongoing Work

In this work we limit our presentation to the simplest form of `doall` and `doacross` parallelism using OpenMP, for illustration purposes. Our ongoing work includes support for a large subset of the OpenMP 4 constructs, including tasks and their dependencies. Clearly, our flexible model for declaring concurrent regions and their synchronizations leads to easily "software-emulate" the various OpenMP constructs, by translating them to C code (e.g. to properly declare private variables, but also scheduling strategies) beforehand. As downside, this translation of pragmas we perform as pre-processing must itself be correct, and matching what the OpenMP-capable compiler implements when translating these pragmas.

Ongoing work also includes support for a subset of other parallel languages such as Habanero [6] and its C variant, essentially implementing a hybrid concrete-symbolic interpretation of program to compute the happens-before and may-happen-in-parallel relations in a manner agnostic to the syntax used to implement the program. Approaches using automated theorem proving have been developed [10], we aim for increased generality to how the code is implemented.

We will discuss this generalized support during our presentation, and its trade-offs versus other verification approaches. We also mention the issue of properly verifying runtime systems in charge of orchestrating such parallel computations. While they can be simply made part of the programs to interpret, it is often desirable to instead perform a slightly more complex interpretation of the program (e.g., using infinitely many logical threads to decompose maximally the concurrency available) to make the verification robust to a variety of concurrent schedules that can be implemented by these runtimes, including work-stealing.

5 Experimental Results

5.1 Experimental Setup

For brevity we limit to illustrating the 2mm benchmark from PolyBench/C [27], and will present extensive results over all PolyBenchs during our talk [26]. 2mm computes $\beta * D + \alpha * A * B * C$ for rectangular matrices A, B, C, D : the product of three matrices. The matrix data type is symbolic (and hence the verification holds for "any" data type to be used), and we vary the problem sizes and program transformations (including OpenMP parallelization) computed by the PoCC compiler [2]. All experiments are conducted on a single core of an AMD Ryzen 5900HX, using 64GB of DDR4 RAM, all optimizations within the interpreter are disabled except storing duplicate sub-CDAGs by pointers.

6 Results on 2mm benchmark

Table 1 displays the time to Interpret two programs: a sequential, base version and a transformed version optimized by PoCC using fusion, tiling and OpenMP parallelization. We display the the number of statement instances in the source code interpreted, and the time to compute equivalence (note the time to detect read-write conflicts, if any, is integrated in the interpretation time), the number of CDAG nodes created, and the maximal memory usage during the full process. All experiments model 8 logical threads. Time is barely sensitive to higher threads count, but it is influenced by the number of non-barrier synchronizations.

Table 1: Summary of experiments on PolyBench/2mm benchmark

Benchmark	Int. seq.	#stmts	Int. PoCC	#stmts	Equiv.	#nodes	Max Mem
2mm-128	30.4s	12M	32.8s	12.7M	2.6s	4.2M	3GB
2mm-200	106s	43M	116s	45M	10.3s	14M	10GB
2mm-32	0.50s	207k	0.54s	207k	0.04s	67k	53MB
2mm-32-ns	0.80s	207k	0.84s	207k	0.47s	67k	938MB
2mm-200-np	106s	43M	116s	45M	N/A	14M	10GB
2mm-200-bt	106s	43M	113s	44M	10.0s	14M	10GB

Table 1 reports two cases of equivalence for square problem size 128, and the rectangular MEDIUM polybench size (-200). Throughput is around 0.4M

statements per second. For 2mm, memory use can exceed 50GB for problem size of 512 [26], showing a scalability limit with this approach. Solutions to overcome this limit includes on-the-fly CDAG compression by affine folding [29], allowing interpretation to scale gracefully to "arbitrary" problem sizes. We also display the benefits of avoiding duplication of CDAG subtrees, with the `-ns` variant for `N=32` disabling this optimization. For this small problem size, memory usage is already 20x larger without this optimization, and would prevent scaling to more realistic problem sizes. We also report two bugs we manually introduced: a missing inner loop iterator in the private clause (`-np`) and a loop bound error in the tiling, leading to skip iterations (`-bt`). For both, the interpreter ran to completion to maximize the number of errors found.

7 Related Work

The detection of concurrency bugs and equivalence between two implementations are often split in two different problems. Detecting bugs in OpenMP programs has been effectively implemented [15], and various static analyses have also been proposed, e.g [7,38]. However none verifies at the same time the compliance of the parallelized program with the semantics of the *original* unoptimized program.

Our framework can prove the equivalence of C-style programs under a wide set of code transformations, albeit limited to the class of Statically Interpretable Control-Flow. An approach complementary to ours is KLEE [28,1,30]. KLEE implements a different symbolic interpretation approach; ours is specialized for equivalence of programs with a *single* concretely interpretable CFG path and concretely interpretable array subscripts, trading off generality for speed. We limit coverage to fixed problem sizes but can operate on a variety of symbolic data types, at order(s) of magnitude faster speed than KLEE(-float [20]).

Other approaches to verifying or guaranteeing the correctness of (parallel) programs include model checking e.g. [21,35,33,17], translation validation e.g. [23,9,25] and of course using a certified compiler e.g. [19,31,16].

8 Conclusion

Although numerous approaches exist to assess the correctness of a program optimization, they are often associated with fundamental limitations to how the program is implemented. By using a hybrid concrete-symbolic interpretation approach, and imposing sensible restrictions to support only programs with statically interpretable control-flow, it is possible to achieve significantly stronger guarantees than testing, while offering full flexibility for *how* the program is implemented: arbitrary statement transformations, schedule (including parallel ones), arbitrary storage and bufferization schemes, etc. In this work we outlined a verification approach for (concurrent) programs based on prior work [26], and simple extensions to handle OpenMP programs and verify their equivalence. Future work includes generalizing to a large subset of OpenMP 4, and improving scalability further using a nearly constant-space approach to encode CDAGs.

References

1. The KLEE symbolic execution engine (2023), <https://klee.github.io>
2. PoCC, the Polyhedral Compiler Collection 1.6 (2023), <https://pocc.sourceforge.net>
3. Atzeni, S., Gopalakrishnan, G., Rakamaric, Z., Ahn, D.H., Laguna, I., Schulz, M., Lee, G.L., Protze, J., Müller, M.S.: ARCHER: Effectively Spotting Data Races in Large OpenMP Applications. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 53–62 (May 2016)
4. Bao, W., Krishnamoorthy, S., Pouchet, L.N., Rastello, F., Sadayappan, P.: Poly-check: Dynamic verification of iteration space transformations on affine programs. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (2016)
5. Bora, U., Das, S., Kukreja, P., Joshi, S., Upadrasta, R., Rajopadhye, S.: LLOV: A Fast Static Data-Race Checker for OpenMP Programs. *ACM Transactions on Architecture and Code Optimization* **17**(4), 1–26 (Dec 2020)
6. Cavé, V., Zhao, J., Shirako, J., Sarkar, V.: Habanero-java: the new adventures of old x10. In: Proceedings of the 9th International Conference on Principles and Practice of Programming in Java. pp. 51–61 (2011)
7. Chatarasi, P., Shirako, J., Kong, M., Sarkar, V.: An extended polyhedral model for spmd programs and its use in static data race detection. In: Languages and Compilers for Parallel Computing: 29th International Workshop, LCPC 2016, Rochester, NY, USA, September 28–30, 2016, Revised Papers 29. pp. 106–120. Springer (2017)
8. Cheng, G.I., Feng, M., Leiserson, C.E., Randall, K.H., Stark, A.F.: Detecting data races in cilk programs that use locks. In: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures. pp. 298–309 (1998)
9. Clément, B., Cohen, A.: End-to-end translation validation for the halide language. *Proceedings of the ACM on Programming Languages* **6**(OOPSLA1), 1–30 (2022)
10. Cogumbreiro, T., Shirako, J., Sarkar, V.: Formalization of habanero phasers using coq. *Journal of logical and algebraic methods in programming* **90**, 50–60 (2017)
11. Cogumbreiro, T., Surendran, R., Martins, F., Sarkar, V., Vasconcelos, V.T., Grossman, M.: Deadlock avoidance in parallel programs with futures: why parallel tasks should not wait for strangers. *Proceedings of the ACM on Programming Languages* **1**(OOPSLA), 1–26 (2017)
12. Cousot, P.: Abstract interpretation based formal methods and future challenges. In: *Informatics: 10 Years Back, 10 Years Ahead*, pp. 138–156. Springer (2001)
13. Elango, V., Rastello, F., Pouchet, L.N., Ramanujam, J., Sadayappan, P.: On characterizing the data access complexity of programs. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (2015)
14. Feautrier, P.: Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. *International journal of parallel programming* **21**, 389–420 (1992)
15. Gu, Y., Mellor-Crummey, J.: Dynamic data race detection for openmp programs. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 767–778. IEEE (2018)
16. Herklotz, Y., Pollard, J.D., Ramanathan, N., Wickerson, J.: Formal verification of high-level synthesis. *Proceedings of the ACM on Programming Languages* **5**(OOPSLA), 1–30 (2021)

17. Jakobs, M.C.: PEQCHECK: Localized and Context-aware Checking of Functional Equivalence. In: 2021 IEEE/ACM 9th International Conference on Formal Methods in Software Engineering (FormaliSE). pp. 130–140 (May 2021)
18. Jia-Wei, H., Kung, H.T.: I/o complexity: The red-blue pebble game. In: Proceedings of the thirteenth annual ACM symposium on Theory of computing. pp. 326–333 (1981)
19. Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., Ferdinand, C.: CompCert-a formally verified optimizing compiler. In: ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress (2016)
20. Liew, D., Schemmel, D., Cadar, C., Donaldson, A.F., Zahl, R., Wehrle, K.: Floating-point symbolic execution: a case study in n-version programming. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 601–612. IEEE (2017)
21. Mercer, E.G., Anderson, P., Vrvilo, N., Sarkar, V.: Model checking task parallel programs using gradual permissions (n). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 535–540. IEEE (2015)
22. Moses, W.S., Chelini, L., Zhao, R., Zinenko, O.: Polygeist: Raising c to polyhedral mlir. In: 2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT). pp. 45–59. IEEE (2021)
23. Necula, G.C.: Translation validation for an optimizing compiler. In: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation. pp. 83–94 (2000)
24. Olivry, A., Langou, J., Pouchet, L.N., Sadayappan, P., Rastello, F.: Automated derivation of parametric data movement lower bounds for affine programs. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (2020)
25. Parthasarathy, G., Dardinier, T., Bonneau, B., Müller, P., Summers, A.J.: Towards trustworthy automated program verifiers: Formally validating translations into an intermediate verification language. Proceedings of the ACM on Programming Languages **8**(PLDI), 1510–1534 (2024)
26. Pouchet, L.N., Tucker, E., Zhang, N., Chen, H., Pal, D., Rodríguez, G., Zhang, Z.: Formal verification of source-to-source transformations for hls. In: Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays. pp. 97–107 (2024)
27. Pouchet, L.N., Yuki, T.: Polybench/c 4.2.1 (2023), <https://polybench.sourceforge.net>
28. Ramos, D.A., Engler, D.: {Under-Constrained} symbolic execution: Correctness checking for real code. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 49–64 (2015)
29. Rodríguez, G., Pouchet, L.N., Touriño, J.: Representing integer sequences using piecewise-affine loops. Mathematics **9**(19), 2368 (2021)
30. Schemmel, D., Büning, J., Rodríguez, C., Laprell, D., Wehrle, K.: Symbolic partial-order execution for testing multi-threaded programs. In: International Conference on Computer Aided Verification. pp. 376–400. Springer (2020)
31. Ševčík, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., Sewell, P.: CompCerttso: A verified compiler for relaxed-memory concurrency. Journal of the ACM (JACM) **60**(3), 1–50 (2013)
32. Siegel, S.F., Zheng, M., Luo, Z., Zirkel, T.K., Marianiello, A.V., Edenhofner, J.G., Dwyer, M.B., Rogers, M.S.: CIVL: The concurrency intermediate verification language. In: Proceedings of the International Conference for High Performance Com-

- puting, Networking, Storage and Analysis. pp. 1–12. ACM, Austin Texas (Nov 2015)
33. Siegel, S.F., Zheng, M., Luo, Z., Zirkel, T.K., Marianiello, A.V., Edenhofner, J.G., Dwyer, M.B., Rogers, M.S.: Civl: the concurrency intermediate verification language. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–12 (2015)
 34. Swain, B., Li, Y., Liu, P., Laguna, I., Georgakoudis, G., Huang, J.: OMPRacer: A Scalable and Precise Static Race Detector for OpenMP Programs. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–14 (Nov 2020)
 35. Vechev, M., Yahav, E., Raman, R., Sarkar, V.: Automatic verification of determinism for structured parallel programs. In: Static Analysis: 17th International Symposium, SAS 2010, Perpignan, France, September 14–16, 2010. Proceedings 17. pp. 455–471. Springer (2010)
 36. Verdoolaege, S., Janssens, G., Bruynooghe, M.: Equivalence checking of static affine programs using widening to handle recurrences. In: Computer aided verification (2009)
 37. Wang, J., Guo, L., Cong, J.: Autosa: A polyhedral compiler for high-performance systolic arrays on fpga. In: The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. pp. 93–104 (2021)
 38. Ye, F., Schordan, M., Liao, C., Lin, P.H., Karlin, I., Sarkar, V.: Using polyhedral analysis to verify openmp applications are data race free. In: 2018 IEEE/ACM 2nd International Workshop on Software Correctness for HPC Applications (Correctness). pp. 42–50. IEEE (2018)