# Parallelising Symbolic State-Space Generators*

Jonathan Ezekiel[1], Gerald Lüttgen[1], and Gianfranco Ciardo[2]

[1] University of York, York, YO10 5DD, U.K.
{jezekiel,luettgen}@cs.york.ac.uk
[2] University of California, Riverside, CA 92521, U.S.A.
ciardo@cs.ucr.edu

**Abstract.** Symbolic state-space generators are notoriously hard to parallelise, largely due to the irregular nature of the task. Parallel languages such as Cilk, tailored to irregular problems, have been shown to offer efficient scheduling and load balancing. This paper explores whether Cilk can be used to efficiently parallelise a symbolic state-space generator on a shared-memory architecture. We parallelise the Saturation algorithm implemented in the SMART verification tool using Cilk, and compare it to a parallel implementation of the algorithm using a thread pool. Our experimental studies on a dual-processor, dual-core PC show that Cilk can improve the run-time efficiency of our parallel algorithm due to its load balancing and scheduling efficiency. We also demonstrate that this incurs a significant memory overhead due to Cilk's inability to support pipelining, and conclude by pointing to a possible future direction for parallel irregular languages to include pipelining.

## 1 Introduction

*Automated verification*, such as temporal-logic model checking [8], relies on efficient algorithms for computing state spaces of complex system models. To avoid the well-known state-space explosion problem, symbolic algorithms working on *decision diagrams*, usually BDDs, have proved successful in practise [7, 16]. Several efforts have been made to implement these algorithms on parallel computer platforms, most notably on networks of workstations and on PC clusters [11, 12, 13, 17, 19]. The efforts range from simple approaches that essentially implement BDDs as two-tiered hash tables [17, 19], to sophisticated approaches relying on *slicing* BDDs [12] and techniques for *workstealing* [11]. However, the resulting implementations show only limited speedups.

While parallel implementations of symbolic model checkers are often successful in increasing available memory, limited speedups can largely be attributed to the irregular nature of the state-space generation task and the resulting high parallel overheads such as load imbalance and scheduling of small computations. When combined with the extra overheads incurred from synchronisation on the symbolic data structure, it is possible for irregularity to severely decrease runtime efficiency. Irregular problems have been addressed in the parallel literature,

* Research funding was provided by the EPSRC under grant no. GR/S86211/01.

W. Damm and H. Hermanns (Eds.): CAV 2007, LNCS 4590, pp. 268–280, 2007.

resulting in languages such as *Cilk* [1, 10] for shared memory architectures. Cilk has been shown to alleviate the irregular overheads by offering efficient scheduling and load balancing. When successfully applied, it offers potential improvements in time efficiency, and a large reduction in effort with respect to deriving and implementing scheduling and load balancing techniques. To date, Cilk has been used for other irregular problems involving searches, but overlooked for parallelising state-space generation, which underlies model checking.

*Saturation* [5], as implemented in the verification tool SMART [4], is a symbolic state-space generation algorithm with unique features (cf. Sec. 2). It is intended for asynchronous system models with interleaving semantics, and exploits the local effect of firing events on state vectors by locally manipulating MDDs, which are a generalisation of BDDs [14]. Saturation has proved to be orders of magnitude more time-efficient and memory-efficient than other symbolic algorithms [5], including the one in NuSMV [7], when applied to asynchronous system models. Like other symbolic algorithms, Saturation is irregular in nature and suffers from high parallelisation overheads. Hence, the question arises as to whether using a proven parallel language for irregular problems is beneficial to the time efficiency of a parallel implementation of Saturation. A previous approach to parallelising Saturation [2] on a PC cluster used a message-passing library, but not a language tailored to irregular problems.

This paper investigates the parallelisability of the Saturation algorithm for shared-memory architectures using Cilk and reports on our experiences made. Our implementation (cf. Sec. 3) focuses on shared-memory architectures, but due to the increasing popularity of *distributed shared-memory* libraries, our results are also of significance for parallelisations of Saturation on PC clusters. To put our results into context, we contrast our Cilk algorithm with our own thread pool parallelisation of Saturation for shared memory architectures [9], which is based on the POSIX Pthreads library [15], and compare run-time and memory efficiency. We extend our investigation to optimise the parallel ordering in which the state space is generated, and determine the effects on run-time and memory for both parallel implementations. Our experimental studies (cf. Sec. 4) using a PC with two dual-core Intel processors show that the efficiency of Cilk improves the run-time of the parallel algorithm when compared to our thread pool implementation, but incurs a significant increase in memory due to Cilk's inability to support pipelining. Our experiences show how parallel irregular languages can be considered when parallelising symbolic state-space generators, and we conclude by pointing to a potential future direction within the parallel community which may allow parallel irregular languages to improve the time-efficiency of parallel state-space generation without severely impacting on memory (cf. Sec. 6).

## 2 Background

A discrete-state model is a triple $(\widehat{\mathcal{S}}, \mathbf{s}^0, \mathcal{N})$, where $\widehat{\mathcal{S}}$ is the set of *potential states* of the model, $\mathbf{s}^0 \in \widehat{\mathcal{S}}$ is the *initial state*, and $\mathcal{N} : \widehat{\mathcal{S}} \to 2^{\widehat{\mathcal{S}}}$ is the *next-state function* specifying the states reachable from each state in one step. Assuming that the

model contains $K$ *submodels*, a (*global*) state $\mathbf{i}$ is a $K$-tuple $(i_K,...,i_1)$, where $i_k$ is the *local* state of submodel $k$, for $K \geq k \geq 1$, and $\widehat{\mathcal{S}} = \mathcal{S}_K \times \cdots \times \mathcal{S}_1$ is the cross-product of $K$ *local state-spaces*. This allows us to use *symbolic* techniques based on decision diagrams to store sets of states. We decompose $\mathcal{N}$ into a disjunction of next-state functions, so that $\mathcal{N}(\mathbf{i}) = \bigcup_{e\in\mathcal{E}} \mathcal{N}_e(\mathbf{i})$, where $\mathcal{E}$ is a finite set of *events* and $\mathcal{N}_e$ is the next-state function for event $e$. We seek to build the *reachable state-space* $\mathcal{S} \subseteq \widehat{\mathcal{S}}$, the smallest set containing $\mathbf{s}^0$ and closed with respect to $\mathcal{N}$: $\mathcal{S} = \{\mathbf{s}^0\} \cup \mathcal{N}(\mathbf{s}^0) \cup \mathcal{N}(\mathcal{N}(\mathbf{s}^0)) \cup \cdots = \mathcal{N}^*(\mathbf{s}^0)$, where "*" denotes reflexive and transitive closure and $\mathcal{N}(\mathcal{X}) = \bigcup_{\mathbf{i}\in\mathcal{X}} \mathcal{N}(\mathbf{i})$.

**Symbolic Encodings of $\mathcal{S}$ and $\mathcal{N}$.** In the sequel, we assume that each $\mathcal{S}_k$ is finite and known a priori. In practise, the local state spaces $\mathcal{S}_k$ can actually be generated on-the-fly by interleaving symbolic global state-space generation with explicit local state-space generation [6]. Without loss of generality, we assume that $\mathcal{S}_k = \{0, 1, \ldots, n_k-1\}$, with $n_k = |\mathcal{S}_k|$. We then encode any set $\mathcal{X} \subseteq \widehat{\mathcal{S}}$ in a *(quasi-reduced ordered) MDD* over $\widehat{\mathcal{S}}$. Formally, an MDD is a directed acyclic edge-labelled multi-graph where:

- Each node $p$ belongs to a *level* $k \in \{K, ..., 1, 0\}$, denoted $p.lvl$.
- There is a single *root* node $r$ at level $K$.
- Level 0 can only contain the two *terminal* nodes *Zero* and *One*.
- A node $p$ at level $k > 0$ has $n_k$ outgoing edges, labelled from 0 to $n_k-1$. The edge labelled by $i_k$ points to a node $q$ at level $k-1$; we write $p[i_k] = q$.
- Given nodes $p$ and $q$ at level $k$, if $p[i_k] = q[i_k]$ for all $i_k \in \mathcal{S}_k$, then $p = q$, i.e., there are no *duplicates*.

The set encoded by an MDD node $p$ at level $k > 0$ is $\mathcal{B}(p) = \bigcup_{i_k\in\mathcal{S}_k} \{i_k\} \times \mathcal{B}(p[i_k])$, letting $\mathcal{X} \times \mathcal{B}(\mathbf{0}) = \emptyset$ and $\mathcal{X} \times \mathcal{B}(\mathbf{1}) = \mathcal{X}$ for any set $\mathcal{X}$

For storing $\mathcal{N}$, we adopt a representation inspired by work on Markov chains. This requires the model to be *Kronecker consistent* [5], a restriction that can often be automatically satisfied by concurrency models such as Petri nets. Each $\mathcal{N}_e$ is conjunctively decomposed into $K$ local next-state functions $\mathcal{N}_{k,e}$, for $K \geq k \geq 1$, satisfying $\mathcal{N}_e(i_K, \ldots, i_1) = \mathcal{N}_{K,e}(i_K) \times \cdots \times \mathcal{N}_{1,e}(i_1)$, in any global state $(i_K, \ldots, i_1) \in \widehat{\mathcal{S}}$. Using $K \cdot |\mathcal{E}|$ matrices $\mathbf{N}_{k,e} \in \{0, 1\}^{n_k\times n_k}$ with $\mathbf{N}_{k,e}[i_k, j_k] = 1 \Leftrightarrow j_k \in \mathcal{N}_{k,e}(i_k)$, we encode $\mathcal{N}_e$ as a boolean Kronecker product: $\mathbf{j} \in \mathcal{N}_e(\mathbf{i}) \Leftrightarrow \bigotimes_{K\geq k\geq 1} \mathbf{N}_{k,e}[i_k, j_k] = 1$, where $\otimes$ indicates the Kronecker product of matrices. The $\mathbf{N}_{k,e}$ matrices are extremely sparse; when encoding a Petri net, for example, each row contains at most one nonzero entry.

**Saturation-Based Iteration Strategy.** In addition to efficiently representing $\mathcal{N}$, the Kronecker encoding allows us to recognise *event locality* [5] and employ *Saturation* [5]. We say that event $e$ is *independent* of level $k$ if $\mathbf{N}_{k,e} = \mathbf{I}$, the identity matrix. Let $Top(e)$ denote the highest level for which $\mathbf{N}_{k,e} \neq \mathbf{I}$. An MDD node $p$ at level $k$ is *saturated* if it is a fixed point with respect to all $\mathcal{N}_e$ such that $Top(e) \leq k$, i.e., $\mathcal{S}_K \times \cdots \times \mathcal{S}_{k+1} \times \mathcal{B}(p) = \mathcal{N}_{\leq k}(\mathcal{S}_K \times \cdots \times \mathcal{S}_{k+1} \times \mathcal{B}(p))$, where $\mathcal{N}_{\leq k} = \bigcup_{e:\, Top(e)\leq k} \mathcal{N}_e$. To saturate MDD node $p$ once all its descendants

are saturated, we *update it in place* so that it encodes also any state in $\mathcal{N}_{k,e} \times \cdots \times \mathcal{N}_{1,e}(\mathcal{B}(p))$, for all events $e$ such that $Top(e) = k$. This can create new MDD nodes at levels below $k$, which are saturated immediately, prior to completing the saturation of $p$. If we start with the MDD encoding the initial state $\mathbf{s}^0$ and saturate its nodes bottom up, the root $r$ will encode $\mathcal{S} = \mathcal{N}^*(\mathbf{s}^0)$ at the end, as shown in [5].

Saturation consists of many "lightweight" nested fixed-point iterations and is completely different from the traditional breadth-first approach that employs a single "heavyweight" global fixed-point iteration. The algorithm contains two main mutually recursive functions (cf. Sec. 3): *Saturate* calls *Fire* to recursively perform the event firings while saturating nodes, while *Fire* calls *Saturate* to saturate nodes that are created as a result of event firings. The algorithm also uses supporting functions for creating and deleting nodes, performing a union on two nodes, storing saturated nodes by checking them into a hash table, and caching results to previous calls of *Fire*. Experimental results reported in [3, 5, 6] consistently show that Saturation outperforms breadth-first symbolic state-space generation by orders of magnitude in both memory and time, making it arguably the most efficient state-space generation algorithm for globally-asynchronous locally-synchronous discrete event systems.

**Cilk.** Symbolic state-space generation algorithms incur significant overheads from parallelisation, making gains in time-efficiency difficult to achieve. The main overheads are *synchronisation overheads* due to frequent locking on the symbolic structure (i.e., nodes stored in hash tables), *load imbalance* from the irregular sizes of computations during state-space generation, and *scheduling overheads* since state-space generation computations can be small. Parallel tools to reduce these overheads are thus desirable. To the best of our knowledge, Cilk [1, 10] is the only parallel language that offers both efficient scheduling and load balancing. The Cilk language simplifies parallel programming by allowing the use of C-based functions to express control over the parallelism of a program. The language is powerful enough to facilitate mutually recursive algorithms such as Saturation [5]. It is designed to run efficiently on symmetric processors, e.g., those found in shared-memory machines, and includes a scheduler employing randomised work-stealing, that is theoretically and practically efficient. To achieve efficiency, Cilk employs its own model of multithreaded computation.

Cilk uses call/return semantics to enable parallelism, and provides keywords that enable the programmer to easily express parallelism. A Cilk function can be specified by using the keyword **cilk** in front of a C function, and can be *spawned* to run in parallel by using the keyword **spawn** when calling it. The C function semantics is preserved by allowing the return value of the spawned function to be stored by the parent. Multiple functions can be spawned within the calling function, and the calling function continues its computation while the spawned functions work in parallel. To permit controlled synchronisation of spawned threads, the **sync** keyword prevents the calling function from continuing its computation until all of its spawned functions have completed. Cilk functions contain an implicit **sync** before they are allowed to return.

The return value of the calling function can either be stored by the parent once the function completes, or can be handled by the parent in a more complex way via the use of an **inlet**. An inlet can be specified as an internal function to a Cilk function, which handles the result of a spawned function. To preserve atomicity, only one completed Cilk function can be handled at a time by the inlet, and further computation by the parent is prevented until the inlet has returned. The spawn and sync keywords cannot be used within an inlet. This restriction arises from Cilk's inability to support pipelining, making it difficult to express *producer/consumer* problems such as state-space generation.

## 3 Parallel Saturation

Using Cilk we can easily interpret Saturation as a parallel algorithm in *divide and conquer* format. The algorithm in Fig. 1 shows the original Saturation algorithm [5] expressed as a parallel algorithm in Cilk. The algorithm is parallelised via *task parallelism* in exactly the same way as in our thread pool implementation using POSIX Pthreads [9], where the $Fire$ function is defined as a parallel task, so that event firings can execute in parallel. We therefore choose to spawn the function *Fire* on line 16 of the algorithm, while the return value of the spawned function is handled using an inlet we call $DoUnion$, specified in lines 1 to 9 of the algorithm. The inlet performs the *Union* on the node being saturated if the firing returns a non-zero node. The calling function synchronises on the spawned firings in line 17 of the Cilk algorithm using the keyword sync. The firing loop continues again when all of the currently spawned firings have completed. Access to the hash table and caches is granted on a per-level basis via a mutex lock that

```
cilk Saturate(in k:lvl, p:node)
  Update p, a node at level k not in the hash table,
  in–place, to encode N*_{≤k}(B(p)).
  declare pCng : bool; e : event; i, j : lcl;
  declare L : set of lcl; u : node;
1.  inlet void DoUnion(f : lcl) {
2.    if f ≠ 0 then
3.    foreach j ∈ N_{k,e}(i) do
4.      u ⇐ Union(k−1, f, p[j]);
5.      if u ≠ p[j] then
6.        p[j] ⇐ u; pCng = true;
7.        if N_{k,e}(j) ≠ 0 then
8.          L=L ∪ {j};
9.  }
10. repeat
11.   pCng ⇐ false;
12.   for each e ∈ E_k do
13.     L = Locals(e,k,p);
14.     while L ≠ ∅ do
15.       i = Pick(L);
16.       DoUnion(spawn Fire(e, k−1, p[i]));
17.   sync;
18. until pCng = false;
```

```
cilk Fire(in e:event, l:lvl, q:node):node
  Build an MDD rooted at level l, encoding
  N*_{≤l}(N_e(B(q))).
  declare L : set of lcl;
  declare i, j : lcl;
  declare f, u, s : node;
  declare sCng : bool;
1.  if l < Last(e) then return q;
2.  if Find(FireCache[l], {q, e}, s) return s;
3.  s ⇐ NewNode(l); sCng ⇐ false;
4.  L ⇐ Locals(e, l, q);
5.  while L ≠ ∅ do
6.    i ⇐ Pick(L);
7.    f ⇐ Fire(e, l−1, q[i]);
8.    if f ≠ 0 then
9.      foreach j ∈ N_{l,e}(i) do
10.       u ⇐ Union(l−1, f, s[j]);
11.       if u ≠ s[j] then
12.         s[j] ⇐ u; sCng ⇐ true;
13. if sCng then Saturate(l, s);
14. CheckIntoHashTable(l, s);
15. Insert(FireCache[l]{q, e}, s);
16. return s;
```

**Fig. 1.** Cilk based Saturation using inlets

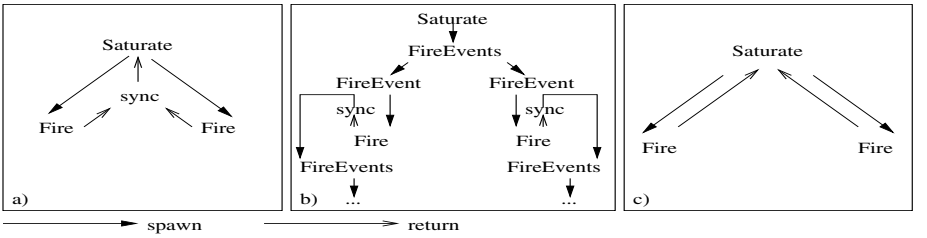


**Fig. 2.** Calling order for spawns

can be specified in Cilk. We argued the correctness of this way of parallelising Saturation in [9].

Unfortunately, this parallelisation approach creates a load imbalance since all firings must be completed before performing the union operation. The ordering can be shown in Fig. 2(a), where function *Saturate* must wait for the two spawned *Fire* calls to synchronise before spawning more work. It would be more efficient to perform the union operation and then immediately spawn new work using the ordering in Fig. 2(c).

**Expressing a Producer/Consumer Problem.** The call/return semantics of Cilk means that we cannot elegantly deal with a spawned function as soon as it has completed, since we cannot tell when an individual firing has completed outside of an inlet. It is desirable to use an inlet to spawn off more work as soon as a firing completes; however, inlets are restricted to prevent new functions being spawned from within them. We could attempt to let the calling function know when a firing has completed via an inlet through the use of a flag or a queue, but Cilk does not allow us to suspend the calling function outside of a sync statement, which means that the calling function would have to continue monitoring for completed child functions. It is undesirable to tie up the processor with a function that is polling in this manner, since it largely performs useless work. This means that the ordering of work shown in Fig. 2(c) cannot be achieved using Cilk, due to the restrictions arising from Cilk's lack of pipelining in its multithreaded computational model.

We can rewrite the algorithm to continue spawning firings when they have completed by utilising the spawn keyword, without exploiting the call/return semantics of Cilk. An example algorithm is shown in Fig. 3, which breaks the original Saturation function into sub-functions. Once a spawned firing has been completed, it performs the union in *DoUnion* and then immediately spawns further firings on the updated state. Expressing our producer/consumer problem by bypassing the call/return semantics is not ideal. When the functions complete, they do not have any further work to do, yet they are left on the Cilk function stack after spawning more work, waiting for it to complete. This ordering is shown in Fig. 2(b). A large number of functions can be unnecessarily left on the stack during the state-space generation process, which potentially increases

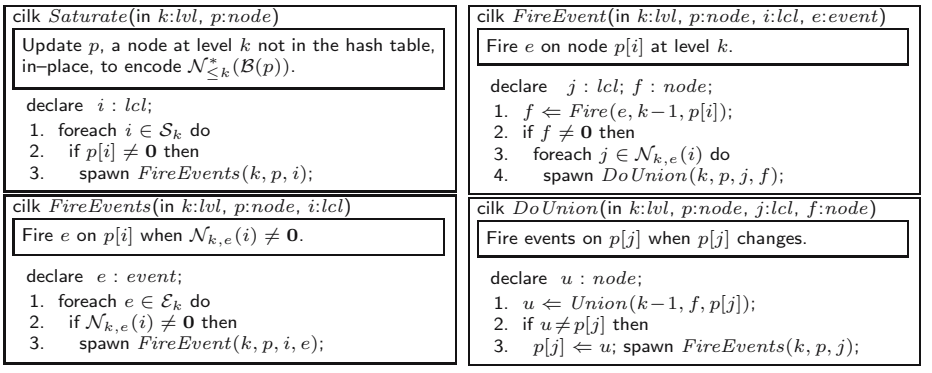


**Fig. 3.** Cilk based Saturation without exploiting call/return semantics

the amount of memory required for the process. The problem is compounded because of the mutually recursive calls between *Saturate* and *Fire*.

**Using a Thread Pool.** To achieve the ideal ordering in Fig. 2(c), we must relinquish functions from the stack while spawned work executes. Since a function frame requires its own storage, a smaller amount of memory could be used by storing only the variables that are required once a spawned child is complete, instead of storing the calling function. We can use a thread pool for load balancing purposes, an auxiliary structure to store required variables, and structure our algorithm to relinquish functions, leaving the child functions to complete the work of the calling function. In our thread pool algorithm [9], we store the variables in *upward arcs* in the MDD structure. Children can be spawned using *tasks* allocated to threads in the thread pool via the use of a FIFO queue. An available thread will pick up a task from the queue and execute it. Tasks can restore the status of their calling function using the upward arcs, which allows calling functions to terminate, leaving spawned tasks to complete their work.

A snippet of the pseudo-code from the thread pool algorithm is shown in Fig. 4. The algorithm behaves (or acts) in much the same way as the sequential Saturation algorithm, except that, when a firing is performed in function *Fire*, an upward arc is set to the node that needs to be updated as a result of the firing. This allows the *Fire* call to terminate since the upward arc contains the information required for spawned tasks to complete the *Fire* function. The mutual recursion on the function stack is broken as *Fire* spawns *Saturate* tasks, i.e., once the node created by *Fire* is ready to be saturated, a saturation task is added to the queue and *Fire* terminates. To determine whether a node has been saturated, the number of tasks performing computations on the node needs to be stored. When all tasks have completed, the node is saturated and the function *NodeSaturated* is called. *NodeSaturated* picks up where *Fire* left off, updating any of the nodes which have upward arcs set to them, and continues to fire events on any updated node until the node is saturated.

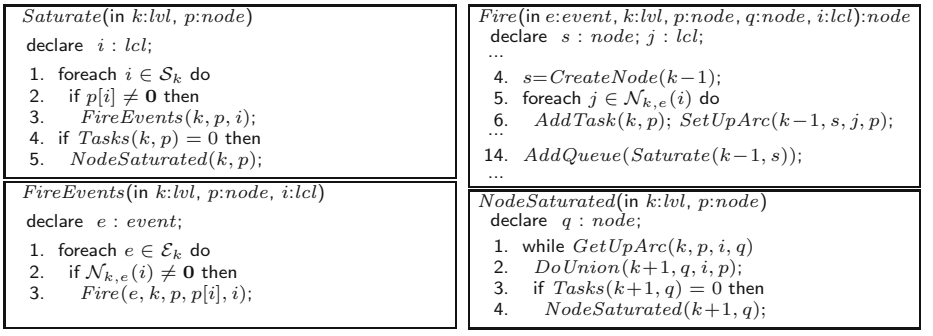


**Fig. 4.** Thread pool Saturation [9]

The use of upward arcs introduces its own overheads [9]: additional locks, task management, and the thread pool queue. The upward arcs also require extra memory for both the arcs and the locks to synchronise the arcs. The thread pool is not as efficient as Cilk, due to the time required to add and remove a task from the queue. When we compare Cilk to a thread pool using the functionally lightweight *Fibonacci* problem in [10] on a dual-processor, dual-core machine, Cilk reports a 3× speedup whereas the thread pool reports a 2× slowdown due to the time spent adding and removing tasks from the queue. Our thread pool is, however, very efficient compared to creating threads on demand, where the allocation of work to a thread is over 10× faster than creating one. [9].

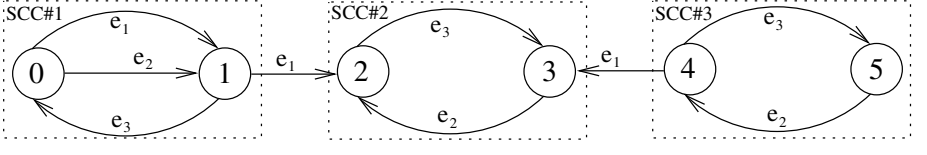


**Fig. 5.** The effect of events with $Top = k$ on the states $\{0, 1, 2, 3, 4, 5\}$ of $\mathcal{S}_k$

**Optimising the Ordering of Events.** The study of the thread pool algorithm in [9] revealed that the ordering in which events are fired in parallel can significantly affect Saturation's run-time and memory requirements. Fig. 5 shows the effect of events $e_1$, $e_2$, and $e_3$, with $Top(e_1) = Top(e_2) = Top(e_3) = k$, on the local states at level $k$ (these events may of course affect lower levels as well). When saturating a node $p$ at level $k$, we must repeatedly fire $e_1$, $e_2$, and $e_3$ in $p$, until no more new states are found, i.e., until $p$ encodes a fixed point. However, Saturation does not dictate the order in which these events should be fired. For example, firing $0 \xrightarrow{e_1} 1$ followed by $1 \xrightarrow{e_3} 0$ might be sub-optimal, since we might have to fire $1 \xrightarrow{e_3} 0$ again once $0 \xrightarrow{e_2} 1$ has been fired, if this causes $p[1]$ to point to a different node encoding more states. Similarly, if we fire $1 \xrightarrow{e_1} 2$ before firing $0 \xrightarrow{e_2} 1$, all transitions in $SCC\#2$ may have to be fired again.

To address this problem, we use the *chaining heuristic* of [3], which extracts the strongly connected components (SCCs) from a *dynamic transition graph* that is built from the static graph of Fig. 5 and the dynamic pattern of non-zero children of node $p$. We use these SCCs to enhance the order of parallel event firings. However, while this chaining heuristic tends to improve run-time and memory in a sequential implementation, it also reduces the potential parallelism and introduces time and memory overheads due to storing the SCC graphs, traversing them and managing parallel access.

## 4 Experimental Results

We implemented four experimental algorithms, two using Cilk and two utilising a thread pool, which either fire events using the unoptimised ordering or the chaining heuristic. Our thread pool algorithms were implemented using C and the POSIX Pthreads library [15]. The machine used for this evaluation is a dual-processor, dual-core PC with 2GB of memory and Intel Xeon CPU 3.06GHz processors with 512KB cache sizes, running Redhat Linux AS 4, Redhat kernel 2.6.9-22.ELsmp, with glibc 2.3.4-2.13. We applied the algorithms to a set of parameterised models previously used to evaluate Saturation [3, 5, 6], where the parameter controls the size of a model's state space. One of our models is the Runway Safety Monitor (RSM) designed by Lockheed Martin and NASA to reduce aviation accidents [18]. For each model, Table 1 shows the run-time and memory when executing our experimental algorithms on four cores, where $N$ is the parameter and $|S|$ is the approximate size of the final state space. The thread pool algorithms are denoted by *TP*, the Cilk algorithms are denoted by *Cilk*, and a *C* at the end of these names indicates the use of chaining. We ran only non-chained versions of the algorithms on the FMS and Philosophers models since the SCC graphs using the chaining heuristic were too large to fit into memory.

The *run-time speedup* shows the comparative speed of the parallel algorithms against the sequential version, where a value greater (less) than 1 indicates a speedup (slowdown). The ideal speedup on our machine is approximately 3.2, since the speedup obtained from a secondary core is less than that of a secondary processor. A speedup greater than 3.2 is a superlinear speedup, which occurs when the parallelism introduced into an algorithm causes it to be more optimal than its sequential counterpart. It is difficult to achieve an ideal speedup for any parallel search algorithm due to the *search overhead factor*. For Saturation this includes synchronisation overheads on the symbolic structure, where the MDD has to be locked frequently, and model specific factors such as how small the computations are. In contrast to standard breadth-first state exploration techniques, Saturation is heavily optimised. Hence, many of its computations are extremely small and are thus difficult to parallelise efficiently [9].

Our experimental results reflect these overheads and model specific factors, since only seven of the models exhibit parallelism, varying from small speedups of just over 1 to a superlinear speedup of over 4. The results, however, also show

**Table 1.** Run-time and memory results on a dual-processor, dual-core machine

| $N$ | $\|S\|$ | | Run-time speedup | | | | | Memory increase | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Seq (s) | Tp | TpC | Cilk | CilkC | Seq (b) | Tp | TpC | Cilk | CilkC |
| **Slotted ring network protocol (Slot)** $N$ = no. of nodes in the network | | | | | | | | | | | |
| 90 | $5.9 \times 10^{94}$ | 6.79 | 0.26 | 0.31 | 0.28 | 0.42 | 5923040 | 11.84 | 5.55 | 2.52 | 1.40 |
| 120 | $5.1 \times 10^{126}$ | 15.80 | 0.29 | 0.33 | 0.29 | 0.44 | 13405440 | 12.00 | 5.67 | 2.68 | 1.44 |
| 150 | $4.5 \times 10^{158}$ | 30.84 | 0.30 | 0.37 | 0.31 | 0.45 | 25441840 | 12.10 | 5.88 | 2.73 | 1.52 |
| **Round robin mutex protocol (Robin)** $N$ = no. of processors | | | | | | | | | | | |
| 180 | $6.2 \times 10^{56}$ | 7.94 | 0.94 | 0.49 | 1.15 | 0.69 | 1165764 | 1.88 | 1.82 | 18.01 | 17.51 |
| 210 | $7.8 \times 10^{65}$ | 15.43 | 0.96 | 0.50 | 1.18 | 0.70 | 1574304 | 1.91 | 1.90 | 18.10 | 17.58 |
| 240 | $9.5 \times 10^{74}$ | 41.71 | 0.99 | 0.50 | 1.22 | 0.72 | 2044044 | 1.94 | 1.92 | 18.13 | 17.59 |
| **Kanban manufacturing system (Kanban)** $N$ = no. of each type of parts | | | | | | | | | | | |
| 25 | $7.6 \times 10^{12}$ | 2.79 | 0.65 | 0.57 | 0.22 | 0.65 | 7334600 | 1.92 | 1.22 | 1.48 | 1.16 |
| 30 | $5.0 \times 10^{13}$ | 5.07 | 0.69 | 0.57 | 0.24 | 0.66 | 13784976 | 2.00 | 1.27 | 1.49 | 1.18 |
| 35 | $2.5 \times 10^{14}$ | 10.25 | 0.71 | 0.58 | 0.25 | 0.66 | 23957940 | 2.08 | 1.31 | 1.54 | 1.19 |
| **Flexible manufacturing system (FMS)** $N$ = no. of each type of parts | | | | | | | | | | | |
| 11 | $1.1 \times 10^{9}$ | 12.22 | 2.11 | N/A | 2.19 | N/A | 3148980 | 1.72 | N/A | 22.89 | N/A |
| 13 | $5.8 \times 10^{9}$ | 55.54 | 2.18 | N/A | 2.35 | N/A | 8173844 | 1.88 | N/A | 23.56 | N/A |
| 14 | $1.3 \times 10^{10}$ | 119.87 | 2.26 | N/A | 2.47 | N/A | 12591300 | 1.97 | N/A | 24.35 | N/A |
| **Queen problem (Queens)** $N$ = no. of queens on an $N \times N$ chessboard | | | | | | | | | | | |
| 11 | 166926 | 1.78 | 0.52 | 0.53 | 1.22 | 0.60 | 4248776 | 1.91 | 2.05 | 17.68 | 17.99 |
| 12 | 856189 | 19.38 | 0.56 | 0.54 | 1.52 | 0.62 | 19920672 | 2.01 | 2.15 | 18.27 | 18.20 |
| 13 | 4674890 | 438.59 | 0.61 | 0.59 | 2.48 | 0.67 | 99807456 | 2.24 | 2.56 | 19.01 | 18.62 |
| **Runway safety monitor (RSM)** Targets = 1, Speeds = 2, X=$N$, Y=3, Z=2 | | | | | | | | | | | |
| 3 | $1.3 \times 10^{10}$ | 5.41 | 0.49 | 0.78 | 0.54 | 0.93 | 6267568 | 2.21 | 1.63 | 16.22 | 15.95 |
| 5 | $3.8 \times 10^{10}$ | 37.62 | 0.55 | 1.42 | 0.84 | 1.75 | 23307704 | 2.36 | 1.72 | 16.91 | 16.55 |
| 8 | $1.0 \times 10^{11}$ | 316.77 | 0.67 | 2.99 | 0.93 | 4.47 | 74024440 | 2.55 | 1.89 | 17.67 | 16.93 |
| **Aloha network protocol (Aloha)** $N$ = no. of nodes in the network | | | | | | | | | | | |
| 40 | $2.3 \times 10^{13}$ | 2.69 | 0.79 | 0.71 | 1.55 | 0.80 | 15879556 | 1.52 | 1.44 | 12.14 | 11.91 |
| 70 | $4.3 \times 10^{22}$ | 22.20 | 0.86 | 0.84 | 1.69 | 0.89 | 82907316 | 1.66 | 1.63 | 13.22 | 12.90 |
| 100 | $6.5 \times 10^{31}$ | 66.28 | 0.87 | 0.85 | 1.74 | 0.91 | 239179076 | 1.78 | 1.75 | 14.38 | 14.33 |
| **Randomised leader election protocol (Leader)** $N$ = no. of processors | | | | | | | | | | | |
| 6 | $1.9 \times 10^{6}$ | 3.72 | 0.48 | 0.71 | 0.89 | 0.97 | 2422704 | 2.81 | 2.01 | 14.34 | 13.71 |
| 7 | $2.4 \times 10^{7}$ | 24.34 | 0.44 | 0.65 | 0.81 | 1.1 | 7063232 | 3.29 | 2.17 | 15.89 | 14.43 |
| 8 | $3.0 \times 10^{8}$ | 128.08 | 0.43 | 0.63 | 0.82 | 1.24 | 16107968 | 3.61 | 2.52 | 16.70 | 15.61 |
| **Bounded open queueing network (BQ)** $N$ = no. of customers | | | | | | | | | | | |
| 30 | $2.4 \times 10^{8}$ | 2.1 | 0.36 | 0.41 | 0.82 | 0.90 | 2241036 | 2.08 | 1.96 | 18.66 | 17.59 |
| 50 | $4.6 \times 10^{9}$ | 24.25 | 0.39 | 0.44 | 0.85 | 0.91 | 15112996 | 2.14 | 2.05 | 18.93 | 17.92 |
| 70 | $3.3\times 10^{10}$ | 146.01 | 0.41 | 0.45 | 0.87 | 0.94 | 54895356 | 2.50 | 2.23 | 19.20 | 18.36 |
| **Dining philosophers (Philosophers)** $N$ = philosophers, phil./level = 6 | | | | | | | | | | | |
| 20 | $3.5 \times 10^{12}$ | 14.82 | 1.12 | N/A | 1.26 | N/A | 569608 | 1.82 | N/A | 14.07 | N/A |
| 40 | $1.2 \times 10^{25}$ | 33.32 | 1.13 | N/A | 1.32 | N/A | 1097560 | 1.96 | N/A | 14.40 | N/A |
| 80 | $1.4 \times 10^{50}$ | 77.35 | 1.19 | N/A | 1.35 | N/A | 2321768 | 2.28 | N/A | 14.93 | N/A |

that the efficient load balancing and scheduling of Cilk is superior to our thread pool in exploiting parallelism where parallelism exists. In comparison, Cilk is able to obtain a speedup for seven of the models instead of three for the thread pool. Where models can be parallelised, the larger the size of the model, the greater the parallelism. The models of particular interest due to their comparatively high speedups against the other models, are the Queens, FMS, and RSM models. On a four processor 2.4GHz Intel Operon machine, the speedups for these models increase to over 3 for the Queens and FMS models and over 5 for the RSM model, demonstrating that all of the cores are being utilised. The RSM model exhibits a superlinear speedup due to the combined effects of chaining and the effective parallelisation using Cilk.

Chaining is practically effective in improving the run-time on both the RSM and Leader models, although it may conceptually hinder run-time due to the synchronisation overhead from managing access and updates to the chaining graphs. This is because chaining can also decrease the overall amount of work by finding an event ordering that leads to firing fewer events. This effect also leads to the use of less memory across all of the models, as shown in the *memory increase* column indicating the relative increase in memory for the parallel algorithms against the sequential version. The column also shows that the Cilk algorithms require significantly more memory than the sequential algorithm, due to the size of the Cilk stack. The only model where Cilk requires less memory than the thread pool algorithm when using chaining is the slotted ring model. For this model, chaining helps the thread pool algorithm, halving the memory used by the non-chained version. Overall, however, the thread pool algorithm significantly outperforms the Cilk algorithm regarding memory, which is due to the fact that it does not have to allocate memory for a waiting stack.

Our results show that Cilk is more effective in exploiting parallelism than our hand-crafted thread pool algorithm, but incurs a significant memory overhead due to its lack of support for pipelining. This is a relevant and timely observation due to the increasing popularity of multi-core machines. However, the scalability of our algorithm across a larger number of processors (or processor cores) requires further study in order to fully understand the impact of the synchronisation overhead introduced by each processor (core). We leave this to future work.

## 5 Related Work

Research on symbolic model checking has primarily focused on networks of workstations (NOWs) [2, 11, 12, 13, 17, 19], using message-passing libraries to communicate between workstations. None of the existing approaches uses a parallel language to facilitate scheduling and load balancing; approaches to dealing with these overheads are implemented by hand. Also, most work on parallel state-space generation considers how to parallelise the underlying data structure. These approaches target the increased memory available on a NOW

by slicing data structures and distributing them across processors. The structure of decision diagrams has previously been sliced horizontally [2] and vertically [13, 17, 19]. Horizontal slicing scales well but prevents significant speedups, since each slice has to complete its work before the next slice can begin.

Grumberg, Heyman, Ifergan, and Schuster [11] parallelised symbolic state-space generation algorithms to gain speedups by developing vertical slices on different processors of a NOW. If the algorithm controlling the slices has to frequently synchronise on the application of the next-state function, each round of computation is only as fast as the slowest time it takes for a slice to develop on a processor. To achieve speedups, the parallel algorithm allows slices to develop asynchronously while the next-state function is applied to create more work. The algorithm is load-balanced using workstealing techniques implemented by hand [12]. For very large circuits, these techniques can lead to efficient parallelisation, showing up to an order of magnitude improvement in time-efficiency.

Our approach is unique in that we consider how to functionally decompose the Saturation algorithm rather than its data structures. In comparing a proven efficient parallel language to our own hand-crafted approach, we determined how efficient both implementations are in terms of run-time and memory. We expect that our observations can be extrapolated to PC clusters when utilising distributed shared-memory (DSM) techniques.

## 6 Conclusions

We investigated whether the parallel language Cilk could improve the efficiency of a parallel variant of the MDD-based Saturation algorithm for computing reachable state spaces of asynchronous systems on shared-memory architectures, such as modern multi-processor multi-core PCs. Our experimental studies showed that Cilk is much more effective than a hand-crafted implementation for addressing load balancing and scheduling. However, while the usage of Cilk led to considerable improvements in time-efficiency, the restrictions imposed by the Cilk language implied an enormous memory overhead.

The results from running our hand-crafted solution demonstrated that preventing idle functions from inhabiting the stack removes this memory overhead. Pipelining is therefore an essential feature of any language for parallelising symbolic state-space generators. To the best of our knowledge, there is currently no parallel language fitting this description. However, a possible future direction of parallel irregular languages extending the Cilk model of multithreaded computation to include pipelining is proposed in [20]. This would enable the truly efficient parallelisation of symbolic state-space generators, thereby making significant progress in utilising parallel architectures in automated verification.

*Acknowledgements.* We wish to thank the anonymous reviewers for their detailed comments and suggestions.

## References


[1] Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. In: PPOPP, pp. 207–216. ACM, New York (1995)

[2] Chung, M.-Y., Ciardo, G.: Saturation NOW. QEST, pp. 272–281. IEEE (2004)

[3] Chung, M.-Y., Ciardo, G., Yu, A.J.: A fine-grained density-guided chaining heuristic for symbolic reachability analysis. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 51–66. Springer, Heidelberg (2006)

[4] Ciardo, G., Jones, R.L., Miner, A.S., Siminiceanu, R.: Logical and Stochastic Modeling with SMART. Performance Evaluation 63, 578–608 (2006)

[5] Ciardo, G., Lüttgen, G., Siminiceanu, R.: Saturation: An efficient iteration strategy for symbolic state-space generation. In: TACAS, vol. 2031, pp. 328–348. LNCS (2001) An extended version is to appear in the FMSD journal

[6] Ciardo, G., Marmorstein, R., Siminiceanu, R.: Saturation unbound. In: Garavel, H., Hatcliff, J. (eds.) ETAPS 2003 and TACAS 2003. LNCS, vol. 2619, pp. 379–393. Springer, Heidelberg (2003)

[7] Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NuSMV: A new symbolic model checker. STTT 2(4), 410–425 (2000)

[8] Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)

[9] Ezekiel, J., Lüttgen, G., Siminiceanu, R.: Can Saturation be parallelised? In: Brim, L., Haverkort, B., Leucker, M., van de Pol, J. (eds.) FMICS 2006 and PDMC 2006. LNCS, vol. 4346, Springer, Heidelberg (2007)

[10] Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: SIGPLAN, pp. 212–223. ACM, New York (1998)

[11] Grumberg, O., Heyman, T., Ifergan, N., Schuster, A.: Achieving speedups in distributed symbolic reachability analysis through asynchronous computation. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 129–145. Springer, Heidelberg (2005)

[12] Grumberg, O., Heyman, T., Schuster, A.: A work-efficient distributed algorithm for reachability analysis. FMSD 29(2), 157–175 (2006)

[13] Heyman, T., Geist, D., Grumberg, O., Schuster, A.: Achieving scalability in parallel reachability analysis of very large circuits. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 20–35. Springer, Heidelberg (2000)

[14] Kam, T., Villa, T., Brayton, R., S-Vincentelli, A.L.: Multi-valued decision diagrams: Theory and applications. Multiple-Valued Logic 4(1-2), 9–62 (1998)

[15] Lewis, B., Berg, D.: Multithreaded Programming with Pthreads. Prentice-Hall, Englewood Cliffs (1998)

[16] McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers, Dordrecht (1993)

[17] Milvang-Jensen, K., Hu, A.J.: BDDNOW: A parallel BDD package. In: Gopalakrishnan, G.C., Windley, P. (eds.) FMCAD 1998. LNCS, vol. 1522, pp. 501–507. Springer, Heidelberg (1998)

[18] Siminiceanu, R., Ciardo, G.: Formal verification of the NASA runway safety monitor. STTT 9(1), 63–76 (2007)

[19] Stornetta, T., Brewer, F.: Implementation of an efficient parallel BDD package. In: DAC, pp. 641–644. ACM, New York (1996)

[20] Yong, X., Wen-Jing, H.: Aligned multithreaded computations and their scheduling with performance guarantees. Par. Proc. Let. 13(3), 353–364 (2003)