Appears in *Proceedings of AAAI-97*

Fast Context Switching in Real-time Propositional Reasoning

P. Pandurang Nayak and Brian C. Williams
Recom Technologies, NASA Ames Research Center
Mail Stop, 269-2
Moffett Field, CA 94035.
{nayak,williams}@ptolemy.arc.nasa.gov

Abstract

The trend to increasingly capable and affordable control processors has generated an explosion of embedded real-time gadgets that serve almost every function imaginable. The daunting task of programming these gadgets is greatly alleviated with real-time deductive engines that perform all execution and monitoring functions from a single core model. Fast response times are achieved using an incremental propositional deductive database (an LTMS). Ideally the cost of an LTMS's incremental update should be linear in the number of labels that change between successive contexts. Unfortunately an LTMS can expend a significant percentage of its time working on labels that remain constant between contexts. This is caused by the LTMS's conservative approach: a context switch first removes *all* consequences of deleted clauses, whether or not those consequences hold in the new context. This paper presents a more aggressive incremental TMS, called the ITMS, that avoids processing a significant number of these consequences that are unchanged. Our empirical evaluation for spacecraft control shows that the overhead of processing unchanged consequences can be reduced by a factor of seven.

Introduction

The unending trend towards cheap processing has generated an explosion of embedded real-time gadgets. Developing robust real-time kernels for these gadgets often require codes that provide a variety of system level tasks such as commanding, monitoring, diagnosis, recovery, and safe shutdown. In (Williams & Nayak 1996) we introduced an embedded real-time execution kernel, called Livingstone, that performs all these functions automatically using a single model of the underlying hardware. To achieve the stringent demands of realtime performance Livingstone reduces each function to a deductive search problem on a propositional database. This search must be completed before the system moves to the next state, with required response times on the order of hundreds of milliseconds. Hence the success of Livingstone's model-based execution paradigm hinges critically upon the efficiency of the propositional deductive database.

A major time saving can be achieved by adopting an event driven approach, propagating the effects to the database as sensor readings and states change. A truth maintenance system (Doyle 1979), in particular the LTMS (McAllester 1980), offers a natural starting point. The LTMS incrementally maintains the deductive closure of unit propagation on a propositional clausal theory as clauses are added and deleted. While our use of an LTMS in Livingtone has been exceedingly favorable, the stringent performance requirements of real time leaves room for improvement. In this paper we present an extension to the LTMS that demonstrates substantial performance improvement.

The best an update algorithm can achieve is for the cost of an LTMS update to be linear in the number of labels that change between successive contexts. Unfortunately an LTMS can expend a significant percentage of its time working on labels that remain constant between contexts. For example, on a realworld spacecraft control problem this overhead was 37% on average and rose to about 670% in the worst case. The source of this added cost is the LTMS's conservative approach to guaranteeing well-founded (*i.e.*, loop-free) support: a context switch first removes *all* consequences of any deleted clauses, whether or not those consequences hold in the new context, prior to propagating with added clauses.

This paper presents a more aggressive incremental TMS, called the ITMS, that avoids processing a significant number of unchanged consequences. The ITMS algorithm is based on two properties. First, it exploits the properties of depth-first numbering to immediately find alternate supports for propositions while guaranteeing well-foundedness. Second, the ITMS provides a novel mechanism for propagating the consequences of

C_1: $\neg nc_i \vee \neg a \vee nc_o$ C_4: $\neg rf \vee ia$ C_7: $\neg ok \vee \neg uf$
C_2: $\neg ia \vee nc_o$ C_5: $\neg uf \vee ia$ C_8: $\neg rf \vee \neg uf$
C_3: $\neg ok \vee a$ C_6: $\neg ok \vee \neg rf$ C_9: $\neg a \vee \neg ia$
where

- ok: bus is operating normally
- rf: bus is recoverably failed
- uf: bus is unrecoverably failed
- a: bus is active
- ia: bus is inactive
- nc_i: no command input to bus
- nc_o: no command output from the bus

Figure 1: A small fragment of the theory describing DS-1. C_1–C_2 specify conditions under which the bus outputs no command. C_3–C_5 define conditions under which the bus is active or inactive. C_6–C_9 are mutual exclusion clauses.

newly added clauses *before* other clauses are deleted, increasing the number of consequences available to be used as alternate support. The improvement is dramatic. On the spacecraft problem the average performance of the ITMS is merely 5% off ideal with a worst case overhead of 100%. This is approximately a factor of seven reduction of overhead over the LTMS. The next section summarizes the traditional LTMS in a nutshell, the following section presents the ITMS, and the paper closes with empirical results and related work.

Background

This section introduces some basic terminology as used in this paper. An LTMS manipulates a set Σ of propositional clauses over a set of boolean propositions. (A clause is a disjuction of literals, where each literal is either a proposition or the negation of a proposition.) An LTMS labeling assigns a label (*true*, *false*, or *unknown*) to each proposition. When the LTMS assigns the label *true* (*false*) to a proposition p, it guarantees that Σ logically entails p ($\neg p$).[2] Given a labeling, a clause C is a unit clause if the label of exactly one literal in C is *unknown* and all other literals are *false*. C is a conflict if all literals in C are *false*.

The LTMS uses unit propagation (also called boolean constraint propagation) to compute proposition labels. Given a labeling, the basic step in unit propagation is to non-deterministically select a unit clause and change the label of the *unknown* literal in the clause to *true*. A terminal labeling is one which either has no unit clauses or has a conflict. The LTMS always computes a terminal labeling.

In computing a terminal labeling, the LTMS also constructs a dependency structure that explains why a proposition has a given label. The dependency structure is derived in the natural way from a proposition's support, which is the clause used by unit propagation to infer a truth value for that proposition. Incremental clause deletion uses the dependency structure to undo all (and only the) propagations that depend on the deleted clause. A proposition p has a well-founded support if and only if the above dependency graph has no cycles containing p.

Incremental Truth Maintenance

Context switching during combinatorial search usually involves simultaneous addition and deletion of clauses because context switches correspond to changing assignments, *e.g.*, model-based diagnosis algorithms change context by changing the mode assigned to a component. Implementing this context switch as a delete clause followed by an add clause is sub-optimal since the LTMS is unable to preserve propagations that hold both before and after the context switch but that do not hold in the intermediate context.

Example 1 Figure 1 shows part of the theory describing the bus controller of Deep Space One (DS-1), the first of NASA's New Millennium spacecraft (Pell *et al.* 1997). All commands from the flight computer to the spacecraft's actuators are routed through the bus. The clauses in Figure 1 show part of a typical command. The clauses state that the bus outputs no command to a specific actuator if either the bus is active and the flight computer is not sending any command or the bus is inactive. Versions of these clauses are repeated for each command type for each actuator.

Assuming that the bus is operating normally (clause $C_{10}: ok$) and that we can infer that there is no input command (nc_i), Figure 2 shows the generated LTMS labels and supports. Suppose now that a problem-solver wants to change the context and assume that the bus is recoverably failed (rf). This is achieved by first deleting clause C_{10} and then adding clause $C_{11}: rf$. Deleting C_{10} undoes the propagation to nc_o and to all propositions dependent on nc_o.[3] Subsequent addition of C_{11} resupports nc_o with clause C_2 (via ia and rf), relabels it *true*, and restores propagations to nc_o's consequences. The point is that nc_o's label and propagations to its consequences are preserved across the context switch, but are not preserved in the intermediate context, leading to excessive repropagation. In the

[2]Previous descriptions of an LTMS introduce the notion of premises or assumptions. For simplicity, we simply represent premises as additional clauses in Σ, with no loss of functionality.

[3]For simplicity, consequences of nc_o have been omitted from this example and from Figures 1 and 2.

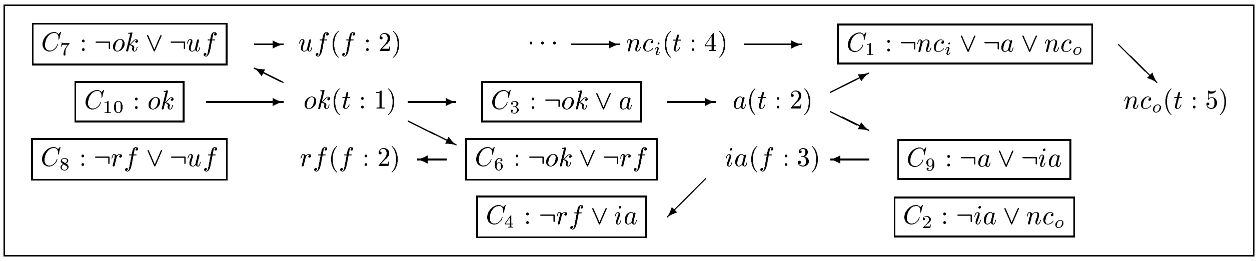

Figure 2: LTMS labels and proposition supports. Arrows have been drawn from clauses to propositions they support, and from propositions to clauses in which the literal occurrence is *false*. Parenthesized expressions specify the proposition labels and propagation numbers (introduced in the subsection on resupporting a proposition).

complete version of this example the supports of over 300 propositions are lost in the intermediate context, only to be restored in the final context.

The critical drawback of the LTMS algorithm is that it is overly conservative, leading it to undo nc_o's label without first looking for ways to *resupport* it. Resupporting propositions during a clause deletion is complicated by the fact that the resupported proposition must be provided with a well-founded support. Resupporting nc_o is further complicated by the fact that it has no resupport without the addition of C_{11}. This suggests that C_{11} should be added *before* C_{10} is deleted, with propagations from C_{11} being used to resupport nc_o. Unfortunately, propagation faces a barrier: C_{11} and C_{10} are mutually inconsistent, so that C_{11} is a conflict in the labeling shown in Figure 2. The conflict is a barrier to propagation, and breaking through this barrier requires an algorithm for *propagating through a conflict*. In the rest of this section we develop the ITMS algorithm which provides a fast context switching algorithm based on the above ideas.

Context switch algorithm

We start by describing the ITMS algorithm for context switching. Two key subroutines for resupporting propositions and propagating through conflicts are described in the subsequent two subsections.

Let A be the added clause and D be the deleted clause. If D supports no proposition in the current labeling, then deleting it leaves the labeling unchanged. Hence, the ITMS algorithm is identical to the LTMS algorithm. Now suppose that D supports a proposition d. Following the above discussion, the ITMS algorithm starts by adding A to Σ and initiates propagation. If the resulting terminal labeling contains no conflicts, then the ITMS algorithm merely deletes D from Σ using the standard deletion algorithm and propagates to a terminal labeling. However, when the LTMS is used in combinatorial search, propagation with both A and D in Σ leads to conflicts, since context switches usually correspond to assignment changes and assignments are required to be unique (*e.g.*, a component can have exactly one mode).

When unit propagation leads to a conflict, the ITMS tries to propagate through the conflict using the algorithm described in the next subsection. Propagating through a conflict C changes the label of a proposition p occurring in C such that (a) p satisfies C; and (b) C provides a well-founded support for p. Since C became a conflict as a result of adding A and propagating, p's new label depends on A. The change in p's label has three important side-effects. First, p's new label may allow us to use the algorithm in the next subsection to resupport a different proposition q, making q dependent on p and hence on A. Second, propagations based on p's old label are undone (unless resupport is possible). Third, p's old support, and possibly other clauses, become conflicts. The ITMS then recursively picks another conflict and tries to propagate through it, until no more propagations are possible. At this point the ITMS deletes D, and propagates to a terminal labeling.

Example 2 Adding C_{11} to Σ results in a conflict as shown on the left side of Figure 3. Propagating through C_{11} is achieved by changing rf's label to *true*. As a result of this change, uf can be resupported using C_8, and C_4 and C_6 become conflicts, as shown on the right side of Figure 3. The ITMS then propagates through C_4, changing the label of ia to *true*. As a result of this change, nc_o is resupported by C_2, as desired. nc_o's resupport and the propagation through C_4 are discussed in detail in Examples 3 and 4.

The ITMS enforces the following conditions on the conflict C picked for propagation and the proposition p whose label is changed by the propagation:

1. d does not occur in C. If d occurs in C, p will lose its support (C) when D is deleted from Σ. This is undesirable since we want p's label and support to be preserved in the new context. Ideally, p's new label

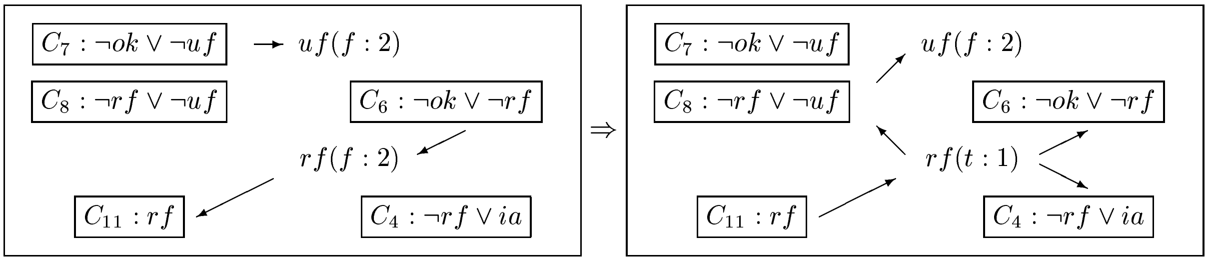

Figure 3: Adding C_{11} to the ITMS results in a conflict as shown on the left. Propagating through C_{11} yields the labeling on the right.

should be independent of d. However, guaranteeing this is expensive since it may involve a complete traversal of the dependency graph. Hence, we use the approximation that d does not occur in C.

2. p's label has not been changed while propagating through another conflict, thus preventing cycles.

Resupporting a proposition

Consider a proposition p supported by a clause C in Σ. A clause R can resupport p if and only if all the following conditions hold:

1. If p occurs positively (negatively) in C then it occurs positively (negatively) in R. This ensures that resupporting with R preserves p's label.

2. All other literals in R are *false*. This ensures that R can propagate a label to p.

3. None of the other literals in R depend on p. This ensures that resupporting p with R provides p with a well-founded support.

The first and second conditions are easy to check. However, condition 3 is potentially time consuming. The straightforward implementation that traces back over proposition supports takes time comparable to the time for a complete repropagation, defeating the very purpose of incremental context switching.

We address this difficulty by using a fast approximation that is sufficient for condition 3 to hold. The key idea is to associate a *propagation number* with each supported proposition that satisfies the following invariant:

- If a proposition is supported by a clause S, then it's propagation number is *greater than* the propagation number of all other propositions occurring in S.

A proposition's propagation number is set whenever unit propagation provides it with with a support. In this case we set the propagation number to be 1 more than the largest of the propagation numbers of the other propositions occurring in the support. If there are no other propositions occurring in the support, the propagation number is set to 1.

The above invariant ensures that a proposition's propagation number is *less than* the propagation numbers of all its consequence. Hence, it follows that if p's propagation number is *greater than or equal to* q's propagation number, then q *cannot* depend on p., *i.e.*, condition 3 holds if the old propagation number of p is greater than or equal to the propagation number of all other propositions occurring in R. Hence, we replace condition 3 with:

$3'$. The prior propagation number of p (when it was supported by C) is greater than the propagation numbers of all the other literals in R.

Condition $3'$ is easy to check, so that it is easy to check if a clause R can resupport a proposition p. Two points are worth highlighting. First, condition $3'$ is sufficient but not necessary for condition 3: if p's propagation number is less than q's propagation number it does not mean that q depends on p. Hence, this algorithm may miss resupport opportunities.

Second, condition $3'$ requires that the prior propagation number of p is *not* equal to the largest propagation number of other propositions in R, even though condition 3 is satisfied when equality holds. Equality is excluded because we need to set the propagation number of p following its resupport with R. The new propagation number of p can be any value greater than the propagation numbers of other propositions in R, but less than or equal to the old propagation number of p. The latter is required to ensure that the propagation number of p continues to be less than the propagation number of all propositions dependent on p. If equality were allowed, we would be forced to increase p's propagation number, potentially violating this restriction.

Example 3 Figure 4 shows the situation before and after nc_o is resupported. In the situation on the left, nc_o is supported by clause C_1. Clause C_2 can resupport nc_o since (a) nc_o occurs positively in both C_1 and

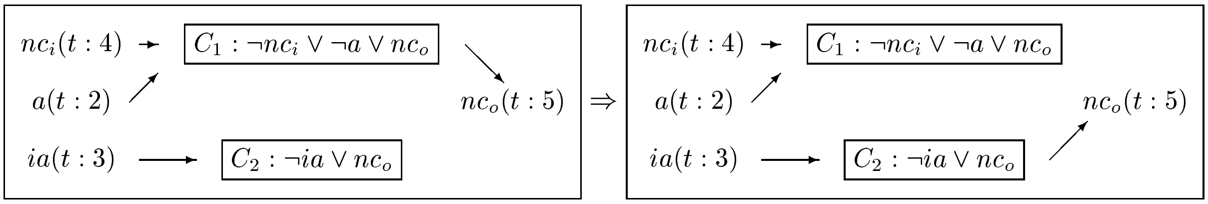

Figure 4: Proposition nc_o can be resupported by clause C_2. The left and right hand sides of the figure show the situation before and after resupport, respectively.

C_2; (b) the other literal in C_2 ($\neg ia$) is *false*; and (c) the propagation number of nc_o on the left is greater than the propagation number of ia. Hence, nc_o can be resupported by clause C_2 as shown on the right.

Propagating through a conflict

We now develop the algorithm to propagate through a conflict C to a proposition p. Since we want C to provide a well-founded support for p, none of the other propositions occurring in C should depend on p. Following the discussion in the previous subsection, we require that:

- In the current labeling, the propagation number of p is greater than or equal to the propagation number of all other propositions occurring in C.

This is sufficient to ensure that none of the other propositions in C depend on p. As before, this is not a necessary condition, but rather a fast approximation. Note that, unlike condition $3'$ in the previous subsection, the above condition includes the case where p's propagation number is equal to the propagation number of some other propositions in C. The reason for this will become clear shortly.

Let us now say that p satisfies the above condition, and we wish to propagate to p through C. We change the current labeling using the following three steps:

1. Change the label of p from *true* to *false* or vice versa, and let C be p's support. The new propagation number of p can be any value greater than the propagation number of all other propositions occurring in C.

2. Resupport any propositions that can be made dependent on p with its new label, using the algorithm in the previous subsection with the added condition that p must occur in the clause used for resupport.

3. Undo any propagations based on p having its previous label. Since we undo all propagations based on p's previous label, p's new propagation number is not required to be less than or equal to its previous propagation number. Hence, in selecting p from the conflict C, we can include the propositions whose propagation numbers are greater than *or equal to* the propagation numbers of other propositions in C.

Example 4 Figure 5 shows the situation before and after propagating through the conflict C_4. On the left, clause C_4 is a conflict, and ia is the proposition occurring in C_4 with the largest propagation number. Hence, we propagate through conflict C_4 to ia. On the right, C_4 supports ia and ia's label has been changed to *true*. This change makes clause C_9 a conflict. When ia becomes *true*, proposition nc_o is resupported as shown in Figure 4.

Discussion

The central invariant associated with propagation numbers, namely that a proposition's propagation number is greater than the propagation number of propositions it depends on, guarantees that the above algorithms for resupporting propositions and propagating through conflicts yield well-founded supports. When taken together with the fact that the ITMS always concludes a context switch by propagating to a terminal labeling, this means that the ITMS implementation of a context switch is sound and complete with respect to unit propagation.

Experimental results

We now present an empirical evaluation of our implementation of the ITMS algorithm. This evaluation compares the ITMS algorithm against the LTMS algorithm that first deletes and then adds the clauses. The comparison is done in two ways. First, we compare the number of operations required by each algorithm. For the LTMS algorithm, the number of operations is the number of times a proposition's label is changed (from either *true* or *false* to *unknown* or vice versa). For the ITMS algorithm we also include the number of times a proposition's label is changed from *true* to *false* or vice versa (which is also the number of conflicts that are propagated). Second, we compare the number of propositions whose labels are changed by the two algorithms against the number of propositions whose labels

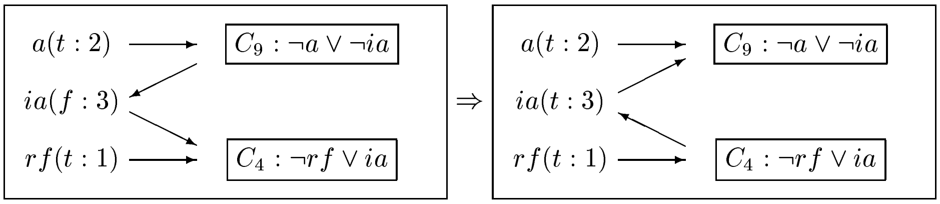

Figure 5: Satisfying a conflict. Clause C_4 is a conflict on the left, and is satisfied by proposition ia on the right.

must change across the context switch. This provides us with a measure of the extra work done by the two algorithms.

The evaluation was done on the propositional theory used in on-board real-time model-based diagnosis and recovery for the DS-1 spacecraft (Williams & Nayak 1996; Pell *et al.* 1997). The theory is based on modeling DS-1 using 145 components and an average of almost 4 modes per component, resulting in a total of 3,905 propositions and 12,693 clauses. A context switch in this application corresponds to changing the mode of a component.

We evaluated the algorithms on 387 distinct context switches. For each context switch we calculated the ratio of the number of operations performed by the ITMS to the number performed by the LTMS. Table 1 summarizes the results. It shows the number of context switches that yielded ratios within various intervals.

Three points are worth noting. First, the 8 cases that give improvements of over 80% provide the most compelling argument for using the new algorithm in a real-time system, and was the central motivation for developing the ITMS. The LTMS provides unacceptably slow response times in those 8 cases, compromising the need for timely fault diagnosis and recovery. Second, the ITMS also provides improved performance in all but 4 cases, with an average improvement of about 30%. Third, the ITMS performs worse than the LTMS by about 5% in 4 cases. The reason for this is related to condition 1 in the subsection describing the context switch algorithm, which introduced an approximation for the condition that propagating through a conflict should not depend on the clause to be deleted. When this approximation is violated, the effort in propagating through the conflict is wasted, which explains the performance in the 4 cases.

Table 2 summarizes the comparison between the number of propositions whose labels are changed by the two algorithms against the number of propositions whose labels must change across the context switch. Three points are worth noting. First, in a majority of cases (264 out of 387) the ITMS does not modify the label of any extra propositions. Second, the ITMS performs significantly better than the traditional algorithm on the average and in the worst case. On average, the ITMS modifies only about 5% more propositions than is required, while the LTMS modifies about 37% more propositions. This means that, on average, the ITMS overhead is about seven times smaller than than the LTMS overhead. The 18 worst cases of the LTMS range from about 2.2 to 7.7 times the required number of label changes. Third, the ITMS still performs poorly in a small number of cases (10), where it modifies about twice the number of required propositions. The reason for this is that our resupport algorithm relies on a sufficient, but not necessary, condition to identify resupport opportunities, leading it to miss some opportunities.

Related work

The main drawback of LTMS algorithms, *viz.*, the need to redo propagations that hold across a context switch, has been identified in the past as the so-called *unouting* problem. The main approach to addressing this problem has been to propose a fundamentally different type of truth maintenance system—the ATMS (de Kleer 1986). The advantage of the ATMS is its ability to switch contexts without any label propagation. However, this comes at the cost of an exponential time and space labeling process, making it inapplicable for embedded, real-time systems. This is not surprising since the original ATMS was designed specifically for problems that require finding all solutions, *e.g.*, envisionment. Real-time systems do not have this luxury, instead having to pick a small number of most preferred solutions, *e.g.*, most likely or least cost solutions.

More recently, various ATMS focusing algorithms have been developed to alleviate the exponential cost of labeling by restricting ATMS label propagation to just the current context (Forbus & de Kleer 1988; Dressler & Farquhar 1990). A context switch in such systems can require label propagation, weakening the main advantage of the ATMS and making LTMSs more attractive. Unfortunately, no one has made precise empirical comparisons between problem solvers based on focused ATMSs and those based on LTMSs. However, recent experience with an LTMS-based diagnosis engine on a standard diagnostic suite have been exceed-

Ratio intervals	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0	1.1
Number	2	6	0	0	0	8	211	96	28	32	4

Table 1: Table summarizing the ratio of the number of operations performed by the ITMS to the number of operations performed by the LTMS. The first row specifies the upper bound of intervals of width 0.1, *e.g.*, the entry 0.7 specifies the interval from 0.6 to 0.7. The second row specifies the number of ratios that fall in the corresponding interval, *e.g.*, 211 context switches yielded improvements of 30–40%.

Ratio intervals	= 1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	2.0	2.1	> 2.1
ITMS	264	94	15	4	0	0	0	0	0	0	8	2	0
LTMS	0	23	89	155	78	20	4	0	0	0	0	0	18

Table 2: Table summarizing the ratios of the number of propositions whose labels are modified by each algorithm to the number of propositions whose labels must change across the context switch. The numbers in the first row specify the upper bounds of intervals of length 0.1, except for the first and last numbers which represent ratios exactly equal to 1.0 and greater than 2.1, respectively. The second and third rows specify the number of ratios that fall into each interval for the ITMS and the LTMS, respectively.

ingly favorable, and appear to be comparable to the very best focused ATMS-based engines (Williams & Nayak 1996).

Recent work by de Kleer has focused on making unit propagation complete using prime implicates (de Kleer 1990). This is related to our use of an LTMS as a propositional reasoning engine, but is orthogonal to the topic of this paper. Everett and Forbus (Everett & Forbus 1996) develop a technique to scale up an LTMS via fact garbage collection. The technique makes sense in their application since they use the LTMS as a cache, though it does not make sense in ours since we use the LTMS as a real-time propositional reasoning engine.

Conclusions

This paper describes the ITMS, an agressive incremental TMS that optimizes context switching. The ITMS uses a resupport algorithm based on propagation numbers and a novel algorithm for propagating through conflicts. As a result the ITMS can propagate the consequences of newly added clauses before other clauses are deleted, making these consequences available to be used for alternate supports. This results in a dramatic reduction in overhead compared to a traditional LTMS, specially in worst-case performance, making the ITMS a critical component of Livingstone's embedded real-time execution kernel.

The main areas of improvement in our context switching algorithm are related to developing more complete, and yet efficient, algorithms for detecting resupport opportunities, and finding efficient ways to ensure that propagations through a conflict do not depend on the clause to be deleted. These improvements will help bring the context switch algorithm closer to the ideal.

Acknowledgements

We would like to thank Jim Kurien, Bill Millar, Howie Shrobe, and the anonymous reviewers for helpful discussions and their comments on the paper.

References

de Kleer, J. 1986. An assumption-based TMS. *Artificial Intelligence* 28(1):127–162.

de Kleer, J. 1990. Exploiting locality in a TMS. In *Procs. of AAAI-90*, 264–271.

Doyle, J. 1979. A truth maintenance system. *Artificial Intelligence* 12:231–272.

Dressler, O., and Farquhar, A. 1990. Putting the problem solver back in the driver's seat: Contextual control of the ATMS. In *LNAI 515*. Springer-Verlag.

Everett, J. O., and Forbus, K. D. 1996. Scaling up logic-based truth maintenance systems via fact garbage collection. In *Procs. of AAAI-96*, 614–620.

Forbus, K. D., and de Kleer, J. 1988. Focusing the ATMS. In *Procs. of AAAI-88*, 193–198.

McAllester, D. 1980. An outlook on truth maintenance. Memo 551, MIT AI Laboratory.

Pell, B.; Bernard, D. E.; Chien, S. A.; Gat, E.; Muscettola, N.; Nayak, P. P.; Wagner, M. D.; and Williams, B. C. 1997. An autonomous spacecraft agent prototype. In *Procs. of the First International Conference on Autonomous Agents*. ACM Press.

Williams, B. C., and Nayak, P. P. 1996. A model-based approach to reactive self-configuring systems. In *Procs. of AAAI-96*, 971–978.