Multi-Core LTSmin: Marrying Modularity and Scalability

Alfons Laarman, Jaco van de Pol, and Michael Weber

Formal Methods and Tools, University of Twente, The Netherlands
{a.w.laarman,vdpol,michaelw}@cs.utwente.nl

Abstract. The LTSmin toolset provides multiple generation and on-the-fly analysis algorithms for large graphs (*state spaces*), typically generated from concise behavioral specifications (*models*) of systems. LTSmin supports a variety of input languages, but its key feature is modularity: language frontends, optimization layers, and algorithmic backends are completely decoupled, without sacrificing performance. To complement our existing symbolic and distributed model checking algorithms, we added a multi-core backend for checking safety properties, with several new features to improve efficiency and memory usage: low-overhead load balancing, incremental hashing and scalable state compression.

1 LTSmin in a Nutshell

The LTSmin[1] toolset serves as a testbed for our research in the design of model checking tools which sacrifice neither modularity and composability nor performance. Previously, we described general features of LTSmin [4]: its wide support for input languages through reuse of existing implementations (mCRL, NipsVM, DVE, Maple and GNA, ETF), which can be combined with algorithms for checking safety properties: *enumerative*, *distributed* and BDD-based *symbolic* reachability analysis, several language-independent on-the-fly optimizations (*local transition caching*, *regrouping*) [2], as well as off-line state-space minimization algorithms.

The unifying concept in LTSmin is an **I**nterface based on a **P**artitioned **N**ext-**S**tate function. PINS connects language frontends, on-the-fly optimizations, and algorithmic backends. In Sec. 2, we describe how our new multi-core (MC) backend utilizes PINS for parallel shared-memory reachability [6] for all supported thread-safe language frontends (DVE, NipsVM, ETF).

Our MC backend provides several new contributions in the area of high-performance model checking: multi-core load balancing (Sec. 2.1), *incremental hashing* (Sec. 3), and scalable *state compression* (Sec. 4). The latter reduces memory requirements drastically, but can also improve running time of the MC tool. This is remarkable, as compression techniques generally trade space off for computational overhead.

[1] `http://fmt.cs.utwente.nl/tools/ltsmin/`, current version: 1.6, open-source.

2 LTSmin Multi-Core Architecture

PINS carefully exposes just enough structure of the models to enable high-performance algorithms and optimizations, while remaining abstract to the specific modeling language. For the purpose of this exposition, we limit the PINS description to the state and transition representation, their dependency matrices, and the next-state function. Further details can be found elsewhere [2].

In LTSmin, states are generally represented by fixed-length vectors of N *slots*: $\langle s_1, \ldots, s_N \rangle \in S$. The transition relation $\rightarrow \subseteq S \times S$ is partitioned disjunctively into K *transition groups* $(\rightarrow_1, \ldots, \rightarrow_K), \rightarrow_i \subseteq \rightarrow$. Language modules provide these subrelations. We exploit that often, a transition group depends not on the full state vector, but only on a small subset of all slots, which can be statically approximated. Hence, a $K \times N$ binary *dependency matrix* records which slots are needed per group (*read matrix*, D^R), and another records which slots are modified (*write matrix*, D^W). A value $D^R_{i,j} = 0$ indicates that all transitions in group $\rightarrow_i$ are independent of slot j, hence its value s_j can be arbitrary. A value $D^W_{i,j} = 0$ indicates that slot j will not be modified by any transition in group $\rightarrow_i$. The dependency matrices are utilized by our multi-core tool via incremental hashing and state compression.

Our multi-core backend is implemented using the pthreads library. The same reachability algorithm [6] is started in multiple threads (*workers*) that share a state storage holding the set of states already visited by the search algorithm (*closed set*). The main operation is the FindOrPut(s) function, which (atomically) reports if state s is already present in the set and otherwise inserts it. We have shown that this architecture is at least as efficient as a widely used approach based on static (hash-based) partitioning [6], despite being simpler.

2.1 Multi-Core Load Balancing

To provide all processors with some initial work, *static load balancing* (SLB) can be used. E.g., we could (sequentially) explore a sufficiently large prefix of the state space, and partition it over all workers. In parallel, each worker then explores all states reachable from its initial partition until no unvisited states are left. This simple scheme is surprisingly effective for many models, but predictably, for some inputs it leads to bad work distribution, or starvation of workers. Therefore, we tailored a *synchronous random polling* (SRP) load balancing algorithm [9] to our multi-core setting by using atomic reads and writes on shared data.

The number of explored transitions are used as measure for the work load, since it gives a close estimation of the number of actual computations (or rather, memory accesses) performed by a worker. Our measurements show that SRP provides almost perfect work distribution (less than 1% deviation from average) with negligible overhead compared to SLB. Together with shared state storage we obtain linear scalability for the LTSmin multi-core backend, which currently outperforms both SPIN [5] and DiVinE [1] on the BEEM benchmark set [8].

2.2 Example Use Cases

LTSmin tool names are composed of a prefix for the language frontend and a suffix for the algorithmic backend: `<language><algorithm>`. For example, the ETF frontend in combination with the multi-core backend is named `etf2lts-mc`.

Multi-Core Reachability Analysis using ETF can be launched with:

```
etf2lts-mc --threads=4 -s22 --lb=srp leader-7-14.etf
```

The command performs multi-core reachability with four workers (`--threads=4`) and the SRP load balancer (`--lb=srp`, default as described in `--help`). The hash table size is fixed to 2^{22} states (`-s`). This parameter needs to be chosen carefully to fit the model size or the available memory of the machine, because of our hash table design decisions [6]. Slow language frontends like NipsVM and mCRL can optionally enable transition caching (`-c`) to speed up state generation. Caching is implemented efficiently using the dependency matrix [2].

The following command searches for deadlocks (`-d`):

```
etf2lts-mc -s22 --strategy=bfs -d --trace=trace.gcf leader-7-14.etf
```

A parallel (pseudo) breadth-first search (`bfs`) generally finds a short counter example, which is stored in file `trace.gcf` and can be analyzed in detail, for example by conversion into *comma-separated value* format (only recording differences between subsequent state vectors), and loading into a spreadsheet:

```
ltsmin-tracepp --diff trace.gcf trace.csv
```

3 Incremental State Hashing

Hash tables are a common implementation choice to represent the closed set of a search. Hence, the previously mentioned FindOrPut(s) operation calculates the hash value of a given state s. For large state vectors and small *transition delays* (the time needed to calculate the effects of a transition on a state), hash calculations can easily take up to 50% of the overall run time (e.g., for C-compiled DVE2 models), even when using optimized hash functions. Given the observation that for most transitions $s \rightarrow s'$, the difference between s and s' are small (often in the order of 1–4 slots), *incremental hashing* has been investigated [7]. We have added an alternative scheme to LTSmin, which is based on Zobrist hashing [10] commonly used in games like computer chess. We believe this is the first time that Zobrist's approach has been used in the context of model checking.

Zobrist hashing incrementally composes a hash value from a matrix Z of random numbers. Each random number is bound to a fixed configuration of the game, for example, pawn at $\mathsf{H3}$. When the numbers are combined using the XOR ($\oplus$) operation, the hash value can be updated incrementally between different game configurations. For example, if a pawn P moves from $\mathsf{H3}$ to $\mathsf{H4}$, we manipulate the hash value h as follows: $h' := (h \oplus Z[\mathsf{P}][\mathsf{H3}]) \oplus Z[\mathsf{P}][\mathsf{H4}]$. Algebraic properties of $\oplus$ guarantee that a hash is unique for a configuration, independently of the path through which the configuration was reached.

The number of possible configurations of our models (slot values) is usually not known up front or too large to generate random numbers for. Therefore, we only generate a fixed amount of L numbers per state slot and map each slot value to one of them using the *modulo* operation (the Z matrix is of size $L \times N$).

Alg. 1 shows how PINS can be used to update only those slots of a state s', which (potentially) changed with respect to its predecessor s. Based on initial experimenting, we concluded that $L = 2^6$ is sufficient to yield a hash distribution at least as good as standard hash functions.[2] The size of the Zobrist matrix Z is insignificant ($4L \times N$ bytes).

Input : transition $s \rightarrow_i s'$
Input : hash value h of s
Output: hash value h' of s'
$s = \langle s_1, \ldots, s_N \rangle$
$s' = \langle s'_1, \ldots, s'_N \rangle$
$h' \leftarrow h$
for $j \in \{j \mid D^W_{i,j} = 1\}$ **do**
  $h' \leftarrow h' \oplus Z[j][s_j \textbf{ mod } L]$
  $h' \leftarrow h' \oplus Z[j][s'_j \textbf{ mod } L]$

Algorithm 1: Calculating a hash h' for successor s' of state s with hash h, using Zobrist and PINS.

The following command launches a multi-core state space exploration (reachability) with the DVE2 frontend using Zobrist hashing with $L = 2^6$ (option `-z6`), and a hash table of size 2^{18} (option `-s18`):

```
dve22lts-mc -s18 -z6 firewire_tree.4.dve
```

While the availability of large amounts of RAM in recent years shifted the "model checking bottleneck" towards processing time (we would run out of patience before running out of memory), with our improved multi-core algorithms we can easily surpass 10 million states/sec with 16 cores, sometimes claiming memory at a rate of 1 GB/sec. This causes memory to be the bottleneck again.

4 Multi-Core State Compression

We improve the memory efficiency of our tools by introducing a multi-core version of *tree compression* [3]. The following command uses it:

```
dve22lts-mc --state=tree --threads=16 firewire_tree.5.dve
```

Compared to a hash table (`--state=table`, default), memory usage for the closed set drops from 14 GB to 96 MB, while the run-time decreases as well, from 5.4 sec to 3.3 sec! The model, `firewire_tree.5.dve`, is an extreme case because of its long state vectors of 443 integers. In Sec. 5, we show that tree compression also performs well for 250 other models from the BEEM database.

The tree structure used for compression is a binary tree of *indexed sets* which map pairs of integers to indices, starting at the fringe of the tree with the slots of a state vector [3]. To provide the necessary stable indexing efficiently, we inject all indexed sets I_k into a single table [6] by appending the set number k to the lookup key. In addition to our earlier work, the tree structure is now updated incrementally using the PINS dependency matrix.

[2] Results available at: `http://fmt.ewi.utwente.nl/tools/ltsmin/nfm-2011/`

Reducing Open Set Memory. In the above case of `firewire_tree.5.dve`, the *open set* becomes the new memory hot-spot, using 200 MB. Hence, we can also opt to only store (32-bit) references to state vectors in the open set, at the expense of extra lookup operations:

```
dve2lts-mc --state=tree --threads=16 --ref firewire_tree.5.dve
```

This reduces the memory footprint of the open set from 200 MB to about 250 KB. Alternatively, depth-first search could be used, which often succeeds with a smaller open set than BFS:

```
dve2lts-mc --state=tree --strategy=dfs firewire_tree.5.dve
```

5 Experiments

We performed benchmarks on a 16-core AMD Opteron 8356 with 64 GB RAM. All models of the BEEM database [8] were used with command lines illustrated in the previous section. The hash table size was fixed for all tools to avoid resizing.

Tab. 1 shows an example of the effects of tree compression, Zobrist and references on the run-time and the memory usage of the different algorithms. The memory totals represent the space occupied by states on the open set and closed set (tree or hash table). Zobrist is not implemented for the tree structure.

Table 1. All possible combinations of the use cases for model `firewire_link.5`.

	Cores:		1			16		
	Options:		none	`--ref`	`-z6`	none	`--ref`	`-z6`
Total time [sec]	bfs	table	5.4	5.7	4.7	0.3	0.3	0.3
		tree	4.8	4.4	–	0.2	0.2	–
	dfs	table	5.7	5.7	4.8	0.4	0.4	0.3
		tree	4.1	4.4	–	0.2	0.2	–
Total mem. [GB]	bfs	table	12.6	12.5	12.6	12.6	12.5	12.6
		tree	0.9	0.7	–	0.9	0.7	–
	dfs	table	12.5	12.5	12.5	12.5	12.5	12.5
		tree	0.7	0.7	–	0.7	0.7	–

Analysis revealed that the compression factors of tree compression and SPIN's COLLAPSE are primarily (linearly) dependent on the state length [3]. Fig. 1 shows *absolute* compression factors as values for all BEEM models that fitted into memory (250 out of 300). We established a maximum line for both compression techniques. On average, tree compression is about four times as effective as COLLAPSE.

Fig. 2 compares the performance of our MC backend with other tools. We translated BEEM models to PROMELA for SPIN; only those 100 models with similar state counts were used (less than 20% difference). Despite slower sequential performance due to the (larger) PINS state format, LTSMIN ultimately scales better than DiVinE and SPIN. Tree compression results in only 20% run-time overhead (aggregated) compared to the fastest hash table-based method.

Future Work. In the lab, we have working versions of LTSMIN that support full LTL model checking, partial-order reduction and multi-core *swarmed* LTL. All of these features are implemented as additional PINS layers and search strategies, building on the current infrastructure.

Conclusions. Several use cases and experiments show how LTSMIN can be applied to solve verification problems. Multi-core runs with Zobrist hashing can solve problems quickly provided that enough memory is available, while tree compression and state references can solve problems with large state vectors or on machines with little memory.

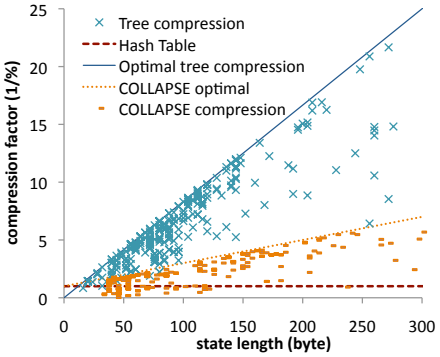

Fig. 1. Tree/COLLAPSE compression for 250 models

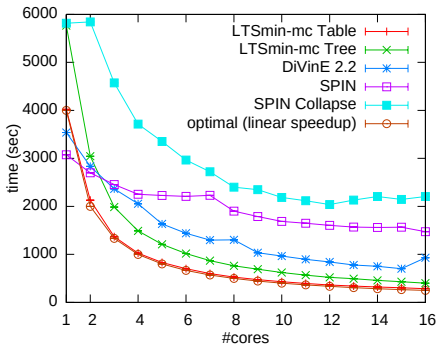

Fig. 2. Aggregate run-times for all tools

References

1. Barnat, J., Ročkai, P.: Shared hash tables in parallel model checking. Elec. Notes in Theor. Comp. Sc. 198(1), 79 – 91 (2008), proc. of the 6th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2007)
2. Blom, S., van de Pol, J., Weber, M.: Bridging the gap between enumerative and symbolic model checkers. Tech. Rep. TR-CTIT-09-30, Centre for Telematics and Information Technology, University of Twente, Enschede (2009)
3. Blom, S., Lisser, B., van de, J.P., Weber, M.: A database approach to distributed state space generation. In: Sixth Intl. Workshop on Par. and Distr. Methods in verifiCation, PDMC. pp. 17–32. CTIT, Enschede (July 2007)
4. Blom, S., van de Pol, J., Weber, M.: LTSmin: Distributed and symbolic reachability. In: Touili, T., Cook, B., Jackson, P. (eds.) Computer Aided Verification, Lecture Notes in Computer Science, vol. 6174, pp. 354–359. Springer Berlin / Heidelberg (2010), 10.1007/978-3-642-14295-6_31
5. Holzmann, G.J., Bošnacki, D.: The design of a multicore extension of the SPIN model checker. IEEE Trans. Softw. Eng. 33(10), 659–674 (2007)
6. Laarman, A.W., van de Pol, J.C., Weber, M.: Boosting multi-core reachability performance with shared hash tables. In: Sharygina, N., Bloem, R. (eds.) Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design, Lugano, Swiss. IEEE Computer Society, USA (October 2010)
7. Nguyen, V.Y., Ruys, T.C.: Incremental hashing for SPIN. In: Havelund, K., Majumdar, R., Palsberg, J. (eds.) SPIN. Lecture Notes in Computer Science, vol. 5156, pp. 232–249. Springer (2008)
8. Pelánek, R.: BEEM: Benchmarks for explicit model checkers. In: Proc. of SPIN Workshop. LNCS, vol. 4595, pp. 263–267. Springer (2007)
9. Sanders, P.: Load Balancing Algorithms for Parallel Depth First Search. Ph.D. thesis, University of Karlsruhe (1997)
10. Zobrist, A.L.: A new hashing method with application for game playing. Tech. Rep. 88, Computer Sciences Department, University of Wisconsin (1969)