KRATOS – A Software Model Checker for SystemC

A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri

Fondazione Bruno Kessler — Irst
{cimatti, griggio, amicheli, narasamdya, roveri}@fbk.eu

Abstract. The growing popularity of SystemC has attracted research aimed at the formal verification of SystemC designs. In this paper we present KRATOS, a software model checker for SystemC. KRATOS verifies safety properties, in the form of program assertions, by allowing users to explore two directions in the verification. First, by relying on the translation from SystemC designs to sequential C programs, KRATOS is capable of model checking the resulting C programs using the symbolic lazy predicate abstraction technique. Second, KRATOS implements a novel algorithm, called ESST, that combines *Explicit* state techniques to deal with the SystemC *Scheduler*, with *Symbolic* techniques to deal with the *Threads*. KRATOS is built on top of NUSMV and MATHSAT, and uses state-of-the-art SMT-based techniques for program abstractions and refinements.

1 introduction

Formal verification of SystemC has recently gained significant interests [19, 13, 18, 23, 16, 22, 10]. Despite its importance, verification of SystemC designs is hard and challenging. A SystemC design is a complex entity comprising a multi-threaded program where scheduling is cooperative, according to a specific set of rules [21], and the execution of threads is mutually exclusive.

In this paper we present KRATOS, a new software model checker for SystemC. KRATOS provides two different analyses for verifying safety properties (in the form of program assertions) of SystemC designs. First, KRATOS implements a sequential analysis based on lazy predicate abstraction [15] for verifying sequential C programs. To verify SystemC designs using this analysis, we rely on the translation from SystemC to a sequential C program, such that the resulting C program contains both the mapping of SystemC threads in the form of C functions and the encoding of the SystemC scheduler. Second, KRATOS implements a novel concurrent analysis, called ESST [10], that combines *Explicit* state techniques to deal with the SystemC *Scheduler*, with *Symbolic* techniques, based on lazy predicate abstraction, to deal with the *Threads*.

KRATOS is available for download at `https://es.fbk.eu/tools/kratos`. In appendix A we include, only for the review process, the results of an experimental evaluation comparing KRATOS with other model checkers on some different benchmarks.

2 Verification Flow

The flow of SystemC verification using KRATOS is shown in Figure 1. The flow consists of two directions. The first direction relies on the translation of a SystemC design into a sequential C program, such that the C program contains a function modelling each of the SystemC threads and the encoding of the SystemC scheduler.

KRATOS implements a sequential analysis that model checks sequential C programs. This sequential analysis is essentially lazy predicate abstraction [15], which is based on the construction of an abstract reachability tree (ART) by unwinding the control-flow automaton (CFA) of the C program. The ART itself represents the reachable abstract state space. The sequential analysis that KRATOS implements is not restricted to the results of SystemC translations, but it can also handle general sequential C programs.

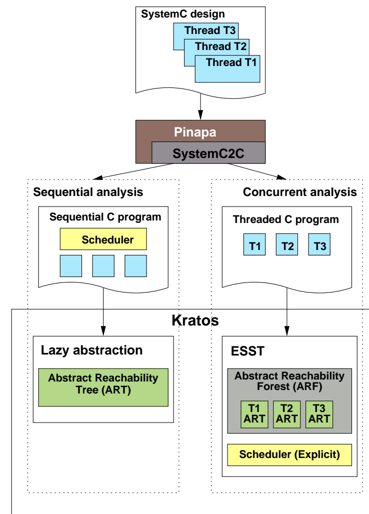

Fig. 1. The SystemC verification flow.

The second direction uses the concurrent analysis, which is the ESST algorithm, to model check SystemC designs. Similar to the first direction, in the second direction the SystemC design is translated into a so-called threaded C program that contains a function for modelling each SystemC thread. But, unlike the sequential C program above, the encoding of the SystemC scheduler is no longer part of the threaded C program. The SystemC scheduler itself is now part of the ESST algorithm and its states are tracked explicitly. ESST is based on the construction of an abstract reachability forest (ARF) where each tree in the forest is an ART of the running thread. The ESST algorithm is described in detail in [10], while its extension to consider partial order reduction is described in detail in [11].

Translations from SystemC designs into sequential and threaded C programs are performed by SYSTEMC2C, a new back-end of PINAPA [20].

3 Architecture

The architecture of KRATOS is shown in Figure 2. It consists of a front-end that includes a parser for C programs, a type checker, a CFA encoder, and static data-flow analyses and optimization phases.

The parser translates a textual C program into its abstract syntax tree (AST) representation. The AST is then traversed by the type checker to build a symbol table. The CFA encoder builds a CFA or a set of CFAs from the AST. Currently, the CFA encoder provides three encodings: small-block encoding (SBE), basic-block encoding (BBE), and large-block encoding (LBE). In SBE each block consists only of at most one statement. In BBE each block is a sequence of statements that can only entered at the beginning and exited at the end. In LBE, as described in [1], each block is the largest directed acyclic fragment of the CFA. LBE improves performances by reducing the number of abstract post image computation [1].

Static analyses and optimizations implemented by KRATOS include a simple cone-of-influence reduction that removes nodes of CFAs that do not lead to the error nodes, dead-code elimination, and constant propagation.

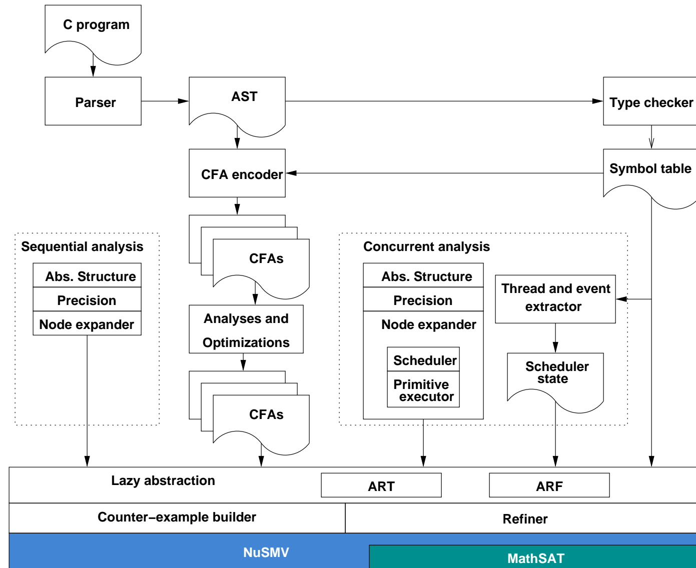

Fig. 2. The architecture of KRATOS.

The sequential analysis is a reimplementation of the same analysis performed by existing software model checkers based on the lazy predicate abstraction, like BLAST [2] or CPACHECKER [3]. The analysis consists of an abstraction structure, a precision, and a node expander. The abstraction structure contains the representation of abstract states that label nodes in ART. A state typically consists of a location (or node) in CFA and a formula denoting the abstract data state. [1] The structure also implements the coverage criteria that stop the expansion of ART nodes. The precision encodes the mapping from locations in CFA to sets of predicates that have been discovered so far. These predicates are relevant predicates used to compute the abstract post images. The node expander expands an ART node by (1) unwinding each of the outgoing edges of the CFA node in the state labelling the ART node, and (2) computing the abstract post image of the state with respect to the statement labelling the outgoing edge. The node expander currently implements depth-first search (DFS), breadth-first search (BFS), and topological ordering strategies for expanding nodes.

For the concurrent analysis, we extract the threads and events from the input threaded C program to create the initial state of scheduler. In this analysis, the node expander is also equipped with a scheduler and a primitive executor. The scheduler explores all possible schedules given a scheduler state as an input. The primitive executor executes calls to functions that modify the state of scheduler. The executor only assumes that the actual arguments of the calls are known statically. Partial order reduction is in-

[1] A state labelling an ART node can also equipped with a stack describing traces of function calls.

tegrated in the concurrent analysis by applying it to the node expander: it only explores a representative subset of all the possible schedules.

We remark that, the architecture of the concurrent analysis does not assume that the scheduler is a SystemC scheduler. In fact any implementation of a cooperative scheduler with one exclusively running thread in each schedule can be plugged into the analysis.

Depending on the analysis, either an ART or an ARF is built by the lazy abstraction component. When the analysis reaches an error location, then the counterexample builder builds a counterexample by constructing the path from the node labelled with the error location to the root of the ART, or to the root of the first ART in the ARF. If the counterexample is spurious, that is the formula representing it is unsatisfiable, then the counterexample is passed to the refiner. The refiner tries to refine the precision by discovering new predicates that need to be kept track by using the unsatisfiable core or interpolation based techniques as described in [14].

KRATOS is built on top of an extended version of NUSMV [7], which is tightly integrated with the MATHSAT SMT solver [5]. KRATOS relies on NUSMV and MATHSAT for abstraction computation, for representing the abstract state within each ART, for the coverage check, for checking the satisfiability of expressions representing counter examples, and for extracting the unsatisfiable core and for generating sequence of interpolants from counterexample paths.

4 Novel Functionalities

KRATOS offers the following novel functionalities.

ESST *algorithm.* The translation from SystemC designs to sequential C programs enables the verification of SystemC using the "off-the-shelf" software model checking techniques. However, such a verification is inefficient because the abstraction of SystemC scheduler is often too aggresive, and thus requires many refinements to reintroduce the abstracted details. The ESST algorithm attacks such an inefficiency by modelling the scheduler precisely, and, as shown in [10], outperforms the SystemC verification through sequentialization.

Partial-order reduction. Despite its relative effectiveness, ESST still has to explore a large number of thread interleavings, many of which are redundant. Such an exploration degrades the run time performance and yields high memory consumptions. POR is a well-known technique that tackles the state explosion problem by exploring only a representative subset of all possible interleavings. Recently partial-order reduction (POR) techniques have been incorporated in the ESST algorithm [11]. KRATOS currently implements POR techniques based on persistent set, sleep set, and a combination of both. (See [11] for details.)

Advanced abstraction techniques. KRATOS implements cartesian and boolean abstraction techniques that are implemented in BLAST and CPACHECKER. In addition, KRATOS also implements hybrid predicate abstraction that integrates BDDs and SMT solvers, as described in [12], and structural abstraction, as described in [8].

Translators. KRATOS is capable of translating the sequential and threaded C to the input languages of other verification engines. For example, KRATOS can translate se-

quential a C program into an SMV model. By such a translation, one can then use the model checking algorithm implemented by, for example, NUSMV [7] to verify the C program. In particular one can experiment with the bounded model checking (BMC) [4] technique of NUSMV that does not exist in KRATOS.

Under-approximation. KRATOS is also able to generate under-approximation for quick bug hunting. To this extent, KRATOS has recently been equipped with a translation from threaded C programs into PROMELA code [6]. Such a translation enables the verification by under-approximation using the SPIN model checker [17].

Transition encoding. Each block in the CFA is translated into a transition expressed by a NUSMV expression. We have observed that different encodings for transitions can affect the performance of KRATOS. In particular, the encoding for the transitions affect the performance of MATHSAT in terms of abstraction computations and also lead MATHSAT to yielding different interpolants, and thus different discovered predicates. KRATOS provides several different encodings for transitions. They differs in the number of variables needed to encode the transition of each block of the CFA, from the fact that intermediate expressions are folded or not, or whether NUSMV if-then-else expressions are used to compactly represent intermediate expressions. More details about the provided encodings can be found in the tool documentation [9]. The availability of several encodings opens for choosing the most effective one depending on the nature of the problem.

5 Conclusion and Future Work

We have presented KRATOS, a software model checker for SystemC. KRATOS provides two different analyses for verifying SystemC designs: sequential and concurrent analyses. The sequential analysis, based on the lazy predicate abstraction, verifies the C program resulting from the sequentialization of the SystemC design. The concurrent analysis, based on the novel ESST algorithm, combines explicit state techniques with lazy predicate abstraction to verify threaded C program that models a SystemC design. The experimental evaluation (see appendix A) shows that ESST algorithm, for the verification of the considered SystemC benchmarks, outperforms all the other approaches based on sequential analysis. On the considered pure sequential benchmarks, the sequential analysis shows better performance than other state-of-the-art approaches for the majority of the benchmarks.

For future work, we will extend KRATOS to handle a larger subset of C constructs like data structures, arrays and pointers (which are currently treated as uninterpreted functions) and to be able to take into account the bit-precise semantics of operations. We will investigate how to extend the ESST approach to deal with symbolic primitive functions to generalize the scheduler exploration. We would also like to combine the over-approximation analysis, based on the lazy abstraction, with an under-approximation analysis, based on PROMELA translation or BMC. Finally, we will consider to extend the ESST techniques to the verification of concurrent C programs from other application domains (e.g. robotics, railways), where different scheduling policies have to be taken into account

References

1. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: FMCAD. pp. 25–32. IEEE (2009)
2. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. STTT 9(5-6), 505–525 (2007)
3. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate Abstraction with Adjustable-Block Encoding. In: FMCAD. pp. 189–197 (2010)
4. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without bdds. In: TACAS. LNCS, vol. 1579, pp. 193–207. Springer (1999)
5. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The MathSAT 4 SMT Solver. In: CAV. LNCS, vol. 5123, pp. 299–303. Springer (2008)
6. Campana, D., Cimatti, A., Narasamdya, I., Roveri, M.: Formal Bug Finding for SystemC, submitted for publication
7. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NuSMV: A New Symbolic Model Checker. STTT 2(4), 410–425 (2000)
8. Cimatti, A., Dubrovin, J., Junttila, T., Roveri, M.: Structure-aware computation of predicate abstraction. In: FMCAD. pp. 9–16. IEEE (2009)
9. Cimatti, A., Griggio, A., Micheli, A., Narasamdya, I., Roveri, M.: KRATOS – A Software Model Checker for SystemC (2011), `http://es.fbk.eu/tools/kratos`
10. Cimatti, A., Micheli, A., Narasamdya, I., Roveri, M.: Verifying SystemC: a Software Model Checking Approach. In: FMCAD. pp. 51–59 (2010)
11. Cimatti, A., Narasamdya, I., Roveri, M.: Boosting Lazy Abstraction for SystemC with Partial Order Reduction. In: TACAS (2011), To appear
12. Cimatti, A., Franzén, A., Griggio, A., Kalyanasundaram, K., Roveri, M.: Tighter integration of BDDs and SMT for Predicate Abstraction. In: Proc. of DATE. pp. 1707–1712. IEEE (2010)
13. Große, D., Drechsler, R.: CheckSyC: an efficient property checker for RTL SystemC designs. In: ISCAS (4). pp. 4167–4170. IEEE (2005)
14. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL. pp. 232–244. ACM (2004)
15. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL. pp. 58–70 (2002)
16. Herber, P., Fellmuth, J., Glesner, S.: Model checking SystemC designs using timed automata. In: CODES+ISSS. pp. 131–136. ACM (2008)
17. Holzmann, G.J.: Software model checking with SPIN. Advances in Computers 65, 78–109 (2005)
18. Kroening, D., Sharygina, N.: Formal verification of SystemC by automatic hardware/software partitioning. In: MEMOCODE. pp. 101–110. IEEE (2005)
19. Moy, M., Maraninchi, F., Maillet-Contoz, L.: Lussy: A toolbox for the analysis of systems-on-a-chip at the transactional level. In: ACSD. pp. 26–35. IEEE (2005)
20. Moy, M., Maraninchi, F., Maillet-Contoz, L.: Pinapa: an extraction tool for SystemC descriptions of systems-on-a-chip. In: EMSOFT. pp. 317–324. ACM (2005)
21. Tabakov, D., Kamhi, G., Vardi, M.Y., Singerman, E.: A Temporal Language for SystemC. In: FMCAD. pp. 1–9. IEEE (2008)
22. Tabakov, D., Vardi, M.Y.: Monitoring Temporal SystemC Properties. In: MEMOCODE. pp. 123–132 (2010)
23. Traulsen, C., Cornet, J., Moy, M., Maraninchi, F.: A SystemC/TLM Semantics in Promela and Its Possible Applications. In: SPIN. LNCS, vol. 4595, pp. 204–222. Springer (2007)

A Experimental Evaluation

We conducted and experimental evaluation on SystemC designs and on pure sequential programs. The results are reported on Tables 1 and on Table 2, respectively. The benchmarks on Table 1 are taken and adapted from literature and SystemC distribution. They are an extended version of those used in [10, 11]. The benchmarks on Table 2 have been taken from [3]. Both family of benchmarks are also available on KRATOS web page: `http://es.fbk.eu/tools/kratos`. The column labeled with 'V' shows the statuses of the benchmarks: S for safe and U for unsafe. All the experiments were conducted on an Intel-Xeon DC 3GHz running Linux equipped with 4GB RAM. We set the time limit to 1000s and the memory limit to 2GB. Experiments denoted by T.O means they reach the time limit.

Tool configurations used for the experiments are as follows. For BLAST, we enable FOCI. For CPACHECKER, we use the LBE configuration used in [3]. For sequential analysis of KRATOS, we optimize CFA and inline functions. For the ESST algorithm, we explore threads' ARTs with depth-first search (DFS) strategy and explore ARF nodes with breadth-first search (BFS) strategy. We also inline functions and enable partial-order reductions in the ESST. Further, for ESST, we use the simple transition encoding and refine ARF in a similar way to the one discussed in [15].

Table 1 clearly shows that, as far as these benchmarks are concerned, KRATOS ESST outperforms the other four approaches in all the cases. KRATOS Seq performs better than CPACHECKER on all the benchmarks, while there are cases where SATABS performs better (see e.g. the `transmitter.5` and `token-ring.4`). This can be explained by the fact the search in the two model checkers is different.

Table 2 shows that, as far as the pure sequential benchmarks discussed in [3] are concerned, CPACHECKER outperforms SATABS on all the benchmarks, CPACHECKER outperforms BLAST on most of the cases, while we have cases where KRATOS Seq performs better than CPACHECKER, and others where it performs worse. This is explained by the fact that the search in the two model checkers, although similar may end-up exploring paths in a different order and thus discovering different sets of predicates.

Table 1. Results of experiments. The following set of options have been used for the various tools: SATABS: `--iterations 1000 --32`, BLAST: `-foci`, CPACHECKER: the LBE configuration used in [3], KRATOS seq: `--opt_cfa --inline_function=1`, KRATOS ESST: `--thread_expand=DFS --node_expand=BFS --transition_method=Simple --cex_build_method=Backward --pivot --inline_threaded_function=1 --po_reduce --po_reduce_sleep`.

Benchmark Names	V	SATABS	BLAST	CPACHECKER	KRATOS Seq	KRATOS ESST
bist-cell	S	36.600	T.O	158.200	1.690	0.490
kundu	S	139.440	T.O	T.O	T.O	1.100
kundu-bug-1	U	41.500	245.850	34.490	39.500	0.400
kundu-bug-2	U	110.550	T.O	156.840	T.O	1.100
mem-slave-tlm.1	S	77.210	T.O	T.O	T.O	2.800
mem-slave-tlm.2	S	202.540	T.O	T.O	T.O	13.490
mem-slave-tlm.3	S	452.860	T.O	T.O	T.O	152.790
mem-slave-tlm.4	S	973.960	T.O	T.O	T.O	42.390
mem-slave-tlm.5	S	T.O	T.O	T.O	T.O	735.340
pc-sfifo-1	S	4.260	46.650	25.300	4.390	0.300
pc-sfifo-2	S	5.210	300.380	40.530	23.390	0.300
pipeline	S	T.O	T.O	T.O	T.O	76.600
token-ring.1	S	16.520	97.200	42.610	4.600	0.100
token-ring.2	S	62.240	888.290	T.O	63.300	0.100
token-ring.3	S	152.360	T.O	T.O	T.O	0.190
token-ring.4	S	602.300	T.O	T.O	T.O	0.190
token-ring.5	S	T.O	T.O	T.O	T.O	0.290
token-ring.6	S	T.O	T.O	T.O	T.O	0.400
token-ring.7	S	T.O	T.O	T.O	T.O	0.500
token-ring.8	S	T.O	T.O	T.O	T.O	0.700
token-ring.9	S	T.O	T.O	T.O	T.O	1.390
token-ring.10	S	T.O	T.O	T.O	T.O	2.400
token-ring.11	S	T.O	T.O	T.O	T.O	4.500
token-ring.12	S	T.O	T.O	T.O	T.O	4.100
token-ring.13	S	T.O	T.O	T.O	T.O	7.300
toy	S	22.790	T.O	T.O	T.O	1.600
toy-bug-1	U	28.050	T.O	T.O	669.830	1.600
toy-bug-2	U	20.290	T.O	320.070	T.O	0.600
transmitter.1	U	2.230	1.270	9.050	1.200	0.010
transmitter.2	U	26.920	29.400	27.780	17.990	0.010
transmitter.3	U	61.460	501.350	115.600	25.790	0.010
transmitter.4	U	190.620	T.O	T.O	T.O	0.010
transmitter.5	U	472.180	T.O	T.O	869.410	0.010
transmitter.6	U	T.O	T.O	T.O	T.O	0.100
transmitter.7	U	T.O	T.O	T.O	T.O	0.090
transmitter.8	U	T.O	T.O	T.O	T.O	0.100
transmitter.9	U	T.O	T.O	T.O	T.O	0.100
transmitter.10	U	T.O	T.O	T.O	T.O	0.100
transmitter.11	U	T.O	T.O	T.O	T.O	0.200
transmitter.12	U	T.O	T.O	T.O	T.O	0.400
transmitter.13	U	T.O	T.O	T.O	T.O	0.200

Table 2. Results of experiments. The following set of options have been used for the various tools: SATABS: `--iterations 1000 --32`, BLAST: `-foci`, CPACHECKER: the LBE configuration used in [3], KRATOS seq: `--opt_cfa --inline_function=1`.

Benchmark Names	V	SATABS	BLAST	CPACHECKER	KRATOS Seq
cdaudio-simpl1.cil.c	S	106.980	T.O	69.150	38.390
cdaudio-simpl1-bug.cil.c	U	104.390	T.O	40.000	34.300
diskperf-simpl1.cil.c	S	25.300	199.340	59.610	22.290
floppy-simpl3.cil.c	S	120.440	95.800	35.500	10.300
floppy-simpl3-bug.cil.c	U	121.020	62.720	26.450	8.900
floppy-simpl4.cil.c	S	272.350	124.730	53.110	17.800
floppy-simpl4-bug.cil.c	U	271.510	108.030	40.150	16.100
kbfiltr-simpl1.cil.c	S	17.120	4.970	12.960	1.490
kbfiltr-simpl2.cil.c	S	56.230	8.370	19.780	3.690
kbfiltr-simpl2-bug.cil.c	U	120.760	17.390	15.300	4.300
s3-clnt-1.cil.c	S	82.780	19.520	28.670	4.890
s3-clnt-1-bug.cil.c	U	16.220	12.140	14.590	2.700
s3-clnt-2.cil.c	S	725.340	24.130	30.370	13.100
s3-clnt-2-bug.cil.c	U	17.250	12.740	10.460	14.590
s3-clnt-3.cil.c	S	355.700	22.890	18.070	10.590
s3-clnt-3-bug.cil.c	U	17.380	30.560	12.250	14.290
s3-clnt-4.cil.c	S	407.830	20.620	58.330	6.300
s3-clnt-4-bug.cil.c	U	17.310	14.270	10.930	16.990
s3-srvr-1.cil.c	S	219.490	154.930	101.170	429.450
s3-srvr-1-bug.cil.c	U	23.990	3.800	9.860	432.250
s3-srvr-2.cil.c	S	313.910	51.680	75.760	110.290
s3-srvr-2-bug.cil.c	U	22.680	3.350	9.610	3.090
s3-srvr-3.cil.c	S	210.190	53.560	31.800	108.490
s3-srvr-4.cil.c	S	321.160	48.030	37.090	96.590
s3-srvr-6.cil.c	S	T.O	T.O	218.580	32.400
s3-srvr-7.cil.c	S	T.O	246.950	124.810	35.390
s3-srvr-8.cil.c	S	T.O	140.590	215.110	136.390
s3-srvr-9.cil.c	S	T.O	278.450	41.940	116.490
s3-srvr-10.cil.c	S	T.O	92.260	31.510	311.760
s3-srvr-11.cil.c	S	914.910	115.700	106.150	47.990
s3-srvr-12.cil.c	S	892.160	153.530	63.280	639.540
s3-srvr-13.cil.c	S	929.420	296.170	56.630	657.430
s3-srvr-14.cil.c	S	T.O	193.210	70.630	45.600
s3-srvr-15.cil.c	S	T.O	225.410	72.160	127.680
s3-srvr-16.cil.c	S	T.O	965.300	51.440	75.480
test-locks-5.c	S	0.330	3.500	4.620	0.100
test-locks-6.c	S	0.670	8.600	4.560	0.090
test-locks-7.c	S	1.390	22.820	4.790	0.100
test-locks-8.c	S	2.110	55.420	4.770	0.190
test-locks-9.c	S	3.420	119.890	4.840	0.200
test-locks-10.c	S	5.400	360.950	4.850	0.200
test-locks-11.c	S	10.250	T.O	5.080	0.190
test-locks-12.c	S	15.070	T.O	5.040	0.290
test-locks-13.c	S	17.540	T.O	5.140	0.300
test-locks-14.c	S	26.580	T.O	4.880	0.390
test-locks-15.c	S	35.060	T.O	5.110	0.390