BMC for Weak Memory Models: Relation Analysis for Compact SMT Encodings

Natalia Gavrilenko[1,4(✉)], Hernán Ponce-de-León[2], Florian Furbach[3], Keijo Heljanko[4], and Roland Meyer[3]

[1] Aalto University, Helsinki, Finland
[2] fortiss GmbH, Munich, Germany
[3] TU Braunschweig, Brunswick, Germany
[4] University of Helsinki and HIIT, Helsinki, Finland
natalia.gavrilenko@helsinki.fi

Abstract. We present Dartagnan, a bounded model checker (BMC) for concurrent programs under weak memory models. Its distinguishing feature is that the memory model is not implemented inside the tool but taken as part of the input. Dartagnan reads CAT, the standard language for memory models, which allows to define x86/TSO, ARMv7, ARMv8, Power, C/C++, and Linux kernel concurrency primitives. BMC with memory models as inputs is challenging. One has to encode into SMT not only the program but also its semantics as defined by the memory model. What makes Dartagnan scale is its relation analysis, a novel static analysis that significantly reduces the size of the encoding. Dartagnan matches or even exceeds the performance of the model-specific verification tools Nidhugg and CBMC, as well as the performance of Herd, a CAT-compatible litmus testing tool. Compared to the unoptimized encoding, the speed-up is often more than two orders of magnitude.

Keywords: Weak memory models · CAT · Concurrency · BMC · SMT

1 Introduction

When developing concurrency libraries or operating system kernels, performance and scalability of the concurrency primitives is of paramount importance. These primitives rely on the synchronization guarantees of the underlying hardware and the programming language runtime environment. The formal semantics of these guarantees are often defined in terms of weak memory models. There is considerable interest in verification tools that take memory models into account [5,9,13,22].

A successful approach to formalizing weak memory models is CAT [11,12,16], a flexible specification language in which all memory models considered so far can be expressed succinctly. CAT, together with its accompanying tool Herd [4],

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 355–365, 2019.
https://doi.org/10.1007/978-3-030-25540-4_19

has been used to formalize the semantics not only of assembly for x86/TSO, Power, ARMv7 and ARMv8, but also high-level programming languages, such as C/C++, transactional memory extensions, and recently the Linux kernel concurrency primitives [11,15,16,18,20,24,29]. This success indicates the need for universal verification tools that are not limited to a specific memory model.

We present Dartagnan [3], a bounded model checker that takes memory models as inputs. Dartagnan expects a concurrent program annotated with an assertion and a memory model for which the verification should be conducted. It verifies the assertion on those executions of the program that are valid under the given memory model and returns a counterexample execution if the verification fails. As is typical of BMC, the verification results hold relative to an unrolling bound [21]. The encoding phase, however, is new. Not only the program but also its semantics as defined by the CAT model are translated into an SMT formula.

Having to take into account the semantics quickly leads to large encodings. To overcome this problem, Dartagnan implements a novel *relation analysis*, which can be understood as a static analysis of the program semantics as defined by the memory model. More precisely, CAT defines the program semantics in terms of relations between the events that may occur in an execution. Depending on constraints over these relations, an execution is considered valid or invalid. Relation analysis determines the pairs of events that may influence a constraint of the memory model. Any remaining pair can be dropped from the encoding. The analysis is compatible with optimized fixpoint encodings presented in [27,28].

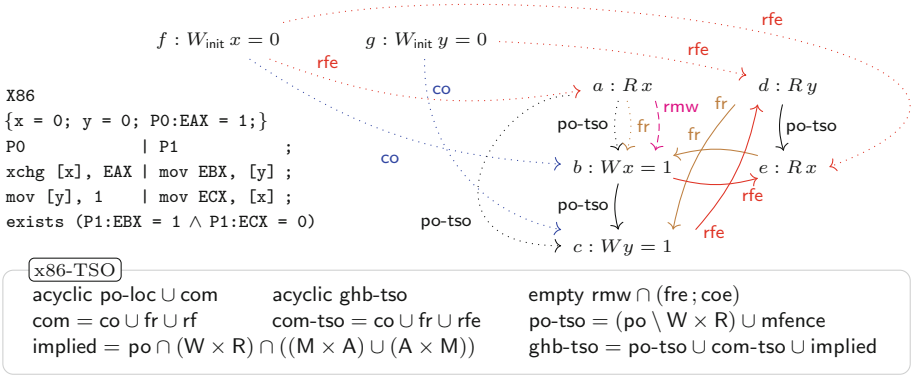
The second novelty is the support for advanced programming constructs. We redesigned Dartagnan's heap model, which now has pointers and arrays. Furthermore, we enriched the set of synchronization primitives, including read-modify-write and read-copy-update (RCU) instructions [26]. One motivation for this richer set of programming constructs is the Linux kernel memory model [15] that has recently been added to the kernel documentation [2]. This model has already been used by kernel developers to find bugs in and clarify details of the concurrency primitives. Since the model is expected to be refined with further development of the kernel, verification tools will need to quickly accommodate updates in the specification. So far, only Herd [4] has satisfied this requirement. Unfortunately, it is limited to fairly small programs (litmus tests). The present version of Dartagnan offers an alternative with substantially better performance.

We present experiments on a series of benchmarks consisting of 4751 Linux litmus tests and 7 mutual exclusion algorithms executed on TSO, ARM, and Linux. Despite the flexibility of taking memory models as inputs, Dartagnan's performance is comparable to CBMC [13] and considerably better than that of Nidhugg [5,9]. Both are model-specific tools. Compared to the previous version of Dartagnan [28] and compared to Herd [4], we gain a speed-up of more than two orders of magnitude, thanks to the relation analysis.

Related Work. In terms of the verification task to be solved, the following tools are the closest to ours. CBMC [13] is a scalable bounded model checker supporting TSO, but not ARM. An earlier version also supported Power.

Nidhugg [5,9] is a stateless model checker supporting TSO, Power, and a subset of ARMv7. It is excellent for programs with a small number of executions. RCMC [22] implements a stateless model checking algorithm targeting C11. We cannot directly benchmark against it because the source code of the tool is not yet publicly available, nor do we fully support C11. Herd [4] is the only tool aside from ours that takes a CAT memory model as input. Herd does not scale well to programs with a large number of executions, including some of the Linux kernel tests. Other verification tasks (e.g., fence insertion to restore sequential consistency) are tackled by Memorax [6–8], offence [14], Fender [23], DFence [25], and trencher [19].

Relation Analysis on an Example. Consider the program (in the .litmus format) given to the left in the figure below. The assertion asks whether there is a reachable state with final values $\texttt{EBX} = 1, \texttt{ECX} = 0$. We analyze the program under the x86-TSO memory model shown below the program. The semantics of the program under TSO is a set of executions. An execution is a graph, similar to the one given below, where the nodes are events and the edges correspond to the relations defined by the memory model. Events are instances of instructions that access the shared memory: R (loads), W (stores, including initial stores), and M (the union of both). The atomic exchange instruction `xchg [x], EAX` gives rise to a pair of read and write events related by a (dashed) rmw edge. Such reads and writes belong to the set A of atomic read-modify-write events.



The relations rf, co, and fr model the communication of instructions via the shared memory (reading from a write, coherence, overwriting a read). Their restrictions rfe, coe, and fre denote (external) communication between instructions from different threads. Relation po is the program order within the same thread and po-loc is its restriction to events addressing the same memory location. Edges of mfence relate events separated by a fence. Further relations are derived from these base relations. To belong to the TSO semantics of the program, an execution has to satisfy the constraints of the memory model: empty rmw ∩ (fre ; coe), which enforces atomicity of read-modify-write events, and the two acyclicity constraints.

Dartagnan encodes the semantics of the given program under the given memory model into an SMT formula. The problem is that each edge (a, b) that may be present in a relation r gives rise to a variable $\mathsf{r}(a, b)$. The goal of our relation analysis is to reduce the number of edges that need to be encoded. We illustrate this on the constraint acyclic ghb-tso. The graph next to the program shows the 14 (dotted and solid) edges which may contribute to the relation ghb-tso. Of those, only the 6 solid edges can occur in a cycle. The dotted edges can be dropped from the SMT encoding. Our relation analysis determines the solid edges—edges that may have an influence on a constraint of the memory model. Additionally, ghb-tso is a composition of various subrelations (e.g., po-tso or $\mathsf{co} \cup \mathsf{fr}$) that also require encoding into SMT. Relation analysis applies to subrelations as well. Applied to all constraints, it reduces the number of encoded edges for all (sub)relations from 221 to 58.

2 Input, Functionality, and Implementation

Dartagnan has the ambition of being widely applicable, from assembly over operating system code written in C/C++ to lock-free data structures. The tool accepts programs in PPC, x86, AArch64 assembly, and a subset of C11, all limited to the subsets supported by Herd's .litmus format. It also reads our own .pts format with C11-like syntax [28]. We refer to global variables as memory locations and to local variables as registers. We support pointers, i.e., a register may hold the address of a location. Addresses and values are integers, and we allow the same arithmetic operations for addresses as for regular integer values. Different synchronization mechanisms are available, including variants of read-modify-write, various fences, and RCU instructions [26].

We support the assertion language of Herd. Assertions define inequalities over the values of registers and locations. They come with quantifiers over the reachable states that should satisfy the inequalities.

We use the CAT language [11,12,16] to define memory models. A memory model consists of named relations between events that may occur in an execution. Whether or not an execution is valid is defined by constraints over these relations:

$$
\begin{aligned}
\langle MM\rangle &::= \langle const\rangle \mid \langle rel\rangle \mid \langle MM\rangle \wedge \langle MM\rangle \\
\langle const\rangle &::= acyclic(\langle r\rangle) \mid irreflexive(\langle r\rangle) \\
&\quad \mid empty(\langle r\rangle) \\
\langle rel\rangle &::= \langle name\rangle := \langle r\rangle
\end{aligned}
\qquad
\begin{aligned}
\langle r\rangle &::= \langle b\rangle \mid \langle name\rangle \mid \langle r\rangle \cup \langle r\rangle \mid \langle r\rangle \setminus \langle r\rangle \\
&\quad \mid \langle r\rangle \cap \langle r\rangle \mid \langle r\rangle^{-1} \mid \langle r\rangle^{+} \mid \langle r\rangle^{*} \mid \langle r\rangle;\langle r\rangle \\
\langle b\rangle &::= \mathsf{id} \mid \mathsf{int} \mid \mathsf{ext} \mid \mathsf{po} \mid \mathsf{fencerel}(fence) \\
&\quad \mid \mathsf{rmw} \mid \mathsf{ctrl} \mid \mathsf{data} \mid \mathsf{addr} \mid \mathsf{loc} \mid \mathsf{rf} \mid \mathsf{co}.
\end{aligned}
$$

CAT has a rich relational language, and we only show an excerpt above. So-called base relations $\langle b\rangle$ model the control flow, data flow, and synchronization constraints. The language provides intuitive operators to derive further relations. One may define relations recursively by referencing named relations. Their semantics is the least fixpoint.

DARTAGNAN is invoked with two inputs: the program, annotated with an assertion over the final states, and the memory model. There are two optional parameters related to the verification. The SMT encoding technique for recursive relations is defined by mode chosen between knastertarski (default) and idl (see below). The parameter alias, chosen between none and andersen (default), defines whether to use an alias analysis for our relation analysis (cf. Sect. 3).

Being a bounded model checker, DARTAGNAN computes an unrolled program with conditionals but no loops. It encodes this acyclic program together with the memory model into an SMT formula and passes it to the Z3 solver. The formula has the form $\psi_{prog} \wedge \psi_{assert} \wedge \psi_{mm}$, where ψ_{prog} encodes the program, ψ_{assert} the assertion, and ψ_{mm} the memory model. We elaborate on the encoding of the program and the memory model. The assertion is already given as a formula.

We model the heap by encoding a new memory location for each variable and a set of locations for each memory allocation of an array. Every location has an address encoded as an integer variable whose value is chosen by the solver. In an array, the locations are required to have consecutive addresses. Instances of instructions are modeled as events, most notably stores (to the shared memory) and loads (from the shared memory).

We encode relations by associating pairs of events with Boolean variables. Whether the pair (e_1, e_2) is contained in relation r is indicated by the variable $\mathsf{r}(e_1, e_2)$. Encoding the relations $\mathsf{r}_1 \cap \mathsf{r}_2$, $\mathsf{r}_1 \cup \mathsf{r}_2$, $\mathsf{r}_1 \ ; \ \mathsf{r}_2$, $\mathsf{r}_1 \setminus \mathsf{r}_2$ and r^{-1} is straightforward [27]. For recursively defined and (reflexive and) transitive relations, DARTAGNAN lets the user choose between two methods for computing fixed points by setting the appropriate parameter. The integer-difference logic (IDL) method encodes a Kleene iteration by means of integer variables (one for each pair of events) representing the step in which the pair was added to the relation [27]. The Knaster-Tarski encoding simply looks for a post fixpoint. We have shown in [28] that this is sufficient for reachability analysis.

3 Relation Analysis

To optimize the size of the encoding (and the solving times), we found it essential to reduce the domains of the relations. We determine for each relation a static over-approximation of the pairs of events that may be in this relation. Even more, we restrict the relation to the set of pairs that may influence a constraint of the given memory model. These restricted sets are the *relation analysis* information (of the program relative to the memory model). Technically, we compute, for each relation r, two sets of event pairs, $M(\mathsf{r})$ and $A(\mathsf{r})$. The former contains so-called *may pairs*, pairs of events that may be in relation r. This does not yet take into account whether the may pairs occur in some constraint of the memory model. The *active pairs* $A(\mathsf{r})$ incorporate this information, and hence restrict the set of may pairs. As a consequence of the relation analysis, we only introduce Boolean variables $\mathsf{r}(e_1, e_2)$ for the pairs $(e_1, e_2) \in A(\mathsf{r})$ to the SMT encoding.

The algorithm for constructing the may set and the active set is a fixpoint computation. What is unconventional is that the two sets propagate their

information in different directions. For $A(\mathsf{r})$, the computation proceeds from the constraints and propagates information down the syntax tree of the CAT memory model. The sets $M(\mathsf{r})$ are computed bottom-up the syntax tree. Interestingly, in our implementation, we do not compute the full fixpoint but let the top-down process trigger the required bottom-up computation.

Both sets are computed as least solutions to a common system of inequalities. As we work over powerset lattices (relations are sets after all), the order of the system will be inclusion. We understand each set $M(\mathsf{r})$ and $A(\mathsf{r})$ as a variable, thereby identifying it with its least solution. To begin with, we give the definition for $A(\mathsf{r})$. In the base case, we have a relation r that occurs in a constraint of the memory model. The inequality is defined based on the shape of the constraint:

$$A(\mathsf{r}) \supseteq M(\mathsf{r})\ (\textit{empty}) \qquad A(\mathsf{r}) \supseteq M(\mathsf{r}) \cap \mathsf{id}\ (\textit{irrefl.}) \qquad A(\mathsf{r}) \supseteq M(\mathsf{r}) \cap M(\mathsf{r}^+)^{-1}\ (\textit{acyclic}).$$

For the emptiness constraint, all pairs of events that may be contained in the relation are relevant. If the constraint requires irreflexivity, what matters are the pairs (e, e). If the constraint requires acyclicity, we concentrate on the pairs (e_1, e_2), where (e_1, e_2) may be in relation r and (e_2, e_1) may be in relation r^+. Note how the definition of active pairs triggers the computation of may pairs.

If the relation in the constraint is a composed one, the following inequalities propagate the information about the active pairs down the syntax tree of the CAT memory model:

$$\begin{array}{lll}
A(\mathsf{r}_1) \supseteq A(\mathsf{r})^{-1} & & \text{if } \mathsf{r} = \mathsf{r}_1^{-1} \\
A(\mathsf{r}_1) \supseteq A(\mathsf{r}) & & \text{if } \mathsf{r} = \mathsf{r}_1 \cap \mathsf{r}_2 \text{ or } \mathsf{r} = \mathsf{r}_1 \setminus \mathsf{r}_2 \\
A(\mathsf{r}_1) \supseteq A(\mathsf{r}) \cap M(\mathsf{r}_1) & & \text{if } \mathsf{r} = \mathsf{r}_1 \cup \mathsf{r}_2 \text{ or } \mathsf{r} = \mathsf{r}_2 \setminus \mathsf{r}_1 \\
A(\mathsf{r}_1) \supseteq \{x \in M(\mathsf{r}_1) \mid x; M(\mathsf{r}_2) \cap A(\mathsf{r}) \neq \emptyset\} & & \text{if } \mathsf{r} = \mathsf{r}_1; \mathsf{r}_2 \\
A(\mathsf{r}_1) \supseteq \{x \in M(\mathsf{r}_1) \mid M(\mathsf{r}_1^*); x; M(\mathsf{r}_1^*) \cap A(\mathsf{r}) \neq \emptyset\} & & \text{if } \mathsf{r} = \mathsf{r}_1^+ \text{ or } \mathsf{r} = \mathsf{r}_1^*.
\end{array}$$

The definition maintains the invariant $A(\mathsf{r}) \subseteq M(\mathsf{r})$. If a pair (e_1, e_2) is relevant to relation $\mathsf{r} = \mathsf{r}_1^{-1}$, then (e_2, e_1) will be relevant to r_1. We do not have to intersect $A(\mathsf{r})^{-1}$ with $M(\mathsf{r})^{-1}$ because $A(\mathsf{r}) \subseteq M(\mathsf{r})$ ensures $A(\mathsf{r})^{-1} \subseteq M(\mathsf{r})^{-1}$. We can avoid the intersection with the may pairs for the next case as well. There, $A(\mathsf{r}) \subseteq M(\mathsf{r})$ holds by the invariant and $M(\mathsf{r}) = M(\mathsf{r}_1) \cap M(\mathsf{r}_2)$ by definition (see below). For union and the other case of subtraction, the intersection with $M(\mathsf{r}_1)$ is necessary. There are symmetric definitions for union and intersection for r_2. For a relation r_1 that occurs in a relational composition $\mathsf{r} = \mathsf{r}_1; \mathsf{r}_2$, the pairs (e_1, e_3) become relevant if they may be composed with a pair (e_3, e_2) in r_2 to obtain a pair (e_1, e_2) relevant to r. Note that for r_2 we again need the may pairs. The definition for r_2 is similar. The definition for the (reflexive and) transitive closure follows the ideas for relational composition.

The definition of the may sets follows the syntax of the CAT memory model bottom-up. With $\oplus \in \{\cup, \cap, ;\}$ and $\otimes \in \{+, *, -1\}$, we have:

$$M(\mathsf{r}_1 \oplus \mathsf{r}_2) \supseteq M(\mathsf{r}_1) \oplus M(\mathsf{r}_2) \qquad M(\mathsf{r}^{\otimes}) \supseteq M(r)^{\otimes} \qquad M(\mathsf{r}_1 \setminus \mathsf{r}_2) \supseteq M(\mathsf{r}_1).$$

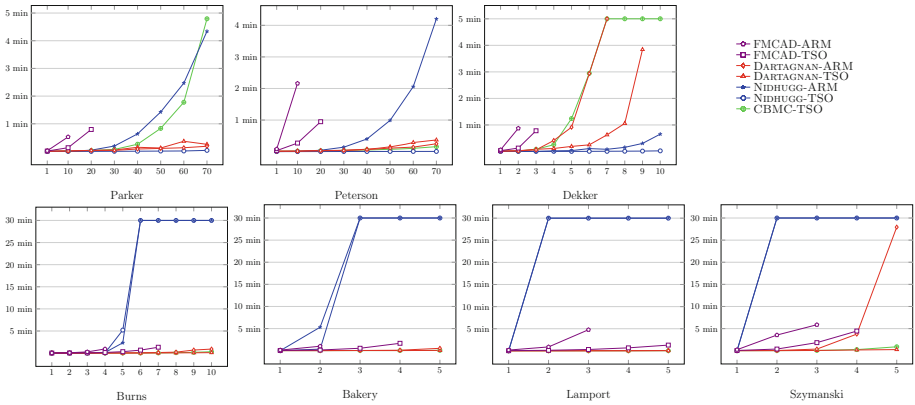


Fig. 1. Impact of the unrolling bound (x-axis) on the verification time (y-axis).

This simply executes the operator of the relation on the corresponding may sets. Subtraction ($\mathsf{r}_1 \setminus \mathsf{r}_2$) is the exception, it is not sound to over-approximate r_2.

At the bottom level, the may sets are determined by the base relations. They depend on the shape of the relations and the positions of the events in the control flow. The relations loc, co and rf are concerned with memory accesses. What makes it difficult to approximate these relations is our support for pointers and pointer arithmetic. Without further information, we have to conservatively assume that a memory event may access any address. To improve the precision of the may sets for loc, co, and rf, our fixpoint computation incorporates a *may-alias analysis*. We use a control-flow insensitive Andersen-style analysis [17]. It incurs only a small overhead and produces a close over-approximation of the may sets. The analysis returns[1] a set of pairs of memory events $PTS \subseteq (\mathbb{W} \cup \mathbb{R}) \times (\mathbb{W} \cup \mathbb{R})$ such that every pair of events outside PTS definitely accesses different addresses. Here, $\mathbb{W}$ are the store events in the program and $\mathbb{R}$ are the loads. Note that the analysis has to be control-flow insensitive as the given memory model may be very weak [10]. We have $M(\mathsf{loc}) \supseteq PTS$. Similarly, $M(\mathsf{co})$ and $M(\mathsf{rf})$ are defined by PTS restricted to $(\mathbb{W} \times \mathbb{W})$ and $(\mathbb{W} \times \mathbb{R})$, respectively.

We stress the importance of the alias analysis for our relation analysis: loc, co, and rf are frequently used as building blocks of composite relations. Excessive may sets will therefore negatively affect the over-approximations of virtually all relations in a memory model, and keep the overall encoding unnecessarily large.

Illustration. We illustrate the relation analysis on the example from the introduction. Consider constraint acyclic ghb-tso. The computation of the active set for the relation ghb-tso triggers the calculation of the may set, following the inequality $A(\mathsf{ghb\text{-}tso}) \supseteq M(\mathsf{ghb\text{-}tso}) \cap M(\mathsf{ghb\text{-}tso}^+)^{-1}$. The may set is the union of the may sets for the subrelations, shown by colored (dotted and solid) edges.

[1] This is a simplification, Andersen returns points-to sets, and we check by an intersection $PTS(r_1) \cap PTS(r_2)$ whether two registers may alias.

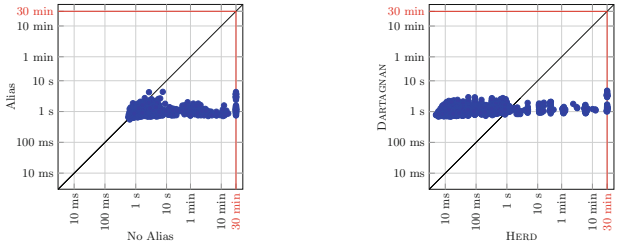


Fig. 2. Execution times (logarithmic scale) on Linux kernel litmus tests: impact of alias analysis (left) and comparison against Herd (right).

The intersection yields the edges that may lie on cycles of ghb-tso. They are drawn in solid. These solid edges in $A(\text{ghb-tso})$ are propagated down to the subrelations. For example, $A(\text{po-tso}) \supseteq A(\text{ghb-tso}) \cap M(\text{po-tso})$ yields the solid black edges.

4 Experiments

We compare Dartagnan to CBMC [13] and Nidhugg [5,9], both model-specific tools, and to Herd [4,16] and the Dartagnan FMCAD-18 version [3,28] (without relation analysis), both taking CAT models as inputs. We also evaluate the impact of the alias analysis on the execution time.

Benchmarks. For CBMC, Nidhugg, and the FMCAD-18 Dartagnan, we evaluate the performance on 7 mutual exclusion benchmarks executed on TSO (all tools) and a subset of ARMv7 (only Nidhugg and Dartagnan). The results on Power are similar to those on ARM and thus omitted. We excluded Herd from this experiment since it did not scale even for small unrolling bounds [28]. We set a 5 min timeout for Parker, Dekker, and Peterson as this is sufficient to show the trends in the runtimes, and a 30 min timeout for the remaining benchmarks. To compare against Herd, and to evaluate the impact of the alias analysis, we run 4751 Linux kernel litmus tests (all tests from [1] without Linux spinlocks). The tests contain kernel primitives, such as RCU, on the Linux kernel model. We set a 30 min timeout.

Evaluation. The times for CBMC, Nidhugg-ARM, and the FMCAD-2018 version of Dartagnan grow exponentially for Parker (see Fig. 1). The growth in CBMC and FMCAD-2018 is due to the explosion of the encoding. For the latter, the solver runs out of memory with unrolling bounds 20 (TSO) and 10 (ARM). For Nidhugg-ARM, the tool explores many unnecessary executions. The verification times for Nidhugg-TSO and the current version of Dartagnan grow linearly. The latter is due to the relation analysis. For Peterson, the results are similar except for CBMC, which matches Dartagnan's performance.

For Dekker, Nidhugg outperforms both CBMC and Dartagnan. This is because the number of executions grows slowly compared to the explosion of the

number of instructions. The executions in both memory models coincide, making the performance on ARM comparable to that on TSO for Nidhugg. The difference is due to the optimal exploration in TSO, but not in ARM. Relation analysis has some impact on the performance (see FMCAD-2018 vs. Dartagnan), but the encoding size still grows faster than the number of executions.

The benchmarks Burns, Bakery, and Lamport demonstrate the opposite trend: the number of executions grows much faster than the size of the encoding. Here, CBMC and Dartagnan outperform Nidhugg. Notice that for Burns, Nidhugg performs better on ARM than on TSO with unrolling bound 5. This is counter-intuitive since one expects more executions on ARM. Although the number of executions coincide, the exploration time is higher on TSO due to a different search algorithm. For Szymanski, similar results hold except for Dartagnan-ARM where the encoding grows exponentially.

Figure 2 (left) shows the verification times for the current version of Dartagnan with and without alias analysis. The alias analysis results in a speed-up of more than two orders of magnitude in benchmarks with several threads accessing up to 18 locations. Figure 2 (right) compares the performance of Dartagnan against Herd. We used the Knaster-Tarski encoding and alias analysis since they yield the best performance. Herd outperforms Dartagnan on small test instances (less than 1 s execution time). This is due to the JVM startup time and the preprocessing costs of Dartagnan. However, on large benchmarks, Herd times out while Dartagnan takes less than 10 s.

References

1. Linux kernel litmus test suite. https://github.com/paulmckrcu/litmus
2. Linux Memory Model. https://github.com/torvalds/linux/tree/master/tools/memory-model
3. The Dat3M tool suite. https://github.com/hernanponcedeleon/Dat3M
4. The herdtools7 tool suite. https://github.com/herd/herdtools7
5. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 353–367. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_28
6. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Leonardsson, C., Rezine, A.: Automatic fence insertion in integer programs via predicate abstraction. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 164–180. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33125-1_13
7. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Leonardsson, C., Rezine, A.: Counter-example guided fence insertion under TSO. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 204–219. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_15
8. Abdulla, P.A., Atig, M.F., Chen, Y.-F., Leonardsson, C., Rezine, A.: Memorax, a precise and sound tool for automatic fence insertion under TSO. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 530–536. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_37

9. Abdulla, P.A., Atig, M.F., Jonsson, B., Leonardsson, C.: Stateless model checking for POWER. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 134–156. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_8
10. Alglave, J., Kroening, D., Lugton, J., Nimal, V., Tautschnig, M.: Soundness of data flow analyses for weak memory models. In: Yang, H. (ed.) APLAS 2011. LNCS, vol. 7078, pp. 272–288. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25318-8_21
11. Alglave, Jade: A Shared Memory Poetics. Thèse de doctorat, L'université Paris Denis Diderot (2010)
12. Alglave, J., Cousot, P., Maranget, L.: Syntax and semantics of the weak consistency model specification language CAT. CoRR, arXiv:1608.07531 (2016)
13. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 141–157. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_9
14. Alglave, J., Maranget, L.: Stability in weak memory models. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 50–66. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_6
15. Alglave, J., Maranget, L., McKenney, P.E., Parri, A., Stern, A.S.: Frightening small children and disconcerting grown-ups: Concurrency in the linux kernel. In: ASPLOS, pp. 405–418. ACM (2018)
16. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst **36**(2), 7:1–7:74 (2014)
17. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, University of Copenhagen (1994)
18. Batty, M., Donaldson, A.F., Wickerson, J.: Overhauling SC atomics in C11 and OpenCL. In: POPL, pp. 634–648. ACM (2016)
19. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 533–553. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_29
20. Chong, N., Sorensen, T., Wickerson, J.: The semantics of transactions and weak memory in x86, Power, ARM, and C++. In: PLDI, pp. 211–225. ACM (2018)
21. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Form. Methods Syst. Des. **19**(1), 7–34 (2001)
22. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for C/C++ concurrency. PACMPL **2**(POPL), 17:1–7:32 (2018)
23. Kuperstein, M., Vechev, M.T., Yahav, E.: Automatic inference of memory fences. SIGACT News **43**(2), 108–123 (2012)
24. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.-H., Kil, Dreyer, D.: Repairing sequential consistency in C/C++11. In: PLDI, pp. 618–632. ACM (2017)
25. Liu, F., Nedev, N., Prisadnikov, N., Vechev, M.T., Yahav, E.: Dynamic synthesis for relaxed memory models. In: PLDI, pp. 429–440. ACM (2012)
26. McKenney, P.E., Slingwine, J.: Read-copy update: Using execution history to solve concurrency problems. In: Parallel and Distributed Computing and Systems, pp 509–518 (1998)
27. Ponce-de-León, H., Furbach, F., Heljanko, K., Meyer, R.: Portability analysis for weak memory models PORTHOS: **One** *Tool* **for all** *Models*. In: Ranzato, F. (ed.) SAS 2017. LNCS, vol. 10422, pp. 299–320. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66706-5_15

28. Ponce de León, H., Furbach, F., Heljanko, K., Meyer, R.: BMC with memory models as modules. In: FMCAD, pp. 1–9. IEEE (2018)
29. Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., Sewell, P.: Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. PACMPL **2**(POPL), 19:1–19:29 (2018)