AN ANTICIPATED FIRING SATURATION ALGORITHM FOR SHARED-MEMORY ARCHITECTURES

Jonathan Ezekiel*, Gerald Lüttgen† and Radu I. Siminiceanu ‡

ABSTRACT

Parallelising symbolic state-space generation algorithms, such as Saturation, is known to be difficult as it often incurs high parallel overheads. To improve efficiency, related work on a distributed-memory implementation of Saturation proposed using idle processors for speculatively firing events and caching the obtained results, in the hope that these results will be needed lateron.

This paper investigates a variant of this anticipated firing approach for shared-memory architectures, such as multi-core PCs. Rather than parallelising Saturation, the idea is to run the sequential Saturation algorithm on one core, while the others are given speculative work. Since computing the optimal strategy for selecting useful work is likely to be an NP-complete problem, the paper devises and implements various heuristics. The obtained experimental results show that moderate speed-ups can be achieved as a result of using anticipated firing. However, the proposed heuristics require further work in order to be truly useful in practice.

1 INTRODUCTION

Automated verification, such as temporal-logic model checking [13], 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 practice [12, 23]. Several efforts have been made to implement these algorithms on parallel computer platforms, most notably on networks of workstations and PC clusters [18, 19, 20, 24, 28]. The efforts range from simple approaches that essentially implement BDDs as two-tiered hash tables [24, 28], to sophisticated approaches relying on *slicing* BDDs [19], to techniques for *workstealing* [18]. However, the resulting implementations typically show limited speed-ups, which is not surprising given that state-space generation is essentially an irregular task.

Saturation [10], as implemented in the verification tool SMART [7], is a symbolic state-space generation algorithm with unique features. It is intended for event-based asynchronous system models that are based on interleaving semantics, and exploits the local effect of firing events on state vectors by locally manipulating *Multi-valued Decision Diagrams* (MDDs) [22], which are a generalised version of BDDs. Saturation has proved orders of magnitude more time- and memory-efficient than other symbolic algorithms [8], including the one implemented in the popular NuSMV model checker [12]. Recently, the local effect of firing events has been exploited to parallelise Saturation on shared-memory architectures, in particular on multi-core PCs [15, 16]. Unfortunately, several overheads are incurred from the parallelisation, such as for scheduling and synchronisation, which can prevent run-time gains.

*Department pf Computer Science, Imperial College London, U.K.; email: jezekiel@doc.ic.ac.uk.
†Department of Computer Science, University of York, U.K.; email: luettgen@cs.york.ac.uk.
‡National Institute of Aerospace (NIA), Hampton, Virginia, U.S.A.; email: radu@nianet.org.

An alternate approach to parallelisation is *anticipated firing* [4, 5], a method for speeding up Saturation by utilising idle processor cores on a parallel architecture to perform useful work during the state-space generation process. Anticipated firing was first introduced and used for a distributed version of Saturation on a Network of Workstations (NOW) [3]. In the distributed setting, a workstation may frequently be idle while it waits for another workstation to complete its work. During this idle time, a workstation can perform predictive work on the state space that it may need to carry out at some point in the future, and store the result of such work in caches. From an efficiency perspective, this can only have a positive impact on the run time of the algorithm, since it does not interfere with the state-space generation task. Hence, the question arises as to whether anticipated firing can be used for a shared-memory version of Saturation to perform useful work for speeding up the sequential state-space generation process without introducing high parallel overheads.

This paper investigates anticipated firing for our Saturation algorithm on shared-memory architectures. We begin by recalling the Saturation algorithm (cf. Sec. 2), before introducing the concept of anticipated firing and investigating the potential, i.e., theoretical, savings when using an optimal strategy for selecting useful work (cf. Sec. 3). We then implement an anticipated firing Saturation algorithm on a shared-memory architecture, namely a multi-core PC, and provide experimental results for the algorithm's performance on a benchmark of models (cf. Sec. 4). Finally, we draw conclusions from both the results of our optimal strategy investigation and the algorithm's actual performance of on a benchmark (cf. Sec. 5).

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* $\mathcal{S} \subseteq \widehat{\mathcal{S}}$, i.e., 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 $\mathcal{N}(\mathcal{X}) = \bigcup_{\mathbf{i} \in \mathcal{X}} \mathcal{N}(\mathbf{i})$.

2.1 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 [11]. Without loss of generality, we further 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* [10], 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.

2.2 Saturation-based iteration strategy.

In addition to efficiently representing $\mathcal{N}$, the Kronecker encoding allows us to recognise *event locality* and employ *Saturation* [10]. 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 [10].

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: *Saturate* and *Fire*. Function *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, called *unique table*, and caching results to previous calls of *Fire* in a *firing cache*. The pseudo code for the Saturation algorithm and supporting functions is shown in App. A.

Experimental results reported in [6, 10, 11] 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. However, the optimal and irregular nature of the algorithm also means that it is difficult to parallelise [15, 14]. Thus, approaches

such as anticipated firing which exploit idle cores, while avoiding the parallelisation overheads associated with irregularity, are desirable for parallelising Saturation.

3 ANTICIPATED FIRING

Anticipated firing is an approach to parallelisation that utilises idle cores to predictively perform work that will likely be required lateron.

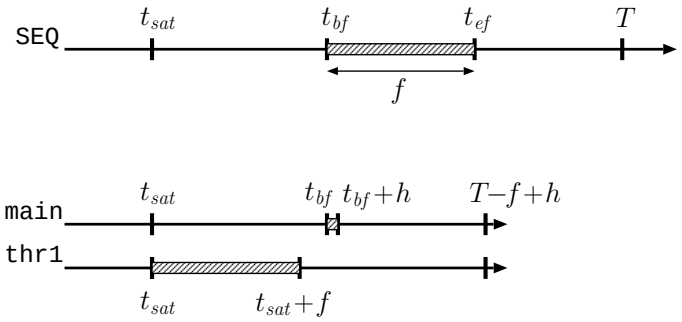

Figure 1: *An example of anticipated firing.*

Fig. 1 illustrates a typical example when anticipation can save execution time for our Saturation algorithm. The top graph shows the relevant timepoints in a sequential execution. If a node is saturated at time t_{sat} and later receives a request to fire some event at time t_{bf}, this firing task can potentially be executed by another thread running in parallel, immediately after the node has been saturated. The result of the firing is cached; therefore, when the request for it occurs on the main thread, the firing operation, which takes f time units, is replaced by a cache lookup, taking h time units. Note that for the result to be ready in time, the duration of the call f needs to satisfy

$$t_{sat} + f < t_{bf} \, .$$

Under most circumstances, h is negligible, in any case much smaller than f, whence the new run time of $T - f + h$ shows a potential savings of $f - h$. Usually, there are numerous fire calls that can be anticipated during a sequential run, and in an ideal case the savings are compounded. A well designed heuristic should correctly anticipate a large portion of these calls, while minimising the inevitable overheads from running multi-threaded applications, i.e., data racing, spin locks, etc. We next look at the practical upper bounds of applying the anticipation technique.

3.1 Perfect knowledge assumption

To evaluate the potential for run-time savings of the heuristics in a shared-memory environment, we look at the hypothetical scenario of a perfect knowledge, i.e., we assume that every event firing origination timepoint is known a priori. To conduct the analysis, we collect all the timepoints that are involved in the firings of events in saturated nodes, from a sequential run of Saturation, and store them in a list of events. A list item includes:

- The node p and event e in the fire request;

- The time of p's saturation, $t_{sat}(p)$;
- The time when the fire procedure starts $t_{bf}(p, e)$;
- The time of completion of the firing $t_{ef}(p, e)$

The duration of each call is then $t_{dur}(p, e) = t_{ef}(p, e) - t_{bf}(p, e)$.

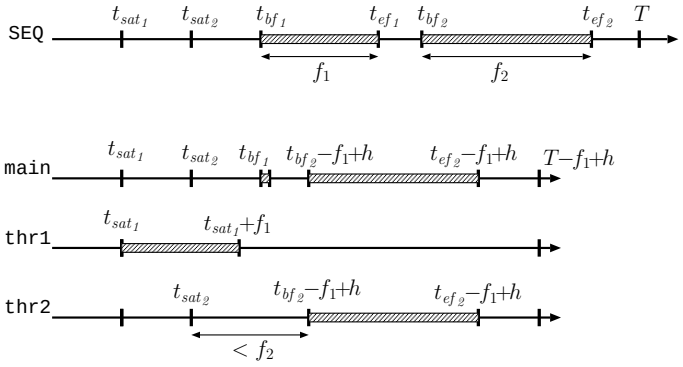

Figure 2: *An example of one anticipation call disabling another.*

A naive analysis would simply add the lengths of the anticipated calls and report that as the maximum run-time savings. This type of analysis, however, does not take into account the snowballing effect of anticipating calls with respect to reducing the window of opportunity for subsequent anticipated calls. Consider the scenario in Fig. 2, where two calls — call them $\mathcal{F}_1$ and $\mathcal{F}_2$, with their corresponding timepoints — are both initially valid for anticipation. If call $\mathcal{F}_1$ is anticipated, the interval $[t_{bf_1}, t_{ef_1}]$ is collapsed. Then each subsequent timepoint in the list needs to be updated as follows, to reflect the new scheduled time:

- All timepoints in the interval $[t_{bf_1}, t_{ef_1}]$ move back by $t_{bf_1} - t_{sat_1}$, since their new scheduled time is on a new thread that starts earlier at t_{sat_1};
- Timepoints after t_{ef_1} move back by f_1.

A collateral result of this collapse is that a later call, for example call $\mathcal{F}_2$, may no longer be anticipated. This happens in case the interval from t_{sat_2} to the new timepoint for the fire request, $t_{bf_2} - f_1 + h$, has become smaller than the necessary f_2 units to complete the call and have the result ready.

A C program (see App. B) computes the new, "realistic" savings due to the collapsing effect. The strategy adopted in the analysis is to anticipate calls strictly in the order in which they arrive, i.e., in a first come first served manner, and then rule out the calls that become disabled. This is an iterative process that updates the list and potentially eliminates some of the entries. Other strategies can be thought of, such as collapsing the intervals in a different order, beginning with the *longest* interval first and updating the list items that follow.

It is easy to see, however, that either one of these strategies might miss the optimal savings. A counterexample for the first-call-first strategy is an instance of the scenario in Fig. 2, where $f_2 > f_1$. A counterexample for the best-call-first strategy (not illustrated) is

Table 1: *Theoretical results for the anticipated firing heuristic*

N	total	naive	first
Slotted Ring Network Protocol			
40	77.55	57.13	29.10
60	81.61	77.20	35.93
80	83.50	79.44	41.37
100	84.22	81.05	42.79
Round Robin Mutex model			
150	72.51	33.89	30.22
200	78.62	33.92	30.93
250	81.85	37.76	35.01
300	85.07	38.77	36.70
Kanban model			
25	41.34	15.48	13.85
30	41.74	15.80	14.46
35	41.69	16.34	15.53
40	41.67	16.54	15.98
Queens model			
9	71.91	61.25	40.00
10	85.34	73.18	58.88
11	92.36	78.04	65.67
12	96.44	81.23	66.78
Leader Election model			
5	79.61	70.15	55.17
6	85.82	83.47	62.37
7	90.67	90.05	66.29
8	94.07	72.06	54.84
FMS model			
40	65.67	18.21	12.14
60	75.43	27.18	20.17
80	80.84	33.44	25.91
100	84.47	35.77	26.04

a scenario with four calls of lengths, f_1, f_2, f_3 and f_4, with f_2 the longest. If $\mathcal{F}_2$ disables both $\mathcal{F}_3$ and $\mathcal{F}_4$ and if $f_2 < f_3 + f_4$, this strategy is not optimal. In general, we believe that computing the optimal strategy of choosing which calls to anticipate, is an NP-complete problem which is most likely reducible from one of the standard scheduling problems.

3.2 Results

Table 1 shows three values for the computed savings on a series of parameterised models. The first column lists the model parameter (size), the second is the sum of all fire call times on saturated nodes, the third eliminates the unfeasible calls due to the condition $t_{sat} + f < t_{bf}$, while the fourth shows the "realistic" savings for the first-call-first strategy.

When considering parallelising Saturation using the anticipated firing approach, the savings indicate that the approach is potentially worthwhile for implementation on a shared-memory architecture. However, we must also consider that a practical implementation is unlikely to improve on the savings indicated in the fourth column.

```
AnticipatedFire(in k:lvl,p:idx)
Fire events on p, a node at level k from above k, in anticipation of future work.
declare e: event;
  1.  foreach e ∈ E^{k+1} do
  2.      RecFire(e, k, p, false);

RecFire(in e:event,l:lvl,q:idx,CT:bool):idx
Build an MDD rooted at s, a node at level l, in UT[l], encoding N*_{≤l}(N_e(B(q))).
Return s.
declare L:set of lcl;
declare f,u,s,r:idx;
declare i,j:lcl;
declare sCng:bool;
  1. if l < Last(e) then return q;
  2. if Find(FC[l], {q, e}, s) then return s;
  3. s ⇐ NewNode(l);
  4. sCng ⇐ false;
  5. l ⇐ Locals(e, l, q);
  6.  while L ≠ ∅ do
  7.    i ⇐ Pick(L);
  8.    f ⇐ RecFire(e, l−1, q[i], CT);
  9.    if f ≠ 0 then
 10.      foreach j ∈ N^l_e(i) do
 11.        u ⇐ Union(l−1, f, s[j]);
 12.        if u ≠ s[j] then
 13.          s[j] ⇐ u; sCng ⇐ true;
 14. if sCng then Saturate(l, s);
 15. r ⇐ Check(l, s);
 16. if CT and r = s then AddTask(l, s);
 17. Insert(FC[l], {q, e}, s);
 18. return s;

ThreadLoop()
If there are no items in the task queue sleep until woken up. Otherwise remove the
head item (k, p) from the task queue, and call AnticipatedFire(k, p).
```

Figure 3: *Pseudo-code for the anticipated firing algorithm.*

4 SHARED-MEMORY ANTICIPATED FIRING

As demonstrated in the previous section, the anticipated firing approach can be used to make savings by utilising idle cores to perform useful work. In practice, anticipated firing can be employed to parallelise Saturation by using idle cores to fire events on newly saturated nodes. If an event e is fired on a node p and results in a new node q, then q can be checked into the unique table, and the pair $\{e, q\}$ can be put into the firing cache, so that q can be retrieved if e is fired on p again at some point in the future. In the following, we adopt the anticipated firing idea to parallelise Saturation on a shared-memory architecture. To do so, we use the thread pool of the Parallel Saturation algorithm in [16] to allow anticipated work to be created and scheduled as tasks, while a main thread constructs the state space and creates anticipated firing tasks. Then, all we are required to decide is when tasks should be created and how they should be dealt with, while leaving the state-space generation task to perform its work without interruption.

We present a naive algorithm for anticipated firing in Fig. 3. The function *AnticipatedFire* carries out firings on a given node p at level k for events at level $k+1$, by calling *RecFire*. The creation of anticipated firing tasks is performed by *RecFire*, which adds a task for anticipated firing as soon as node s becomes saturated at line 16, by calling *AddTask*. We do not wish to initiate anticipated firings on nodes that are created as a result of other anticipated firings for now, as this may lead to a large amount of unnecessary work. Thus, we include a boolean flag called create task (CT) which indicates whether *RecFire* should create an anticipated firing task after a node has become saturated, and *AnticipatedFire* calls *RecFire* with CT set to false. Although we do not show it in our pseudo-code, any function call to *RecFire* from functions being executed by the main thread set CT to true. We also do not wish to perform anticipated firing on nodes that have previously been checked into the unique table, as they are more likely to have events fired upon them.

For our thread pool, the *ThreadLoop* function allows a thread to pick an anticipated firing task and carry it out by calling *AnticipatedFire*. We still have a one-to-one thread to processor allocation [16], but we allocate one thread as the *main* thread, which does not synchronise on the task queue and is left uninterrupted to saturate the root node. We do not show locks in our pseudo-code, but as with our thread pool implementation we lock the operation caches and the unique table on a per level basis for atomic access. This is the only part of the anticipated firing that can interfere with the state-space generation task.

4.1 Selecting events

As soon as a node has become saturated, anticipated firings are performed on that node to carry out work that may be used in the future. Key to the efficiency of the anticipated firing approach is selecting tasks and events that will carry out useful work. While we cannot accurately predict whether an event will carry out useful work, there are a number of heuristics that we can employ in our algorithm which are potentially useful for selecting appropriate events. These are based on the following observations:

- The number of events we wish to fire on a node is significant. If we fire a large number of events on a node, we potentially tie up a processor working on that node. It may be better to fire less events on that node, since the anticipated firing will complete quickly. This reduces the chance that the main thread will begin firing events on the node before the anticipated task has completed. Firing less events also frees up a thread quickly to perform newly created anticipated firing tasks.

- Instead of firing events from one level above the node selected for anticipated firing, we may wish to fire events from several levels above this node. This is potentially beneficial since there is less chance that the main thread will start firing the same events on the node as the anticipated task, before the anticipated firings have completed.

- We may wish to discard anticipated firing tasks for saturated nodes that are at lower levels of the MDD, since the closer to the level that the main thread is saturating for, the greater the possibility for new and potentially useful work. For example, by the time the main thread reaches level fifty of the MDD, nodes below level ten are likely to have been fired upon several times, creating more cached work, and less new work than for nodes at higher levels.

We can use these heuristics for selecting anticipation tasks and events with our anticipated firing algorithm in order to try and increase the amount of useful work performed by idle processors.

4.2 Run-time and memory results

The benchmark we selected for our algorithms contains ten models that have been previously used to parallelise Saturation [6, 10, 11]. The models are *parameterised*, i.e., a parameter can be set for the model, typically N, which alters the size of the state space.

- The classic *dining philosophers* (**Philosophers**) [9] protocol is a solution for mutual exclusion problems. N indicates the number of philosophers, and $K = N/2$.
- The flexible manufacturing system (**FMS**) [25] consists of three types of machines that process three types of parts. The parameter N indicates the initial number of each type of parts, and $K = 19$.
- The *slotted ring network protocol* (**Slot**) [26] is a well known model of a local area network. N is the number of processors in the ring, and $K{=}N$.
- The *Kanban manufacturing system* (**Kanban**) [29] contains sixteen stations that process parts. The parameter N sets the initial number of parts within a station. K is the number of stations.
- The *randomised leader election protocol* (**Leader**) [21] solves the problem of designating a processor as leader within a unidirectional ring. The parameter N of our model defines the number of processors in the ring, and $K = 11N$.
- The *round robin mutex exclusion protocol* (**Robin**) [17] is used to control access for a ring of N processors that access a shared resource.
- The classic *queens problem* (**Queens**) models a game that finds a way to position N queens on an $N \times N$ chessboard without the queens attacking each other.
- The *runway safety monitor* (**RSM**) [27] was developed by NASA and Lockheed Martin, as a protocol to detect runway safety incidents. The RSM protocol defines *targets* T on the ground such as ground vehicles on the runway, which have speeds S, and represents the takeoff and landing zone for aircraft as a 3D grid $X \times Y \times Z$, where X is the width of the runway, Y is its length and Z is its height. For our RSM model we fix $T{=}1$ and $S{=}2$. The parameters of the model are X, Y and Z.
- The *Aloha network protocol* (**Aloha**) [1] defines a simple mechanism for transmitting data across a network. The parameter N represents the number of nodes in the network, and $K = N{+}3$.

We implemented our anticipated firing algorithm using C and the POSIX Pthreads library [2]. The machine used for our experiments 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 our benchmark shown in Table 2, and included three heuristics for selecting events.

Table 2: *Parameters and state-space sizes of the benchmark*

Parameter (N)	**State-space size ($\|S\|$)**
Slotted Ring Network Protocol	
150	4.5×10^{158}
Round Robin Network Protocol	
240	9.5×10^{74}
Kanban Manufacturing System	
35	2.5×10^{14}
Flexible Manufacturing System	
14	1.3×10^{10}
Queen Problem	
13	4674890
Randomised Leader Election Protocol	
8	3.0×10^{8}
Bounded Open Queueing Network	
70	3.3×10^{10}
Dining Philosophers	
80	1.4×10^{50}
Runway Safety Monitor	
832	1.0×10^{11}
Aloha Network Protocol	
100	6.5×10^{31}

- The events fired, *EVF*, is the number of events fired on a node by an anticipated firing task.
- The moving minimum, *MM*, is the minimum of number of MDD levels between the node considered for anticipated firing, and the node that is being saturated by the main thread.
- The firing level *FL* is the number of MDD levels above the node considered for anticipated firing, that events will be fired for.

We present the results for our benchmark models in Figs. 4 and 5, showing the different heuristic settings we tried. For the default settings, *EVF* is 1, *MM* is unset and *FL* is 1. We show the run-time differences when employing heuristics by changing *EVF* to 2, *MM* to 3 and *FL* to 3.

For five of the models, a run-time improvement is made by the anticipated firing algorithm over the sequential Saturation algorithm. The improvements are approximately between 5% and 20% over the sequential algorithm. Slowdowns are relatively small, with the largest approximately 10% less than the sequential algorithm. Typically, the largest increase in speed is observed between cores 1 and 2, while cores 3 and 4 tend to make little difference to the run-time. This suggests that relatively small amounts of useful work are performed on these cores, or that not enough anticipation tasks are created. On one core, the algorithm demonstrates a slowdown across all models. This is due to the instrumentation of the Pthreads library, and the code overhead from mutex locks that causes a small parallel overhead. Any further

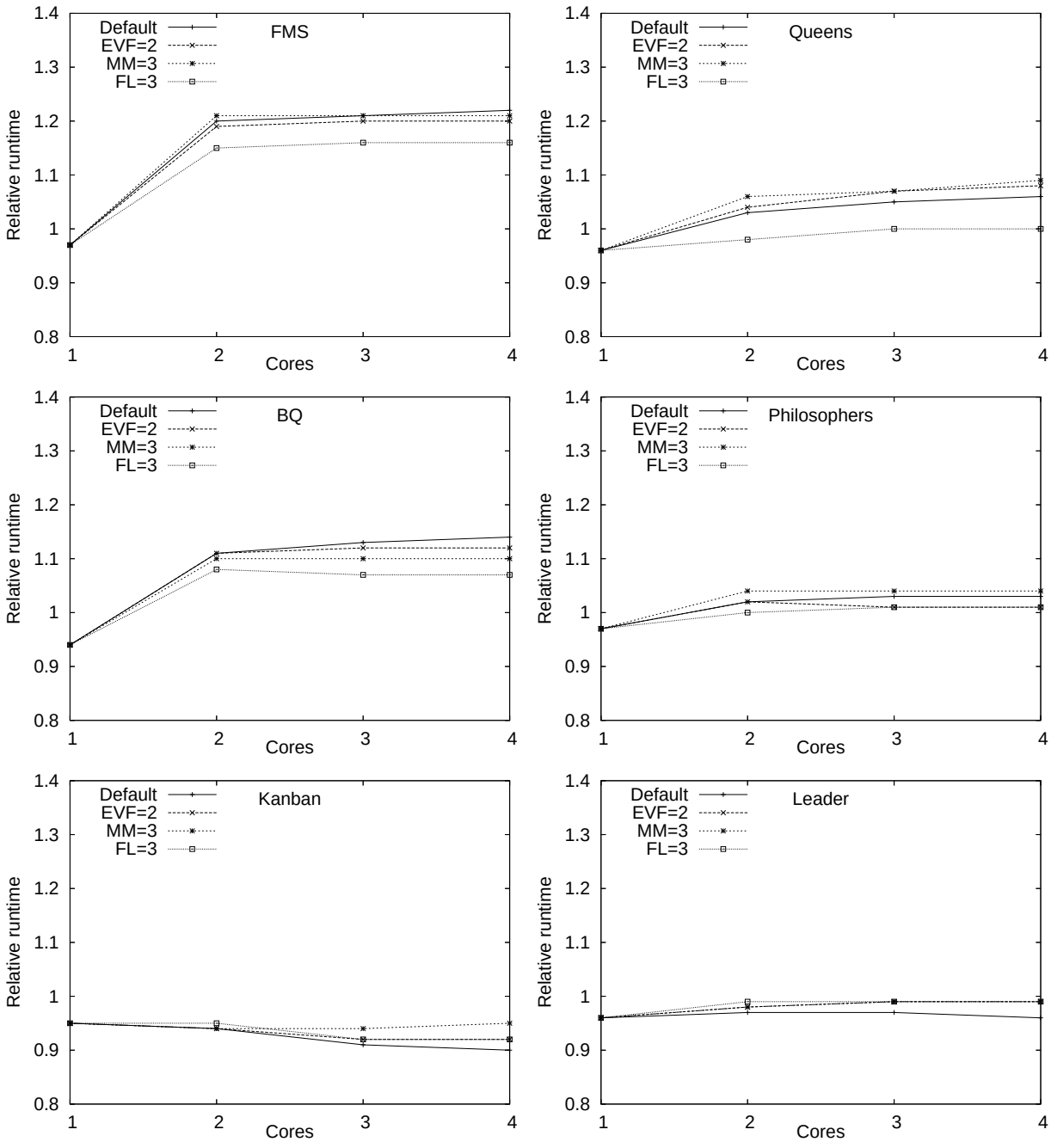

Figure 4: *Run-time results for the anticipated firing algorithm (1).*

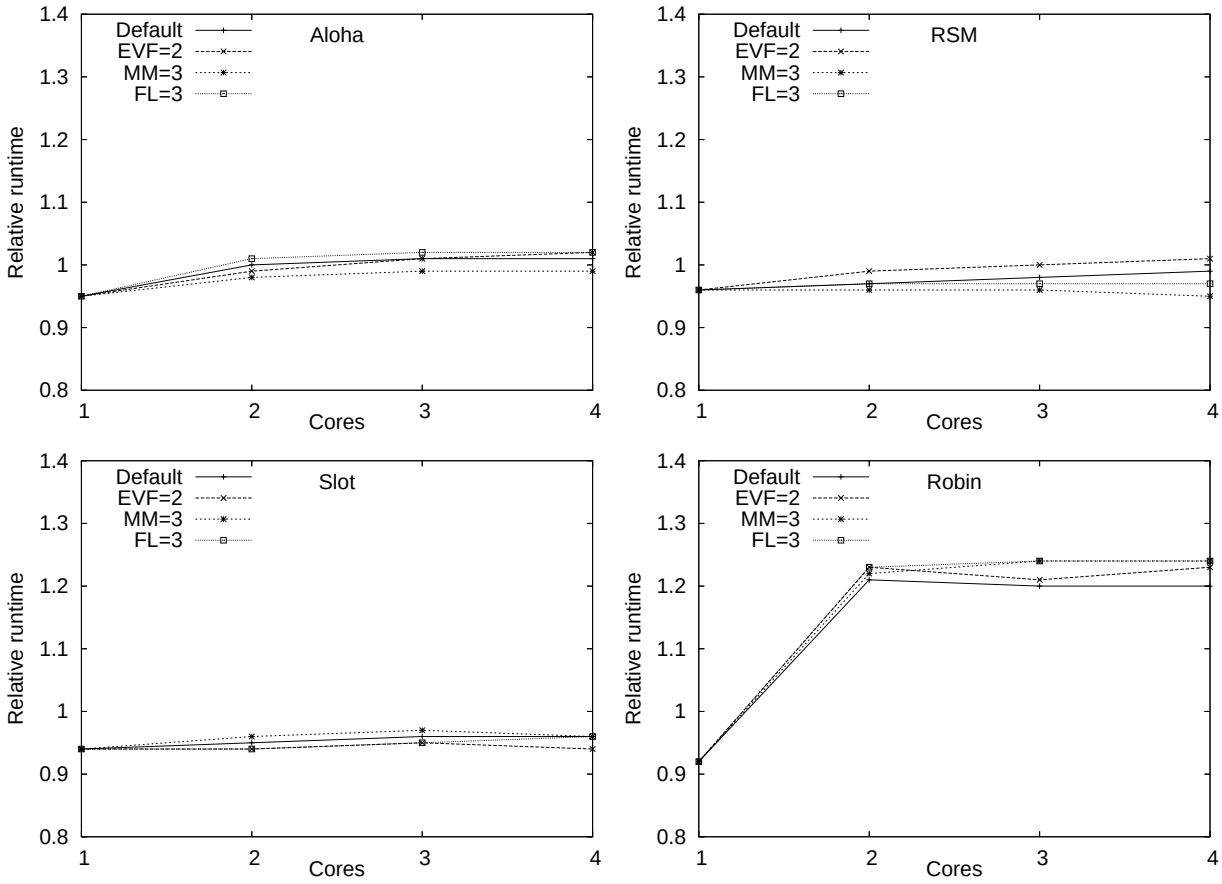

Figure 5: *Run-time results for the anticipated firing algorithm (2).*

slowdowns on the other cores are caused by synchronisation, since this is the only way in which the threads used for anticipated firing can interfere with the main thread. The use of heuristics makes a small and unpredictable difference to the run-time across all models, where the largest difference in run-time for employing a heuristic is around 5%. This would suggest that more complex factors influence the amount of useful work that can be performed.

Our results suggest that anticipated firing is useful for facilitating small run-time improvements on several models without significantly impacting on the state-space generation task. This is due to the use of idle processors without interfering with the main task of saturating the root node. Greater run-time improvements are not possible by employing heuristics for creating tasks and selecting events based on the measures we defined.

5 CONCLUSIONS

We investigated an anticipated firing Saturation algorithm on a shared-memory architecture, namely on multi-core PCs. We began by examining the potential savings that could be made by employing the method, noting that the optimal strategy of choosing which firing calls to anticipate is likely to be an NP-complete problem. We then implemented the anticipated firing algorithm using a thread pool to schedule anticipated work on idle cores.

The results from running the algorithm on our benchmark showed that using anticipated firing with Saturation can facilitate small run-time improvements on several benchmark models without incurring high parallel overheads, but employing simple heuristics for selecting events was unable to improve either the run-time or scalability of the algorithm. Thus, a more optimal strategy for selecting firing tasks is required for achieving greater run-time improvements, which we will leave for future work.

Acknowledgements. We wish to thank Gianfranco Ciardo for several fruitful discussions on the topic of anticipated firing.

REFERENCES

[1] Abramson, N. and Kuo, F. The Aloha system. *Computer Networks*, pp. 501–518, 1973.

[2] Butenhof, D. R. *Programming with POSIX threads*. Addison-Wesley, 1997.

[3] Chung, M.-Y. and Ciardo, G. Saturation NOW. In *QEST*, pp. 272–281. IEEE, 2004.

[4] Chung, M.-Y. and Ciardo, G. A dynamic firing speculation to speedup distributed symbolic state-space generation. In *IPDPS*. IEEE, 2006.

[5] Chung, M.-Y. and Ciardo, G. A pattern recognition approach for speculative firing prediction in distributed saturation state-space generation. In *PDMC*, vol. 135(2) of *ENTCS*, pp. 65–80, 2006.

[6] Chung, M.-Y., Ciardo, G., and Yu, A. J. A fine-grained fullness-guided chaining heuristic for symbolic reachability analysis. In *ATVA*, vol. 4218 of *LNCS*, pp. 51–66. Springer, 2006.

[7] Ciardo, G., Jones, R., Miner, A., and Siminiceanu, R. Logical and stochastic modeling with SMART. *Performance Evaluation*, 63:578–608, 2006.

[8] Ciardo, G., Lüttgen, G., and Miner, A. Exploiting interleaving semantics in symbolic state-space generation. *Formal Methods in System Design*, 31(1):63–100, 2007.

[9] Ciardo, G., Lüttgen, G., and Siminiceanu, R. Efficient symbolic state–space construction for asynchronous systems. In *ICATPN*, vol. 1839 of *LNCS*, pp. 103–122. Springer, 2000.

[10] Ciardo, G., Lüttgen, G., and Siminiceanu, R. Saturation: An efficient iteration strategy for symbolic state-space generation. In *TACAS*, vol. 2031 of *LNCS*, pp. 328–348. Springer, 2001.

[11] Ciardo, G., Marmorstein, R. M., and Siminiceanu, R. Saturation unbound. In *TACAS*, vol. 2619 of *LNCS*, pp. 379–393. Springer, 2003.

[12] Cimatti, A., Clarke, E. M., Giunchiglia, F., and Roveri, M. NUSMV: A new symbolic model checker. *Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[13] Clarke, E., Grumberg, O., and Peled, D. *Model Checking*. MIT Press, 1999.

[14] Ezekiel, J. and Lüttgen, G. Measuring and evaluating parallel state-space exploration algorithms. In *PDMC*, ENTCS (To Appear), 2007.

[15] Ezekiel, J., Lüttgen, G., and Ciardo, G. Parallelising symbolic state-space generators. In *CAV*, vol. 4590 of *LNCS*, pp. 268–280. Springer, 2007.

[16] Ezekiel, J., Lüttgen, G., and Siminiceanu, R. Can Saturation be parallelised? On the parallelisation of a symbolic state-space generator. In *PDMC*, vol. 4346 of *LNCS*, pp. 331–346. Springer, 2007.

[17] Graf, S., Steffen, B., and Lüttgen, G. Compositional minimisation of finite state systems using interface specifications. *Formal Aspects of Computing*, 8(5):607–616, 1996.

[18] Grumberg, O., Heyman, T., Ifergan, N., and Schuster, A. Achieving speedups in distributed symbolic reachability analysis through asynchronous computation. In *CHARME*, vol. 3725 of *LNCS*, pp. 129–145. Springer, 2005.

[19] Grumberg, O., Heyman, T., and Schuster, A. A work-efficient distributed algorithm for reachability analysis. *Formal Methods in System Design*, 29(2):157–175, 2006.

[20] Heyman, T., Geist, D., Grumberg, O., and Schuster, A. Achieving scalability in parallel reachability analysis of very large circuits. In *CAV*, vol. 1855 of *LNCS*, pp. 20–35. Springer, 2000.

[21] Itai, A. and Rodeh, M. Symmetry breaking in distributed networks. *Information and Computation*, 88(1):60–87, 1990.

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

[23] McMillan, K. *Symbolic Model Checking*. Kluwer, 1993.

[24] Milvang-Jensen, K. and Hu, A. J. BDDNOW: A parallel BDD package. In *FMCAD*, vol. 1522 of *LNCS*, pp. 501–507. Springer, 1998.

[25] Miner, A. S. and Ciardo, G. Efficient reachability set generation and storage using decision diagrams. In *ICATPN*, vol. 1639 of *LNCS*, pp. 6–25. Springer, 1999.

[26] Pastor, E., Roig, O., Cortadella, J., and Badia, R. M. Petri net analysis using boolean manipulation. In *PNPM*, vol. 815 of *LNCS*, pp. 416–435. Springer, 1994.

[27] Siminiceanu, R. I. and Ciardo, G. Formal verification of the NASA runway safety monitor. *Software Tools for Technology Transfer*, 9(1):63–76, 2007.

[28] Stornetta, T. and Brewer, F. Implementation of an efficient parallel BDD package. In *DAC*, pp. 641–644. ACM, 1996.

[29] Tilgner, M., Takahashi, Y., and Ciardo, G. SNS 1.0: Synchronized network solver. In *ICATPN*, pp. 215–234, 1996.

A PSEUDO-CODE FOR THE SATURATION ALGORITHM

Saturate(in k:*lvl*,p:*idx*)

Update p, a node at level k not in $UT[k]$, in–place, to encode $\mathcal{N}^*_{\leq k}(\mathcal{B}(p))$.

declare e:*event*;
declare $\mathcal{L}$:set of *lcl*;
declare f,u:*idx*;
declare i,j:*lcl*;
declare $pCng$:*bool*;

1. repeat
2. $pCng \Leftarrow false$;
3. foreach $e \in \mathcal{E}^k$ do
4. $\mathcal{L} \Leftarrow Locals(e,k,p)$;
5. while $\mathcal{L} \neq \emptyset$ do
6. $i \Leftarrow Pick(\mathcal{L})$;
7. $f \Leftarrow RecFire(e,k-1,p[i])$;
8. if $f \neq 0$ then
9. foreach $j \in \mathcal{N}^k_e(i)$ do
10. $u \Leftarrow Union(k-1,f,p[j])$;
11. if $u \neq p[j]$ then
12. $p[j] \Leftarrow u$; $pCng \Leftarrow true$;
13. if $\mathcal{N}^k_e(j) \neq \emptyset$ then
14. $\mathcal{L} \Leftarrow \mathcal{L} \cup \{j\}$;
15. until $pCng = false$;

RecFire(in e:*event*,l:*lvl*,q:*idx*):*idx*

Build an MDD rooted at s, a node at level l, in $UT[l]$, encoding $\mathcal{N}^*_{\leq l}(\mathcal{N}_e(\mathcal{B}(q)))$. Return s.

declare $\mathcal{L}$:set of *lcl*;
declare f,u,s:*idx*;
declare i,j:*lcl*;
declare $sCng$:*bool*;

1. if $l < Last(e)$ then return q;
2. if $Find(FC[l],\{q,e\},s)$ then return s;
3. $s \Leftarrow NewNode(l)$;
4. $sCng \Leftarrow false$;
5. $l \Leftarrow Locals(e,l,q)$;
6. while $\mathcal{L} \neq \emptyset$ do
7. $i \Leftarrow Pick(\mathcal{L})$;
8. $f \Leftarrow RecFire(e,l-1,q[i])$;
9. if $f \neq 0$ then
10. foreach $j \in \mathcal{N}^l_e(i)$ do
11. $u \Leftarrow Union(l-1,f,s[j])$;
12. if $u \neq s[j]$ then
13. $s[j] \Leftarrow u$; $sCng \Leftarrow true$;
14. if $sCng$ then $Saturate(l,s)$;
15. $Check(l,s)$; $Insert(FC[l],\{q,e\},s)$;
16. return s;

Generate(in s:array[1..K] of lcl):idx

Build an MDD rooted at r, a node at level K, encoding $\mathcal{N}^*_{\mathcal{E}}(s)$ and return r, in $UT[K]$.

declare r,p:idx;
declare k:lvl;

1. $p \Leftarrow \mathbf{1}$;
2. for $k = 1$ to K do
3. $\quad r \Leftarrow NewNode(k)$; $r[s[k]] \Leftarrow p$;
4. $\quad Saturate(k, r)$; $Check(k, r)$;
5. $\quad p \Leftarrow r$;
6. return r;

Union(in k:lvl, p:idx, q:idx):idx

Build an MDD rooted at s, in $UT[k]$, encoding the Union of the nodes p and q at level k. Return s.

declare i:lcl;
declare s,u:idx;

1. if $p = \mathbf{1}$ or $q = \mathbf{1}$ then return $\mathbf{1}$;
2. if $q = \mathbf{0}$ or $p = q$ then return q;
3. if $p = \mathbf{0}$ then return p;
4. if $Find(UC[k], \{p, q\}, s)$ then return s;
5. $s \Leftarrow NewNode(k)$;
6. for $i = 0$ to n^k - 1 do
7. $\quad u \Leftarrow Union(k-1, p[i], q[i])$;
8. $\quad s[i] \Leftarrow u$;
9. $Check(k, s)$;
10. $Insert(UC[k], \{p, q\}, s)$;
11. return s;

Find(in tab, key, out v, sat:$bool$):$bool$

If (key, x, y) is in hash table tab, set v to x and sat to y and return $true$. Else, return $false$.

Insert(inout tab, in key, v, sat:$bool$)

If key is not $(\mathbf{0}, \mathbf{0})$ insert (key, v, sat) in hash table tab, if it does not contain an entry $(key, \cdot, true)$.

Locals(in e:$evnt$, k:lvl, p:idx):set of lcl

Return all of the local states in p locally enabling e. If there are no states in p locally enabling e then return $\emptyset$.

Pick(inout $\mathcal{L}$:set of lcl):lcl

Remove and return an element from $\mathcal{L}$.

NewNode(in k:lvl):idx

Create p, a node at level k with arcs set to $\mathbf{0}$. Return p.

Check(in k:lvl, inout p:idx)

If p, a node at level k not in $UT[k]$, duplicates q, in $UT[k]$, delete p and set p to q. Else, insert p in $UT[k]$. If $p[0] = \cdots = p[n^k-1] = \mathbf{0}$ or $\mathbf{1}$, delete p and set p to $\mathbf{0}$ or $\mathbf{1}$.

B C CODE FOR COMPUTING IDEAL RUN-TIME SAVINGS

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define SAT  0
#define FIRE 1
int runtime;

struct list_item {
  int li_type;
  int level;
  int index;
  int event;
  int sat_time;
  int end_time;
  int dur;
  list_item *next;
  list_item(int, int, int, int, int, int, int, list_item *);
};

list_item::list_item(int w, int k, int p, int e,
           int st, int et, int d, list_item *n) {
  li_type = w;
  level = k;
  index = p;
  event = e;
  sat_time = st;
  end_time = et;
  dur = d;
  next = n;
}

struct list {
  list_item *head;
  list_item *tail;
  list();
  void load();
  void add(list_item *);
  void sort();
  int process();
  int savings();
};

list::list() {
  head = tail = NULL;
}
```

```
void list::add(list_item *t) {
  if (head == NULL)
    head = tail = t;
  else {
    tail->next = t;
    tail = tail->next;
  }
}


void list::load() {
  char s[1000], *tok[20];
  while (!feof(stdin)) {
    fgets(s, 1000, stdin);
    if (strstr(s, "ANTICIPATE") !=NULL) {
      tok[0] = strtok(s, "(");
      tok[1] = strtok(NULL, ",");
      tok[2] = strtok(NULL, ", e");
      tok[3] = strtok(NULL, ")");
      tok[4] = strtok(NULL, " ");       // done
      tok[5] = strtok(NULL, " ");       // at
      tok[6] = strtok(NULL, " ,");      // time
      tok[7] = strtok(NULL, " :");      // duration
      tok[8] = strtok(NULL, ", <");     // dur value
      tok[9] = strtok(NULL, "<,");      // empty
      tok[10] = strtok(NULL, "<,");     // lev
      tok[11] = strtok(NULL, " >");     // idx
      tok[12] = strtok(NULL, " ");      // saturated
      tok[13] = strtok(NULL, " ");      // at
      tok[14] = strtok(NULL, " ");      // sat_time
      int flv = atoi(tok[1]);
      int fid = atoi(tok[2]);
      int fev = atoi(tok[3]+2);
      int fet = int(atof(tok[6]));
      int fdr = atoi(tok[8]);
      int slv = atoi(tok[10]);
      int sid = atoi(tok[11]);
      int stm = int(atof(tok[14]));
      list_item *its = new list_item(SAT,  slv, sid,   0, stm, stm,   0, NULL);
      list_item *itf = new list_item(FIRE, flv, fid, fev, stm, fet, fdr, NULL);
      add(its);
      add(itf);
    }
```

```
    else if (strstr(s, "al time for s") !=NULL) {
      tok[0] = strtok(s, " ");
      for (int i=1; i<8; i++) tok[i] = strtok(NULL, " ");
      double rt = atof(tok[7]);
      runtime = (int) (1000000.0*rt);
    }
  }
}



int collapse(list_item *l, int stm, int fbt, int fet) {
  int jump = fbt - stm; // interval between end of saturate to begin firing
  int durf = fet - fbt;
  // - every event falling within the interval [begin_fire .. end_fire]
  //   is moved back by adv units
  // - every event after end_fire
  //   is moved back by fet-fbt (= duration of fire) units
  list_item *p = l;
  while (p!=NULL) {
    int begin_fire = p->end_time - p->dur;
    if (begin_fire >= fbt && begin_fire <= fet) {
      p->end_time -= jump;
    }
    else if (begin_fire > fet) {
      p->end_time -= durf;
    }
    // also adjust saturation time for each item
    if (p->sat_time >= fbt && p->sat_time <= fet) {
      p->sat_time -= jump;
    }
    else if (p->sat_time > fet) {
      p->sat_time -= durf;
    }
    p = p->next;
  }
  return durf;
}
```

```
int list::process() {
  int savings = 0;
  for (list_item *p = head; p != NULL; p = p->next) {
    if (p->li_type==FIRE) {
      int stm = p->sat_time;
      int fet = p->end_time;
      int fbt = fet - p->dur;
      if (stm + p->dur < fbt) {
        printf("\n\nAnticipating FIRE(%d,%d, e%d):
         moved from %d to %d\n", p->level, p->index, p->event, fbt, stm);
        savings += collapse(p, stm, fbt, fet);
      }
      else {
        printf("\n\nFIRE(%d,%d, e%d) can no longer be anticipated.\n",
                p->level, p->index, p->event);
      }
    }
  }
  return savings;
}

int list::savings() {
  int savings = 0;
  for (list_item *p = head; p != NULL; p = p->next) {
    if (p->li_type==FIRE) {
      savings += p->dur;
    }
  }
  return savings;
}

int main(int argc, char **argv) {
  list *l = new list;
  l->load();
  int ideal_savings = l->savings();
  l->sort();
  int real_savings = l->process();
  printf("\n\nRelative savings: %d/%d (%6.2f\%)\n", real_savings, ideal_savings,
          ((double)(100.0*real_savings))/((double)ideal_savings));
  printf("Total runtime: %d ms\n", runtime);
  printf("\nIdeal savings:      %d/%d (%6.2f\%)\n", ideal_savings, runtime,
          ((double)(100.0*ideal_savings))/((double)runtime));
  printf("\nComputable savings: %d/%d (%6.2f\%)\n", real_savings, runtime,
          ((double)(100.0*real_savings))/((double)runtime));
  return 0;
}
```