The Benefit of Concurrency in Model Checking

Baruch Sterin, Niklas Een, Alan Mishchenko, and Robert Brayton

Berkeley Verification and Synthesis Research Center (BVSRC)

Department of EECS, University of California, Berkeley

{sterin, een, alanmi, brayton}@eecs.berkeley.edu

1 Abstract

Model checking (MC) offers an application amenable to the use of concurrency in some innovative ways. In this paper we focus on solving single, hard properties, and show how concurrency can be used to orchestrate the MC engines in a more robust way compared to traditional, non-concurrent solutions. Through a Python front-end to ABC, with concurrency capabilities, we demonstrate enhanced performance on both academic and industrial problems.

2 Introduction

Model checking has seen significant advances in the last few years, both in terms of capabilities and industrial adoption. There are many commercial offerings, as well as state-of-the-art model checkers inside major systems houses such as Intel [17] and IBM [16]. In academia, several dozen model checkers have been developed over many years. Biennial or annual model checking competitions have been held at CAV; the latest one, HWMCC'10, was held in July 2010 [11]. These competitions have greatly spurred research in the MC area.

Most of the recent successful MC programs, in academia as well as industry, use multiple verification and synthesis engines. The top two entries in HWMCC'10 [11] were ABC [5][9][18] and PdTRAV [12][24], which use multiple engines, although the third place program used a single newly developed engine [4][6], which at this point has been adopted as another engine in almost all the multi-engine MC programs, commercial as well as academic.

While new and stronger engines are extremely valuable contributions, allotting time between the plethora of existing engines has become a challenging research problem in its own right. This is particularly true when also considering abstraction, speculation etc., as well as how much time and resource to allow before backing out of such under- or over-approximations of the current proof-obligations.

Most commercial offerings probably use concurrency and multi-cores in MC, but little is known outside of the companies about how concurrency is used. An exception is the paper [33] which will be discussed later in Section 9. With the wide availability of multi-cores in desktops and servers, the use of concurrency in model checking becomes increasingly compelling. We argue that even for a single core the case for concurrency is still compelling.

In this paper, we propose "hybrid" concurrent algorithm for use of concurrency in a multi-core setting and argue that, not only does it lead to speedups due to parallelism, but also to increased capability in solving hard MC problems due to the greater number of execution paths that can be explored.

In Section 3 we outline the various basic engines that are available for model checking. In Section 4, we describe a concurrent version of verification, *c_verify,* using multiple engines concurrently. Section 5 discusses several ways this can be used in model checking. Section 6 gives some details our implementation using Python and the Linux *fork* command. In Section 7, we give some experimental results on "hard" industrial examples supporting our claim and observation that concurrent model checking can be more powerful and rugged. In Section 8, we discuss issues concerning speedup, improved ruggedness and possible memory conflicts.

3 Basic MC engines

In ABC, the following MC engines are available:
1. Random simulation
2. Semi-formal simulation
3. Bounded model checking (BMC) [15]
4. BDD-based reachability [7][25]
5. Property directed reachability (PDR) [4]
6. Interpolation [14]
7. Synthesis:
    a. rewriting [10]
    b. retiming [13]
    c. sequential signal correspondence [26]
    d. phase abstraction [27]
    e. temporal decomposition [23]
8. Abstraction: [8]
    a. counterexample-based (CB) [19]
    b. proof-based (PB) [20][21]
9. Speculation [2][3]

An engine can be categorized as a (i) *verification engine*, that either finds a bug-trace or proves the property (engines 1-6), or a (ii) *transformation engine*, which attempts to re-shape or deconstruct the problem into one or more simpler problems (engines 7-9).

Verification engines can be classified further into *complete* ("proof-producing", engines 4-6) and *incomplete*

("bug-hunting only", engines 1-3).[1] Transformation engines can be either *equivalence preserving* (engines of 7) or *abstracting* (8 and 9). Once abstraction has been applied, bug-traces may be spurious and only proofs can be considered conclusive. However, such traces can be used to refine the current abstraction until the property, if true, can be proved.

In addition to the basic engines listed above, variations exist within each engine through the sometimes numerous tuning parameters. Particularly for hard problems, algorithms often are quite sensitive to small variations in these parameters, and thus it is useful to try different versions of the same engine in a proof orchestration.

This leads to the problem of algorithmically deciding when and how to employ the different engines, what time-outs to use, and what values to give to the parameters. Although there exist some rules of thumb, such as "don't use BDDs if there are more than 200 flops", it is still the case that in a sequential (non-concurrent) flow, orchestration problems become accentuated because a wrong decision can be very costly. Therefore, running several engines concurrently, even beyond the supported parallelism of the underlying hardware, can lead to speedups simply by avoiding some worst-case behaviors.

An outline of typical sequential (non-concurrent) proof-script is given below. For discussion, we will assume that there is only one output representing a single property to be proved (property checking), or representing the miter of two circuits that are supposedly equivalent (sequential equivalence checking).

Simplify phase. Usually the first step done is to simplify[2] the circuit by employing methods such as rewriting, forward- and minimum- register retiming [13], phase abstraction [27], reparametrization [22], temporal decomposition [23], and signal correspondence (with and without using constraints) [26]. The order in which these are deployed can lead to different outcomes. Although a circuit simplified (in terms of the numbers of PIs, POs, flip-flops, and gate count), does not always lead to an easier proof or disproof, there is a good correlation, so it is generally a good idea to simplify the circuit initially.

Verify Phase. Next, a *verify* phase commences. One of the MC engines is chosen to be run for a given amount of time and if there are parameters to be chosen, these are set. If there is no definitive result (proof or disproof), another engine is chosen and run. The same engine can be chosen with different resources and different parameters. This is iterated until some stopping criteria is reached.

Abstraction phase. The next phase employs abstractions. Usually, the CB/PB localization abstractions [8] are done first followed by refinement using a *verify* phase. Then speculation [3] is done, again followed by refinement using a *verify* phase.

Speculation is the process where equivalences between signals in the circcuit are postulated based on extensive random or semi-formal simulation. By simplifying the circuit using the speculated equivalences, a "speculatively reduced model" is produced with additional outputs representing proof-obligations for the speculations. These obligations can be dispatched all at once or individually. Each of these sub-problems can be attacked as a separate MC problem using any of the sequence of MC engines in the arsenal.

As can be imagined, the orchestration of the use of these engines and their variations becomes quite complex requiring intricate code or use of expert systems [30] [Ziv].

4 Concurrent *verify* phase

If there are initially many properties to be proved for the same design, the obvious way to exploit multi-cores is just to solve each property separately. However, as soon as all easy properties have been solved, we are left with the hard ones, and we claim that CPU power is better used by a well-tuned concurrent engine. In the following, we describe our efforts along these lines and some experimental results.

The scripting layer of our model checker[3] was modified to use a concurrent verify phase, which can fork off a subset of basic MC engines. These run concurrently as separate processes.[4] At the time of the fork, the global state of the Python interpreter and the current AIG for the problem are copied for each process. The list of basic engines that we use are:

1. Four different versions of BDD reachability, *reachx, reachm, reachn, reachp*
2. Four different versions of property directed reachability[5], *pdr, pdrs, pdrm, pdrmm*
3. *Two* different implementations of bounded model checking, *bmc, bmc3*
4. Interpolation, *interpolate*
5. Repeated random simulation, *simulate*

Each engine is given the same timeout. The fork terminates when any algorithm returns a definitive result, SAT or UNSAT, or when all engines have timed out. Only the winning result (which may include a counter-example (CEX), used for refinement) is kept. In the *verify* phase, the AIG is not changed, but the transformation engines used in the next section may also return a new AIG. The Python function call is "*verify*(*list*, *time*)", where *list* is a list of engines to be run concurrently, and *time* is the timeout in seconds for each engine. In contrast to a sequential verify phase, the code for this is markedly simpler.

5 Deployment of a concurrent *verify*

A rough outline of our hybrid concurrent MC algorithm called *c_prove*, which uses the concurrent *verify* is shown in Figure 1.

---

[1] In some rare cases, the forward diameter of the design is know, in which case BMC can be considered a complete method.
[2] In theory, if the synthesis algorithms were powerful enough and the property holds, this would be all that is needed since the circuit can be reduced to a constant. Sometimes this happens.

[3] The ABC model checker is scripted in Python with the capability of orchestrating ABC or ZZ/BIP commands which are written in C or C++.
[4] In our experiments we used an 8-core server running Ubuntu Linux.
[5] PDR is based on Bradley's method and program, which he called *IC3*.

c_prove:
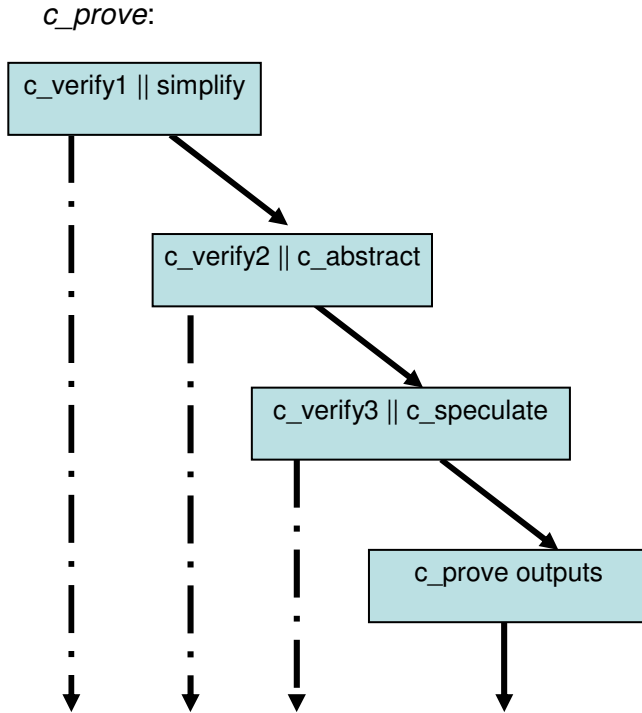
Figure 1. An outline of a hybrid concurrent MC algorithm.

where methods separated by ‖ in boxes are run concurrently, a solid arrow means the result is passed on from a terminating engine, and a dotted arrow means that the *c_verify* from the upper level continues in parallel with other engines that are started later.

If *c_verify*1 or *c_verify*2 returns SAT or UNSAT, the problem is solved and all processes are killed. If *c_verify*3 or *p_prove* at the last level returns UNSAT, the problem is solved, but if either returns SAT, then this may mean only that the output of *abstract* was not a valid abstraction. Thus *simplify*, *c_abstract,* and *c_speculate* are run in sequence and the different *c_verifies* continue to run until killed. Also, if there is reason to suspect that the problem comes from sequential equivalence checking, it may be better to interchange the order of *c_abstract* and *c_speculate*.

Note that there is no separate verify phase as in the sequential flow. This is because *c_abstract* and *c_speculate* are divided into an initial part and a concurrent refinement part. The refinement part is similar to *c_verify* except that when a CEX is found, it is used to refine the abstraction or speculation. Note also that at the end we typically have multiple outputs to prove and *c_prove* is called on each. Since each *c_prove* is concurrent, this may result in many engines running concurrently, so strategies for addressing this are needed. We use one called "poor man's concurrency", which is discussed in Section 8.

A more practical variation on the algorithm is to kill *c_verify1* when *simplify* terminates since it is more likely that *c_verify2*, working on a simpler but equivalent model can terminate sooner. Similar variations can be done to re-

duce the number of concurrent processes for a limited core server and to reduce memory conflict.

We argue that these concurrent versions of MC are stronger than their sequential counterparts, where a particular engine is chosen up front using a heuristic analysis of the design. The argument is that the DAGs created by the forks and joins during abstraction or speculation refinement represent an exponential number of possible sequential executions (shown in Figure 2). Between each fork/join are *p* paths representing the execution of *p* MC engines. For example, if there are 5 engines deployed during a refinement and say 10 refinements are required to find a valid abstraction, then there are $5^{10} \sim 10^7$ different paths effectively explored, all possibly with different outcomes. After an *abstraction* refinement, usually a *speculation* refinement follows with a similar number of paths - so $\sim 10^{14}$ paths are being explored. Thus the sequence of winning engines (in Figure 2, the dotted line indicates the "winning path") is essentially impossible to guess. Many outcomes may be the same, but it is hard to predict what sequence will lead to a useful outcome. Thus, in practice by exploring an inordinate number of paths, the concurrent versions are arguably stronger, with a significantly increased possibility of proving problems not proved by any practical sequential version.

We note that it is not the case that the best result is obtained by first running a chosen engine until a given timeout, then choosing a different engine etc. This would cut down the number of paths that would need to be considered. To illustrate this, below is an actual trace of a run during a refinement of a speculation where the engine that found a CEX first is reported. This is used to refine a set of equivalence classes; then a new speculatively reduced model is created with different proof obligation outputs, representing equivalence classes that need to be proved.

Example: In the example, the sequence of winning engines, the time taken, and the length of the CEX found at each refinement step is shown.

Initial size: PIs=171,POs=41,FF=255, ANDs=2275

```
SIMULATION:     cex 4.268283 sec, frame 911
SIMULATION:     cex 0.096659 sec, frame 17
BMC:            cex 6.534474 sec, frame 17
SIMULATION:     cex 0.726484 sec, frame 1363
SIMULATION:     cex 5.740357 sec, frame 391
BMC:            cex 9.506526 sec, frame 17
SIMULATION:     cex 6.436064 sec, frame 984
SIMULATION:     cex 1.212145 sec, frame 444
PDRM:           cex 4.335237 sec, frame 18
BMC:            cex 9.853237 sec, frame 17
SIMULATION:     cex 6.335866 sec, frame 81
SIMULATION:     cex 4.595637 sec, frame 22
SIMULATION:     cex 4.594522 sec, frame 40
SIMULATION:     cex 9.182059 sec, frame 58
PDRM:           cex 5.637425 sec, frame 20
BMC:            cex 9.861210 sec, frame 17
....
      33 interleavings of PDR PDRM and BMC
....
PDR:            cex 47.217215 sec, frame 29
PDR:            cex 31.134045 sec, frame 76
BMC:            cex 55.010524 sec, frame 23
PDRM:           UNSAT in 66 sec.
```
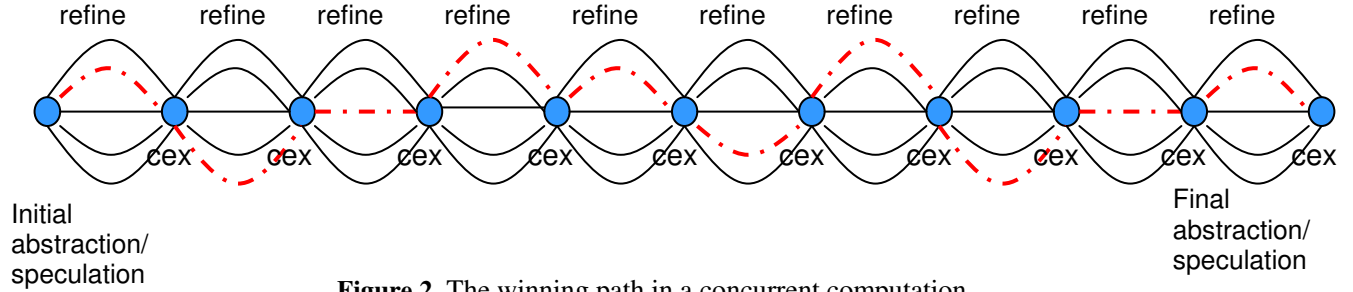
Final size: PIs=171, POs=17, FF=260, ANDs=2346

refine refine refine refine refine refine refine refine refine refine

Initial abstraction/ speculation

cex cex cex cex cex cex cex cex cex cex

Final abstraction/ speculation

Figure 2. The winning path in a concurrent computation

The final size shows that the proof obligations were reduced from 41 to 17 reflecting that many of the initial equivalences were invalid. Additionally, the example illustrates why a separate verify phase is not needed, since *c_speculate* essentially contains *c_verify*.

Note that there is much interleaving between simulation, BMC, and the two PDR engines. Only BMC is necessarily giving a shortest trace. Also, although interpolation was one of the engines used at each refinement, it never was a winner in this benchmark. This is quite typical in our experience. In addition, BDD-based reachability engines were not employed because a heuristic estimates that the problem is too large for BDDs. At the end of the hunt for a CEX, the problem was proved UNSAT by one of the complete engines, PDRM, in 66 sec., illustrating that PDR is a useful bug hunter as well as prover.

6 Implementation in Python and ABC

Python [28] is a scripting language which we use to schedule the calls to the different MC engines in ABC. The Python interface interrogates the solution and the problem size to make decisions about which engine to call next and how much resource to allocate. Python is used in both the older sequential and the new concurrent versions of the model checking done in ABC. We describe some of the details required to make a concurrent version.

A simple Python API was created that accepts a list of Python functions[6] and executes them in independent child processes. When a function terminates, it immediately returns results to the parent process using a pipe, and the child process then terminates. Depending on the result, the parent process can either kill all remaining child processes, or continue waiting for another child process to finish.

These functions can be nested. That is, each child process may fork off a new set of functions etc., thus creating a tree of processes. If a branch returns an "acceptable" result, its parent can cause the siblings to terminate, finally returning this result to the grandparent. This can cause the entire tree to be collapsed, whence the original parent process will continue on to the next verification task.

---

[6] A special Python function *abc(S)* takes a set of ABC commands, *S*, and executes them in sequence by calling ABC code.

This mechanism is implemented on Linux using the *fork*() system call. The parent process uses *fork*() to spawn a child process for each function be run concurrently. Each child process is a copy of the Python interpreter having the same state as the parent -- same global variables, execution hierarchy, copy of the current AIG, and other ABC state information. The child process runs a Python function and as soon as it terminates, the function passes its return values via a pipe to its parent process. In addition, the state of ABC, including a possibly reduced AIG and possibly a CEX, is returned using temporary files.

Process termination must be orderly, so that resources can be cleaned up. Because of the possible large number of processes that can be created and destroyed, leaking even a small percentage would quickly consume more resources than the actual verification job. Ideally, for orderly cleanup to be performed, verification engines should check periodically with the calling code if they should terminate and, if so, return control to the caller. Unfortunately, it is not practical to modify a large code base to do that because it would require pervasive changes to the code as well as constant polling which can hurt performance.

As an alternative, we chose to use the Linux signal mechanism [29] to control termination of processes. Every process installs signal handlers for SIGINT and SIGCHLD signals. SIGINT is sent to a process to request it to terminate, and SIGCHLD is sent to a process when one of its child processes has died. Our signal handler just writes a single byte into a pipe. To handle SIGINT, a thread waits for the signal handler to write to the pipe; then it wakes up, sends SIGINT to all child processes, waits for them to terminate, performs any necessary cleanup and kills itself. To handle SIGCHLD, a thread wakes up as soon as the signal hander writes to the pipe, and then waits for the child process to terminate.

7 Experimental results

We experimented with two sets of "hard" model checking problems, (a) a subset of 381 benchmarks obtained from IBM under a non-disclosure agreement (Table 1) and (b) a subset of hard problems from HWMCC'10 (Table 2). We report initial sizes of these examples and their MC result.

Each problem was run using our Python/ABC concurrent MC version, which is structured similarly to *c_prove* out-

lined in Section 6. Subsets of problems were chosen that we considered "hard":

1) A subset of the IBM benchmarks, not solved by *SixthSense* using its default Expert System flow [30] in two hours (Table 1).[7]
2) A subset of problems not solved by ABC's sequential MC program *super_prove*, the overall HWMCC'10 competition winner, as well as not solved by most other MC programs (including *IC3*) in the competition (Table 2) [8].

We were primarily interested in whether the concurrent version is stronger. This is difficult to measure since the concurrent version uses more resources, so the conclusions about being able to solve problems hitherto "unsolvable" are largely subjective anecdotal. However, the experimental results shown in Table 1 might be convincing. Also see the discussions in Section 8. All experiments were done on an 8-core server with 24 gigabytes of memory.

Table 1- Industrial Examples					
Name	Primary Inputs	Flip flops	And nodes	Result	Time (sec)
bypass33	856	781	11945	Unsat	84
GCT_38	266	607	14308	Unsat	188
pmu_wr_11	74	1072	7155	Unsat	875
tp_p_w_0	35	208	1228	Unsat	601
KML_M_21*	155	3795	20098	Unsat	353
test_hit_4	1570	3107	16701	Unsat	153
two_back62	144	1660	13411	Sat	173
bypass_28_0	156	68	3504	Unsat	9
MCS_MCS_13	247	2654	9985	Unsat	30
sc_sc_0*	249	5609	31029	none	-
DA_DA_11	168	429	4771	Unsat	37
p3_d_n_0	17	197	1355	Sat	180
pclem_0	77	1564	9460	Unsat	193
assert_p_7_0	207	157	3549	Unsat	396
MCA_MCA_0	131	1718	6615	Unsat	24
MCS_rand5	144	2707	10239	Unsat	441
mcx_z_10	4	2269	9974	none	-
sc_ver2_0	19	959	3274	Sat	433
symm_0	34	815	4101	Sat	56
Erat_0	86	396	3016	Unsat	720

*Had multiple outputs; all but the first were folded in as constraints

We tried a few of these examples with the old code but did not succeed in proving any, even when we allowed almost unlimited compute time. Giving it more options would mean a substantial re-write of our old obsolete code.

We comment that roughly an equal number of problems (not shown in Table 1) were tried but still could not be solved by any of the tools. However, Table 1 indicates that many more problems are now solved using the concurrent approach[9].

We attribute this enhanced problem-solving capability mostly to an improved ability to obtain valid refinements after abstraction and speculation. This is absolutely necessary to solve hard problems since it is extremely important to obtain a valid abstraction finally, before proceeding further. Otherwise two things can happen.

1. A CEX is found later, in which case, the abstraction has to be thrown out or repeated using many more resources.
2. A CEX can't be found, but the abstraction may be invalid. However, this can't be determined. So after much time and effort, it is unknown if the abstraction/speculation was valid but still too hard, or if it was invalid.

Also, it can happen that it is harder to disprove an invalid abstraction than to prove a valid more-refined version. The use of concurrency provides the ability to produce more reliable abstractions and hence ultimately solve more problems.

Table 2 - Hard HWMCC'10 Examples					
Name	Prim. Inputs	Flip flops	And nodes	Result	Time (sec.)
bobsmhdlc[0]	61	291	1647	Unsat	434
bobsmhdlc1[0]	61	290	1628	Unsat	450
bobsmhdlc2[0]	61	289	1612	Unsat	1002
bobsmhdlc3[0]	61	300	1574	Unsat	1245
pdtrod6x8p2[1]	9	84	4318	Unsat	1224
pdtpmsudc12[2]	16	36	553	Unsat	48
bobpcihm[0]	304	1422	9627	none	-
bobsminiuart[0]	16	114	571	none	-
bobsmcodic[0]	34	1850	18762	none	-
nusmvqueue[1]	82	84	2376	none	-
pdtpmsudc16[1]	20	48	741	none	-

Notes:
[0] not solved by anyone
[1] solved only by pdtrav
[2] solved only by pdtrav and ABC

---

[7] At the time, the IBM SixSense program did not have a PDR engine, so we eliminated those problems that were made easier because of PDR in our code.

[8] Since HWMCC'10 in July, our entry, *super_prove,* was enhanced considerably by using PDR [4] as another engine. However, we only consider those problems which could still not be solved by the enhanced *super_prove*.

[9] To enhance this argument, we modified our sequential version of *super_prove* by replacing its use of interpolation with PDR, but could not prove any of the hard problems listed even with relatively unlimited time-outs.

In Table 2 we show "hard" problems (with the exception of the Intel examples)[10], from HWMCC'10, some of which we could newly solve only with the concurrent version of our model checker. Four of the eleven problems listed are relatively small and could be solved by *pdtrav* with its advanced BDD reachability engine. The problems starting with '*bob*' are SEC problems. The newly solved ones labeled '*bobsmhdlc**' are those which were created initially by applying increased synthesis effort and then forming the SEC miter with the original. The increased synthesis effort is reflected by increasing runtimes needed to solve the resulting SEC problem. Unless synthesis has a bug, all the '*bob*' miters should be UNSAT. So far no one has proved the last three *bob* problems, possibly emphasizing that SEC is PSPACE-complete.

8 Discussion Topics

Memory Use and Conflicts: We ran all experiments on a server with 24 GB of main memory, two processors, each with 4 cores. Each core had 64KB of L1 cache per core 256KB of L2 cache per processor and 4MB of L3 cache per processor. L2 and L3 caches are not shared across the two processors. Each engine ran in its own part of main memory which is not shared with other engines. Linux does not impose any limitation on the portion of main memory any core can use. That memory did not seem to be an issue in this application, can be attributed to AIG's being very compact; thus duplicating the AIG for each process was not a problem, even when we ran more than 100 processes concurrently. This happens sometimes during speculation, but many of the tasks are easy and dispatched quickly, leaving a small subset on which to work harder but with less memory contention.

Some MC engines can be memory hogs, such as BDD reachability and interpolation, and when their memory usage gets large, the effect on other concurrent engines needs to be analyzed. The more likely effect is through the increase of cache misses on the engines that are concurrent on the same processor. We have not noticed this effect but it should be measured.

Run-time speedup: It has been suggested that an alternative for comparison would be to use the "poor-man's" concurrency approach: use only a single core, initially trying several MC engines in sequence, each with a short timeout, and then doubling the timeout limit and repeating until a definitive answer is found or timeout is reached. In comparison to a 1-core concurrent version, at most a factor of 2 would be lost in time spent, but each engine would have full access to memory. Compared to this, as the number of cores is increased, a concurrent version would see a linear speedup, up to the number of cores available or engines tried concurrently. The concurrent version may be slowed down due to memory contention and the question is by how

much? This experiment remains to be done, but our guess is that this would not be a significant factor, at least with the problems that we have seen.

Poor man's concurrency has its use even when used with concurrent algorithms. For example, if there are a large number of outputs to prove, one could process them in parallel using *c_prove* on each. However, at some limit in the number of outputs, this may be too much concurrency and might bog down a server with a limited number of cores and cache. An alternative is to group the outputs into subsets of size N and prove each set of N outputs in parallel using *c_prove*, sweeping through the groups initially with 2 seconds timeout, then 4 etc. The reason for using the grouping is to avoid too much inefficiency because of memory conflicts.

Beyond the speedup due to the use of more cores, we do see additional speedups on those problems where the older program was able to provide a result and hence provide a total time for comparison. These speedups are due mainly to the sequential program wasting time by making a wrong decision:

1. abstraction or speculation terminates too early with an invalid result,
2. to save time on average, it was decided not to apply BMC to the initial problem when in some cases the problem would have been proved directly,
3. the fastest algorithm is not always chosen for the present design status, in contrast to concurrency which always "chooses" the fastest algorithm.
4. the most powerful algorithm was not applied at the time when a result could have been proved.

In addition, concurrency leads to solving previously unsolved problems, so the time spent on these by the non-concurrent approach is completely wasted. It seems impossible to measure meaningful average speedups under these circumstances.

Wasting processor power: Another topic is that the concurrent version wastes processor power by running so many engines at once, when in the end, only one of the engine's computation is used. On the other hand, there is 'wastage' when any of the cores is sitting idle. In addition, since concurrency gives a speedup even on a single core processor, we are saving power. The alternative would be to run the wrong engine for a longer time.

Other use of concurrency: Finally, we emphasize that this is just a first step in our use of concurrency. We explicitly avoided trying to parallelize any of the individual engines [31] because this quickly becomes a quagmire and many such attempts have not been very successful in the past. In trying to prove more of the hard problems, a next step will be to enable communication between the different engines running concurrently so that information gleaned from one can be used with another, (see [32]). An example would be to use the invariant learned during PDR to enhance interpolation or to enhance proving several properties on the same design at once.

---

[10] These have some problems because of the way they were generated originally and so are not representative.

9 Conclusions and future work

Many MC engines have been developed over the last decade. This has led to significant advances in model checking and its spreading use in industry. Often the techniques are orthogonal in the sense that each engine excels at solving its own but different sub-class of problems. In the absence of good oracles, it is hard to determine which engine is best for a particular problem. The beauty of concurrency applied to model checking is that we do not need to know this. Since this leads to faster more powerful model checkers, simpler code, and makes good use of widely available multi-cores, there is no reason not to invest in concurrent model checkers.

An interesting alternative to the approach proposed in this paper that of [33], which is discussed briefly here.

There are many *basic* model checking engines and within each of these there are different parameter settings. In addition, algorithms can be combined sequentially. The term state-of-the-art (SOTA) model checker has been proposed [34], which runs "all" algorithms in parallel and takes the best result. It represents a theoretical means of assessing the relative performance of a particular engine. With the multiplicity of the MC engines and their variations and combinations, SOTA is not even close to practical, so approximations are used. In [33], is described an expert system which learns which algorithms and parameter settings are better for a particular design project and runs several in parallel dynamically starting and stopping them depending on their current performance. Good results are obtained in comparison with an approximate 16-algorithm SOTA. In contrast, our hybrid MC algorithm uses concurrent execution of basic as well as hybrid engines. It would be interesting to compare the two approaches on a common set of benchmarks.

Future work will be to study more carefully how memory conflicts show up on larger problems or those where hard problems give rise to many engines using a large amount of memory. Also, there are still some problems which are temptingly small but surprisingly hard and do not yield to any of the techniques devised so far. A continuing challenge is to discover new engines to fill these gaps. The development of the PDR method [4] is a good example that this is still possible.

10 Acknowledgements

11 References

[1] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen, "Scalable sequential equivalence checking across arbitrary design transformations." *Proc. ICCD'06*.

[2] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman. "Exploiting suspected redundancy without proving it". *Proc. DAC'05*, pp. 463-466.

[3] H. Mony, J. Baumgartner, A. Mishchenko, and R. Brayton, "Speculative reduction-based scalable redundancy identification", *Proc. DATE'09*, pp. 1674-1679.

[4] A. R. Bradley, "k-step relative inductive generalization", http://arxiv.org/abs/1003.3649

[5] Berkeley Verification and Synthesis Research Center (BVSRC). http://www.bvsrc.org

[6] A. R. Bradley and Z. Manna, "Checking safety by inductive generalization of counterexamples to induction", *Proc. FMCAD'07*.

[7] F. Somenzi, *BDD package CUDD*. http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html

[8] N. Een, A. Mishchenko, and N. Amla, "A single-instance incremental SAT formulation of proof- and counterexample-based abstraction". *Proc. FMCAD'10*.

[9] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool", *Proc. CAV'10*, Springer, LNCS 6174, pp. 24-40.

[10] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis", *Proc. DAC '06*, pp. 532-536.

[11] http://fmv.jku.at/hwmcc10/

[12] G. Cabodi, P. Camurati, L. Garcia, M. Murciano, S. Nocco, and S. Quer, "Speeding up model checking by exploiting explicit and hidden verification constraints". *Proc. DATE '09*, pp. 1686-1691

[13] A. P. Hurst, A. Mishchenko, and R. K. Brayton, "Fast minimum-register retiming via binary maximum-flow", *Proc. FMCAD '07*.

[14] K. L. McMillan, "Interpolation and SAT-based model checking". *Proc. CAV'03*, pp. 1-13.

[15] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. "Bounded model checking using satisfiability solving", *Proc. Formal Methods in System Design (FMSD)*, vol. 19(1), Kluwer 2001

[16] http://domino.research.ibm.com/comm/research_projects.nsf/pages/sixthsense.index.html

[17] L. Fix, "Fifteen years of formal property verification in Intel". *25 Years of Model Checking*, 2008, pp. 139-144.

[18] Berkeley Verification and Synthesis Research Center. *ABC: A System for Sequential Synthesis and Verification*. http://www.eecs.berkeley.edu/~alanmi/abc/

[19] R. P. Kurshan, *Computer-Aided-Verification of Coordinating Processes*. Princeton Univ. Press, 1994.

[20] K. McMillan and N. Amla, "Automatic abstraction without counterexamples". *Proc. TACAS'03*.

[21] A. Gupta, M. Ganai, Z. Yang, and P. Ashar. "Iterative abstraction using SAT-based BMC with proof analysis". *Proc. ICCAD'03*.

[22] J. Baumgartner and H. Mony, "Maximal input reduction of sequential netlists via synergistic reparameterization and localization strategies", *Proc. CHARME'05*, pp. 222-237.

[23] M. L. Case, H. Mony, J. Baumgartner, R. Kanzelman. "Enhanced verification by temporal decomposition". *Proc. FMCAD'09*, pp.17~24

[24] P. Cabodi and S. Quer, *PdTrav Package*, http://fmgroup.polito.it/quer/research/tool/tool.htm

[25] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan and D. L. Dill, "Symbolic model checking for sequential circuit verification", *IEEE TCAD*, vol. 13(4), 1994, pp. 401-424.

[26] A. Mishchenko, M. L. Case, R. K. Brayton, and S. Jang, "Scalable and scalably-verifiable sequential synthesis", *Proc. ICCAD'08*, pp. 234-241.

[27] P. Bjesse and J. Kukula, "Automatic generalized phase abstraction for formal verification", *Proc. ICCAD '05*.

[28] Python programming language, official website - http//python.org

[29] The single UNIX specification http://pubs.opengroup.org/onlinepubs/007908799/

[30] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable Automated Verification via Expert-System Guided Transformations." *FMCAD*, 2004

[31] J. Barnat, L. Brim, M. Ceška, and T. Lamr, "CUDA accelerated LTL Model Checking", *ICPADS 2009*, pages 34–41.

[32] P.-H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, J. Long, "Smart Simulation Using Collaborative Formal and Simulation Engines," ICCAD 2000.

[33] Z. Nevo. User-Friendly Model Checking: Automatically Configuring Algorithms with RuleBase/PE. Proceedings of Haifa Verification Conference 2008. pp.210~214.

[34] M. Narizzano, L. Pulina, A. Tacchella,, "The 3rd QBF solvers comparative evaluation", Journal on Satisfiability, Boolean Modeling and Computation, 2006, pp. 145–164.