Bounded-Overhead Caching for Definite-Clause Theorem Proving

Alberto Segre & Daniel Scharstein
segre@cs.cornell.edu
schar@cs.cornell.edu

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

Abstract

In this paper we describe the design of an effective caching mechanism for resource-limited definite-clause theorem-proving systems. Previous work in adapting caches for theorem proving relies on the use of unlimited-size caches. We show how unlimited-size caches are unsuitable in application contexts where resource-limited theorem provers are used to solve multiple problems from a single problem distribution. We introduce bounded-overhead caches; that is, those caches that contain at most a fixed number of entries and entail a fixed amount of overhead per lookup, and examine cache design issues for bounded-overhead caches. Finally, we present an empirical evaluation of bounded-overhead cache performance, relying on a specially designed experimental methodology that separates hardware-dependent, implementation-dependent, and domain-dependent effects.

1. Introduction

A cache is a device that stores the results of a previous computation so that it might be reused. While the term is usually associated with a hardware device in a hierarchical computer memory system [Hennessy90], software caches also exist. For example, *memoization* — originally introduced as a form of machine learning [Michie68] but often viewed as a program optimization technique [Mostow85] — is one example of a software cache. Regardless of whether a cache is implemented in software or hardware, the same underlying principle is being exploited. Stated precisely, a cache trades off increased storage cost in return for reduced dependency on a slow resource. In the case of a hardware cache, the chip area devoted to the cache is justified by reduced reliance on slower layers of the memory hierarchy. The increased space used by a memoized function to retain associations of function arguments and output values is similarly justified by reduced cost of future computations.

From: *Journal of Automated Reasoning* **11**:1 (1993)

Caches[1] have previously been proposed for use in theorem-proving systems [Dietrich87, Tamaki86, Warren92]. Plaisted [Plaisted88] argues in favor of using *success caches* to store previously proven subgoals. Here the extra cost of maintaining and consulting the cache is balanced against the time required to search for proofs of repeated subgoals. In addition to caching successfully proven subgoals, recording information about failed subgoals has also been suggested as a means of providing additional speedup [Elkan89]. Such *failure caches* record a previously attempted — but as yet unproven — subgoal so that future attempts to prove the subgoal which can be certain *a priori* to fail are not undertaken.

The tradeoff inherent in the use of a cache implies that adding a cache to a theorem-proving system may or may not improve the system's performance. Deciding whether a cache can be useful depends not only on the design of the caching system itself, but the target problem domain, the underlying theorem proving system, and the supporting hardware architecture:

(1) Some problem domains display greater subgoal redundancy and thus are more amenable to caching. Consider, for example, the presence of frame axioms in situation-calculus planning domains [Green69].

(2) Some theorem-proving systems rely on iterative deepening [Korf85] to force completeness. This strategy increases the prevalence of repeated subgoals.

(3) The speed of the underlying hardware's virtual memory system comes into play as the number of cached entries becomes large. A large cache may cause excessive paging.

Because of these factors, it is often difficult to make absolute judgements about the utility of caching for theorem proving.

The message of this paper is that properly designed *bounded-overhead caches*, that is, caches that contain a fixed number of entries and thus entail a fixed amount of overhead per lookup, can improve the performance of definite-clause theorem-proving engines in at least some contexts where more traditional

[1] While the use of the term *cache* varies in the literature, some authors draw distinctions between the related concepts *lemmaizing*, *caching*, and *heuristic caching* [Astrachan91]. The different caches discussed in this paper fall into different categories, thus we use the word only in its most generic sense.

cache designs do not. We motivate the use of bounded-overhead caches for precisely these theorem-proving contexts, and explore a broad range of design and implementation issues associated with such caches. We present an empirical evaluation of bounded-overhead cache performance, relying on a specially designed experimental methodology that separates hardware-dependent, implementation-dependent, and domain-dependent effects. This methodology serves not only to more clearly illustrate the design tradeoffs just mentioned but also to allow us to draw broader conclusions from the experimental data than would be warranted using only elapsed CPU time measurements.

2. Caching for Resource-Limited Definite-Clause Theorem Proving

Definite-clause theorem provers perform search. A typical theorem prover searches the implicit AND/OR tree defined by the domain theory and the query, or goal, under consideration. This is the case with pure PROLOG, where the domain theory corresponds to the PROLOG program. Each OR-node in this implicit AND/OR tree corresponds to a subgoal that must be unified with the head of some matching clause in the domain theory, while each AND-node corresponds to the body of a clause in the domain theory.

Success and failure caches both affect the search at OR-node choice points. In their simplest forms, they serve to prune the search space rooted at the current subgoal. Success caches act as extra database facts, pruning the search process at a previously proven subgoal. Failure caches censor a search at a subgoal which is certain to be fruitless. Failure cache entries may record either an outright failure (*i.e.*, the entire search tree rooted at the subgoal is exhausted without success) or a resource-limited failure (*i.e.*, the search tree rooted at the subgoal is examined unsuccessfully as far as current resource limits, *e.g.*, depth limits, permit). Resource-limited failure cache entries must therefore contain an additional annotation, describing the resource bound below which future attempts to solve the goal are certain to fail.

Caches are used in theorem proving in the hope that they will improve a system's performance, where performance is considered improved if less time is required to solve a problem. Caches affect the performance of a system both by changing the number of nodes explored during search and by changing the time required to explore a single node. Since exploring fewer nodes generally corresponds to better (*e.g.*, faster) performance, we call this a *beneficial search effect* due to caching. Similarly, an increase in nodes explored generally corresponds to slowing down the theorem prover, so we call this an *adverse search effect*. In addition to search effects, cache implementations generally entail a higher per-node exploration cost due to the overhead associated with manipulating and accessing the cache. Every cache implementation entails some of these additional costs; the trick to building a useful cache is to balance the increased node exploration cost — the *cache overhead* — against any beneficial search effects obtained, all the while avoiding adverse search effects.

As an example, consider a simple success cache implementation that adds every successfully-proven subgoal directly to the theorem prover's database as a new axiom. If the problems solved by the theorem prover contain repeated references to identical subgoals, then we can expect this particular cache to exhibit the maximal beneficial search effect. Whether or not this particular cache implementation is effective depends on the interplay between beneficial search effects, adverse search effects, and the cache overhead.

Although — as discussed above — making performance projections *a priori* is simply not possible, we can outline a situation where this simple implementation might not be effective. The argument relies on the fact that exploring a node requires accessing the database: as more nodes are searched, the number of cached entries will inevitably grow, increasing the cost of accessing the database, albeit perhaps slowly. Depending on the balance between the efficiency of the database implementation and the usefulness of the cached entries, at some point the beneficial search effect due to caching may be outweighed by the increased overhead associated with accessing an ever-growing database; an effect that is only more

pronounced if any adverse search effects are encountered. On the other hand, if using the cache provides a relatively large beneficial search effect (for example, an exponentially large reduction with respect to the number of cache entries), even a relatively naive cache implementation that incurs overhead costs proportional to the size of the cache could ultimately provide a performance advantage if offsetting adverse search effects are not encountered. Exactly which side of this tradeoff we are operating on depends to a large extent on the problems being solved; that is, the application context.

2.1. Application Contexts

Key to understanding the tradeoff between search effects and cache overhead is understanding how a particular theorem prover is meant to be used. The goal of most work in the theorem-proving community is to construct provers that are fast and powerful enough to solve difficult problems exactly once. Clearly of interest to this community are problems which were previously not mechanically solvable in a reasonable amount of time. Once a mechanical solution has been obtained, another, typically unrelated, problem is attempted.

We are interested in using our theorem prover in a markedly different manner. Our theorem prover is intended to solve a possibly infinite series of related problems using the same domain theory.[2] The problems are drawn from a fixed — but unknown — problem distribution, and the theorem prover is operating under externally-imposed resource constraints. Each problem is posed along with a resource limit, generally expressed as a search depth limit, a limit on the number of nodes explored, or a CPU time limit. If the theorem prover cannot solve the problem within the specified resource limit, it gives up.[3] We

[2] The theorem prover is a prototype implementation of a component from an autonomous agent architecture. Our *SEPIA intelligent agent* [Segre93b, Segre92b] builds on our previous work in learning and planning [Elkan90, Segre87, Segre88, Segre91a, Segre90, Turney89a, Turney89b]. The goal of the SEPIA project is to build a scalable, adaptive, real-time, agent. Since the SEPIA agent is to be situated in a real world environment, it will face an essentially infinite series of problems posed by interaction with its surroundings.

[3] In practice, all theorem provers are essentially resource limited since we can't wait an eternity for a solution to be found. Of course, the effects of these more informal kinds of resource limits will not be obvious except on the very largest problems.

would like our system to adapt over a series of problems, so that as each problem is solved, the performance of the system on the remaining problems can be expected to improve. Caching is one way of accomplishing this goal; it can be seen as a way of dynamically injecting bias into the system. Indeed, cached subgoals may make subsequent problems solvable, even though they may not have previously been solved within the resource bound.

To make this more concrete, consider using a theorem prover as the core of a trip-planning expert system, such as might be used by a travel agent to plan a client's itinerary. While the domain theory will contain information pertaining to every scheduled flight by any airline, the trips planned by this particular travel agent have a very high probability of starting and ending in the agent's home city. This implies that the series of problems posed (the trip queries) are drawn from the universe of well-formed problems according to an unspecified — but clearly skewed — distribution, since problems drawn from a uniform distribution would have a uniform chance of starting in any city. Our goal is to reduce the solution cost over time for new problems drawn in accordance with this particular distribution by exploiting of this additional, implicit, information about the problem distribution.

Clearly, there are some fundamental differences between these two application contexts. In the former context, caching is useful inasmuch as it may reduce the solution time without compromising correctness or completeness. The number of entries in the cache can only grow as the number of nodes searched in solving the problem. Since clever indexing techniques can make the cost of accessing the cache grow sublinearly with its size, even the simple caching strategy introduced previously may be sufficiently useful in this context. In the latter context, however, we can view the series of related problems presented as effectively one — possibly infinitely large — problem with the theorem prover providing solutions to small portions of the problem as they are posed. Since, as we have argued previously, a cache that grows infinitely large is unlikely to be useful in the limit, an unlimited-size caching implementation will also probably prove wanting in this application context.

2.2. Bounded-Overhead Caching

To make caching practical for our application context, we propose using a fixed-size cache and an affiliated cache management policy. We know that the time required to explore a single node is a function of the branching factor of the search space at that node. Since matching cache entries can be seen as a way of selectively increasing the branching factor at certain nodes, imposing an upper bound on the number of cached entries also bounds the time required to search a single node. Naturally, such a limit on cache size implies that at some point we may have to remove a previously cached entry to make room for a new one.

We can implement this modified caching scheme by separating the cache from the theorem prover's database and adjusting the theorem prover's normal inference procedure. In order to avoid having to determine *a priori* the relative sizes of success and failure caches, we use a single cache where both types of entries coexist (alternatively, one might consider keeping success and failure entries in separate caches as in [Elkan89]). At each OR-node choice point, the theorem prover first checks the cache for a matching success entry, or *cache hit*. If a cache hit occurs, the subgoal is considered solved, even though the solution generated may not be consistent with the rest of the problem and may eventually have to be discarded.[4] If no matching success entry is found, the prover next checks for a matching failure entry. If one is found, the subgoal is considered unsolvable, and the theorem prover is forced to backtrack. If no cache hit occurs, the theorem prover proceeds to to prove the subgoal normally. If it is eventually solved and the solution does not correspond directly to an axiom of the domain theory, a new success entry representing the solved (instantiated) subgoal is inserted in the cache. If instead all branches of the OR-node choice point are exhausted without finding a solution, a new failure entry is inserted into the cache.

[4] Solving sibling subgoals independently may not be sufficient to obtain a proof of the original query, since any constraints that exist between subgoals must also be satisfied. Sibling subgoals require the generation of mutually-consistent variable substitutions; solving a subgoal independently may generate a variable substitution which is not consistent with the rest of the problem.

Once the fixed-size cache is filled, adding a new entry entails deleting an existing cache entry. A *cache-management policy* is used to decide which existing cache entry should be replaced. Cache-management policies are nothing more than heuristics designed to assign relative importance to cache entries. Simple replacement policies such as *first-in-first-out* (FIFO), *least-recently used* (LRU), and *least-frequently used* (LFU) are suggested by analogy with paged memory systems. Empirical studies of memory traces have shown that both programs and data exhibit *locality of reference*; that is, both *temporal locality* (if an item is referenced, it will tend to be referenced again soon) and *spatial locality* (if an item is referenced, nearby items will tend to be referenced as well) [Hennessy90]. The cache-management strategies used for paged memory systems exploit knowledge about memory access patterns. Clearly, theorem-proving systems cannot be expected to exploit locality of reference in exactly the same fashion as paged memory systems. For example, paged memory systems exploit spatial locality by caching collections of entries (a page) as a single entity; theorem-proving caches must deal instead with single entries. Nevertheless, one might hope that some analogue to temporal locality exists on which to base an effective cache-management policy.[5]

2.3. Imposing Cache Hit Generality Constraints

Using a fixed-size cache will limit the maximum cache overhead associated with a caching scheme. More importantly, it will permit us to apply information acquired in the course of solving one problem to subsequent problems without incurring unreasonable cache overhead costs. Unfortunately, even a fixed-size cache may negatively affect performance in domains where adverse search effects occur. To see why

[5] Implicit in our strategy of limiting the number of cache entries is the assumption that all cache entries are of roughly the same size. In some domains, however, the size of individual cache entries may not be at all uniform: in such domains it may make more sense to place limits on the space available for storing cache entries instead of limiting the absolute number of cache entries. This fixed-space strategy does make deciding what to remove from a full cache more complicated, since removing a useless cache entry that occupies little space may not free enough space to insert a larger new cache entry.

this is so, consider the interaction of our simple success caching scheme with a theorem prover's backtracking behavior. Let us assume that the system is forced to backtrack over a subgoal that had previously matched a success cache entry. This situation occurs when the solution obtained from the cache is not consistent with possible solutions of the subgoal's sibling subgoals. In order to provide alternate solutions to the subgoal, the theorem prover will necessarily consider all alternate paths at that choice point. Since success cache entries represent deductively entailed information, some of the alternate paths considered at this choice point are subsumed by the matching cache entry that was just found wanting. Thus the theorem prover will waste time exploring redundant paths which are known *a priori* to be fruitless. By increasing the branching factor with extra choice points, success cache entries may actually cause an inflated number of nodes to be searched.[6]

We can avoid this problem in a general sense by restricting the applicability of cache entries and changing the backtracking behavior of the theorem prover at cache hits [Elkan89]. By permitting success cache hits only where the candidate cache entry is at least as general as the current subgoal, we can ignore alternate choice points when backtracking over a cache hit. These *cache hit generality constraints* essentially prevent a success cache hit from binding variables in the current search context, obviating the need to consider any alternate search paths that may exist at this subgoal. Once a success cache hit occurs, therefore, the entire search space rooted at that subgoal is effectively pruned and needn't be explored upon backtracking. Imposing cache hit generality constraints — while producing less frequent success cache hits — restricts adverse search effects, with a concomitant increase in the role of beneficial search effects due to success caching.[7]

---

[6] This problem is an instance of the *utility problem* common to speedup learning systems [Minton90].

[7] Note that imposing cache hit generality constraints is somewhat problematic in certain domains. Consider those domains with subgoals where the variables are strictly constrained to be either input or output variables: thus certain variables will always be instantiated when the subgoal is posed, while the rest will be instantiated during the course of the search. For these subgoals, a success cache that imposes generality constraints will never provide any cache hits, since queries will necessarily contain unbound variables while cache entries will necessarily be fully instantiated — and thus, by definition, less general.

While generality constraints serve strictly to limit adverse search effects for success caches, they serve quite a different purpose for failure caches. In fact, for failure caching, cache hit generality constraints are necessary in order to avoid violating the completeness of the search. To see why this is so, consider a failure cache entry that is more specific than the current query. Allowing this entry to satisfy the query (thus binding some number of variables) amounts to taking a specific instance as evidence that no other instance of the more general query can exist. In other words, should some different, more specific, instance of the query be in fact provable, taking this cache entry as a cache hit would preclude finding the solution. Thus in order to constitute a failure cache hit, an entry must, first, be at least as general as the query, and, second, should have a resource annotation that equals or exceeds the resources remaining in the current search state.

Thus in this paper we are advocating the use of fixed-size caches with explicit cache management policies to store previously proven subgoals as well as subgoals which are known to fail within a certain resource bound. The caches we advocate rely on cache hit generality constraints to ensure correctness for failure caches and to avoid adverse search effects for success caches. Imposing these constraints brings our caching scheme more into line with memoization in the traditional sense.

3. An Empirical Evaluation of Bounded-Overhead Caching

Given the general framework for bounded-overhead caching outlined in the previous section, we now explore different aspects of cache design empirically. Among the aspects of cache design we examine, we consider the relative performance of different cache management policies, the coexistence of success and failure entries in a unique cache, and the impact of redundant cache entries on system performance. Empirical evaluations in the literature typically compare solution costs (usually in terms of CPU time) both with and without caching over a collection of problems. Unfortunately, this methodology yields results that are necessarily dependent on a particular problem, theorem prover, domain theory, and hardware configuration (in fact, even just running the identical experiment twice on the same hardware

may well yield different results). Nonetheless, reporting CPU time does support at least some form of qualitative comparison.

While such qualitative results are no doubt useful, we would ideally like to obtain at least some results that are divorced from the theorem prover and hardware used in the experiments (experimental results will unavoidably depend on the problem set and domain theory used for the experiment). Thus, in addition to the bottom-line judgements provided by CPU time comparisons, we rely on a model-based experimental methodology to draw conclusions which are more generally valid.

3.1. Methodology

It is difficult to extrapolate reliably from empirical data. In [Segre91b] we outline some common methodological problems encountered in experimental evaluations of speedup learning systems. In [Segre91c] we present an experimental methodology for comparing speedup learning systems that avoids many of these pitfalls. Since caching can also be viewed as a form of speedup learning, we can adopt some of these techniques to our evaluation of caching. These techniques allow us to obtain a more precise, quantitative, picture of the effect caching has on performance.

Our experimental methodology is based on a mathematical model of theorem proving as search. The search space explored by a theorem prover is a function of the problem being solved, the domain theory used, and the theorem prover (*e.g.*, the search strategy employed). Our basic assumption is that, independent of a particular theorem-proving system's implementation details, the size of the space explored — and therefore the time required to search — grows exponentially with the difficulty of the problem being solved. More formally, we can relate the time t to solve a problem of difficulty δ in a search space with average branching factor b and per-node exploration cost c as:

$$t = cb^{\delta}. \tag{1}$$

By measuring t over a collection of problems of known difficulties, we can derive estimates of b and c

using standard methods of parametric statistics. Direct performance comparisons between two different theorem provers — or the same theorem prover operating with different cache configurations — solving representative suites of test problems can be made by comparing their respective b and c parameters. If b for one is lower than b for the other, then, in the limit (*i.e.*, for difficult enough problems), we can conclude the first theorem prover will perform systematically faster than the second.

There remains a final, pragmatic, issue to address before we can apply this model to our experiments: we must somehow obtain values of problem difficulty for each of the problems in the test suite. There are several options. One might, for instance, approximate δ using the depth or size of the first proof found. While this estimate fits the underlying mathematical model of Equation 1 quite well, it has some problems. In particular, if a problem has more than one solution and these solutions vary in size or depth, changing the caching strategy may change which of these solutions is found first. Thus, this practice violates the assumption that difficulty does not depend on the theorem prover, but rather is an intrinsic characteristic of the problem itself.

Therefore, we propose to use a breadth-first search control system to solve each problem in the test suite. The time to solution is recorded in terms of elapsed CPU time. Since the control system is itself a search system, it should also adhere to our mathematical model of Equation 1. Thus for the control system:

$$t_{bfs} = c_{bfs} b_{bfs}^{\ \delta}. \tag{2}$$

We note that the b_{bfs} and c_{bfs} parameters are independent of the problem being solved; instead, they are constants characterizing the theorem prover, domain theory and problem set as a whole. Recognizing that $t_{bfs} = e_{bfs} c_{bfs}$ where e_{bfs} is the number of nodes explored by the control system to find a solution, we can cancel c_{bfs} from Equation 2, take the logarithm of both sides of the equation and fold b_{bfs} into a constant of proportionality to solve for δ:

$$\delta \approx log(e_{bfs}). \quad (3)$$

In summary, given a number of datapoints of the form $(log(e_{bfs}), log(t))$ obtained on a collection of test problems, we can obtain estimates of the regression parameters $log(b)$ and $log(c)$ using linear regression in accordance with the following regression model:

$$log(t) \approx log(b)log(e_{bfs}) + log(c). \quad (4)$$

A lower regression slope $log(b)$ in general corresponds to a theorem prover whose performance scales better to larger problems. The main advantage of this methodology is that it allows us to predict performance on relatively larger problems from data collected on relatively smaller problems. We can extrapolate with confidence as long as we believe the underlying mathematical model of theorem proving just discussed.

3.2. Experiment 1

An example should help make our methodology clear. In this first experiment, we are interested in comparing the performance of a non-caching theorem prover with an identical theorem prover that uses an unlimited-size success and failure cache.

This study relies on a heavily-instrumented, depth-first iterative-deepening definite-clause theorem prover implemented in Common Lisp. As the theorem prover expands each subgoal, it checks the cache first, and only then, if necessary, resorts to checking the database. The cache implementation is flexible, allowing the user to vary cache size and to specify arbitrary cache management strategies. Note that the theorem prover is not particularly fast, since it was designed primarily in order to support principled experimentation. For example, like the caching subsystem, the search strategy used by the theorem prover is flexible; it can be configured to perform iterative deepening, iterative broadening, conspiratorially best-first iterative deepening, or even simple breadth-first search. In fact, the same theorem prover (configured to perform breadth-first search) is used as the control system.

The domain theory and problem set used for this experiment are shown in Appendix A. The problem set consists of 26 problems drawn from a simple situation-calculus formulation of the classic AI block-stacking world [Sussman73]. Each problem is solved by the control theorem prover, a non-caching breadth-first search configuration of the theorem prover. The smallest problem requires searching 4 nodes and corresponds to a derivation tree consisting of 4 nodes 1 level deep. The largest problem requires searching approximately 16,000 nodes and corresponds to a derivation tree of 84 nodes 7 levels deep. The logarithm of the number of nodes explored $log(e_{bfs})$ is recorded for each problem for use as the estimator of problem difficulty δ.

Two trials using the depth-first iterative-deepening theorem prover were performed. The first trial involved no caching, while the second trial used an unlimited-size cache. Each trial consisted of solving all 26 problems presented in the same random order using a resource limit of 30,000 nodes explored. In the second trial, the cache is not cleared between problems. Both trials were performed using a unit-increment iterative deepening strategy.[8] Elapsed time to solution (in milliseconds) is recorded for each of 26 problems solved and a two-parameter linear regression is performed using Equation 4 as the regression model.

Figure 1 illustrates the performance of the non-caching system. As might be expected, this system achieves an excellent fit ($r^2 = 99.8\%$), since the unit-increment iterative deepening system and the control system explore the search space in identical order. Intuitively, this helps to lend credence to our methodology's underlying mathematical model by illustrating how the number of nodes exploited by a control system can in fact be excellent predictors of CPU time performance for a different system operating with the same domain theory on the same problem set.

---

[8] A unit increment may well produce the worst-case performance for iterative deepening. Depending on the problem population, increasing the increment value may substantially improve the system's performance.

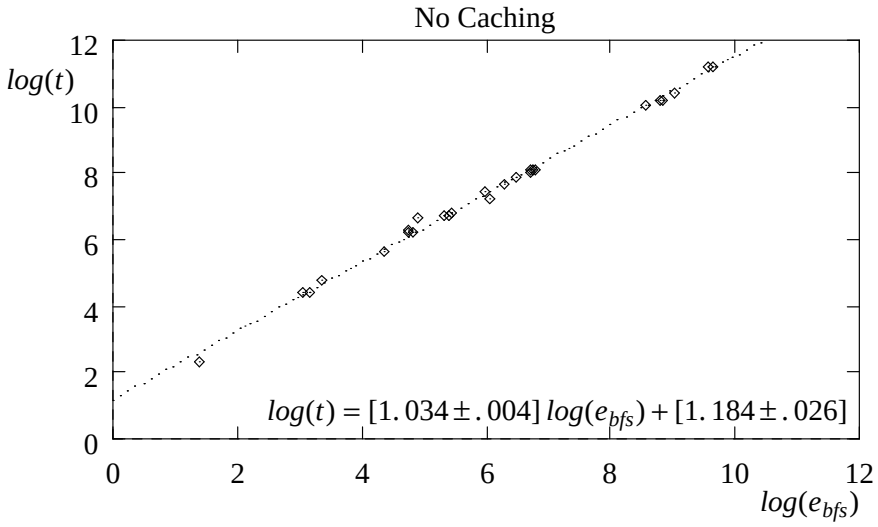

Figure 1: Performance of a non-caching iterative-deepening theorem prover on 26 problems from the situation-calculus domain theory of Appendix A. Each datapoint shown corresponds to one or more problems, since some problems have exactly the same solution characteristics. Total time to solve 26 test problems was 273.9 seconds.

Figure 2 shows the performance of the unlimited-size caching system. While the performance of the unlimited-size caching system is dependent on problem ordering, the performance of the non-caching system is not: nonetheless, the problems are presented in exactly the same random order as for the non-caching system of Figure 1. As noted previously, the cache is not flushed between problems. The 26 problems are solved in a total of 669.8 seconds, at which point the cache contains a total of 10,753 entries, 1,304 of which served to provide a cache hit at some time during the trial.

The plot in Figure 2 suggests several striking observations. First, we note that almost all datapoints in this second plot have greater y-values than their corresponding datapoint in Figure 1: thus, on this randomly-ordered set of test problems, the unlimited-size caching system is slower than the non-caching system on almost every problem. In fact, the caching system is more than twice as slow than the non-caching system over the test problem set as a whole. Second, from the regression parameters obtained, it

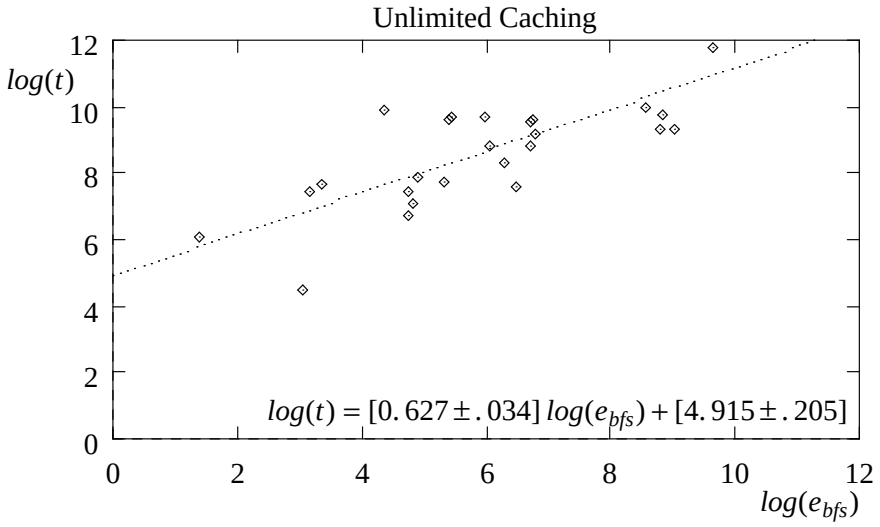

Figure 2: Performance of an unlimited-size caching iterative-deepening theorem prover on the same situation-calculus problem set of Figure 1. Total time to solve all 26 test problems was 669.8 seconds.

appears that the unlimited-size caching system has, as expected, a greater node exploration cost than the non-caching system. This greater node exploration cost c reflects the cache overhead costs and shows up in the plot as a larger y-intercept ($log(c)$ in Equation 4) value. Finally, given the lower computed regression slope for the unlimited-size caching system, we might expect that, on large enough problems, the caching system will be faster.

Is this last conclusion warranted? Unfortunately, no: the problem lies in our methodological assumption that c is invariant for a given theorem prover, domain theory, and problem set. For the unlimited-size caching system, however, c clearly grows monotonically as more items are added to the cache. Thus while it might appear by extrapolation that, for large enough problems, the unlimited-size caching system will prove faster than the non-caching system, this may not be true given that the intercept

value for the unlimited-size caching system will continue to increase.[9] Whether or not the unlimited-size caching system will ever prove to be quicker than the non-caching system depends on the implementation, domain theory, and problem distribution.

There is one other interesting piece of information which can be reliably extracted from our experiment with the unlimited-size caching system. In particular, we would very much like to know the magnitude of the beneficial search effect possible due to caching. As noted previously, the beneficial search effect due to unlimited-size caching represents a sort of empirically-measured best-case reduction in search available for any caching scheme.

To isolate search effects due to caching from cache overhead, we make a small modification to our experimental methodology. Substituting $t = ec$, we can cancel $log(c)$ from Equation 4 and use $log(e)$ directly as the dependent variable in the experiment. In this fashion, we factor out cache overhead, leaving:

$$log(e) = log(b)log(e_{bfs}) \tag{5}$$

as the experimental regression model. This simplified model highlights implementation-independent search effects without conflating implementation-dependent cache overheads. The single-parameter regression equation also reflects the fact that proofs of problems which require exploring a single node (*e.g.*, retrieving an axiom from the database) without caching will still require an identical amount work even if a cache is in use; thus the plot goes through the origin as expected.

Figure 3 shows the search performance of the unbounded overhead success and failure caching system. Certain problems are helped (*i.e.*, fewer nodes are explored) by the presence of cache entries, and corresponding datapoints shift downwards since the cost of solving any given problem with the control system is invariant. Other problems are not affected by the presence of cache entries, so their respective

[9] This last observation, of course, also implies that the computed regression parameters are not very meaningful here since they are computed using a regression model that assumes fixed c.

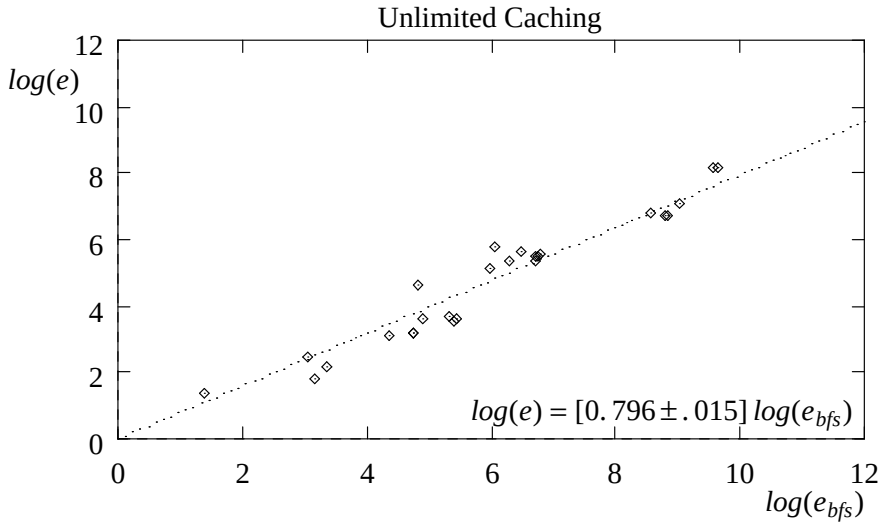

Figure 3: Search performance of an unlimited-size caching iterative-deepening theorem prover on the same situation-calculus problem set of Figure 1. Cache overhead effects are factored out.

datapoints remain unchanged.

Since the problems are presented in random order, linear regression — by minimizing the sum of the squares of the errors — provides a good estimate of the slope over the problem distribution as a whole. As the datapoints spread downwards, the regression slope decreases, reflecting the need to search fewer nodes on average over all problems in the population. The regression slope obtained here ($log(b) = [0.796 \pm .015]$) implies that the system searches significantly fewer nodes than the breadth-first search control system, which would, by definition, yield a slope of exactly $log(b) = 1$ when measured against itself. A similar analysis for the noncaching system (plot not shown) yields a one-parameter regression slope of $log(b) = [1.033 \pm .004]$, indicating that the noncaching system explores a larger number of nodes than the control system. Again, this is to be expected, since unit-increment depth-first iterative deepening explores the space in precisely the same order as breadth-first search, but by performing iterative deepening will explore some nodes more than once.

Thus this simple situation-calculus domain serves as an example of an application where unlimited-size caching is inadequate. More precisely, while this kind of caching may reduce the number of nodes explored in search of a solution (as is evident from our analysis of the search effects), it causes an overall decrease in performance in this domain presumably due to increased overhead. The goal of bounded-overhead caching is to capture as much as possible of the beneficial search effect without incurring excessive node exploration costs.

3.3. Experiment 2

In this experiment, we evaluate the performance of bounded-overhead caches across a variety of cache sizes and management strategies. The configurations tested here are:

- LRU replacement,
- LFU replacement,
- FIFO replacement, and
- RANDOM replacement.

These four systems were tested on all 26 problems using caches ranging from 10 to 1000 elements. As before, the problems were presented in the same random order for all trials; in addition, the caches are left undisturbed between problems.[10]

The first question we would like to answer is whether or not a bounded-overhead caching system can outperform both the non-caching and the unlimited-size caching systems of the previous section. Since the resource limit given for each query was sufficient to solve every problem, as a first

[10] The RANDOM cache management strategy involves selecting an arbitrary cache entry for replacement and therefore involves minimal overhead. FIFO maintains the cache entries as a queue, placing new entries at the end of the queue while deleting the first queue element, while LRU is implemented as a modification of FIFO where a cache hit causes the corresponding cache entry to move to the end of the queue. Both of these strategies also entail minimal overhead. More problematic is LFU, since a naive implementation would entail a substantially higher cache overhead than the other bounded-overhead strategies. Instead, a variant of LRU called a *creeping cache* is used to approximate a real LFU policy while displaying exactly the same overhead characteristics as LRU. A creeping cache operates by demoting a corresponding cache entry pointer by one position in the queue for every cache hit. An oft-hit entry will thus trickle back to the end of the queue where it is unlikely to be replaced.

approximation we can simply plot cumulative solution times over the entire test suite.[11]

Figure 4 shows the results of this experiment. There are two observations that bear mentioning. First, for every cache management strategy tested, using a small cache initially causes an increase in time to solution. As the cache size is increased, performance improves and then degrades again. This behaviour is consistent with our expectations; a very small cache bears much of the overhead costs yet yields little of the beneficial search effects. As the size grows larger, the beneficial effects of caching become evident but are eventually overwhelmed by increasing cache overhead. This analysis, of course, relies on a hidden, yet perhaps unwarranted, assumption that beneficial search effects increase monotonically with cache

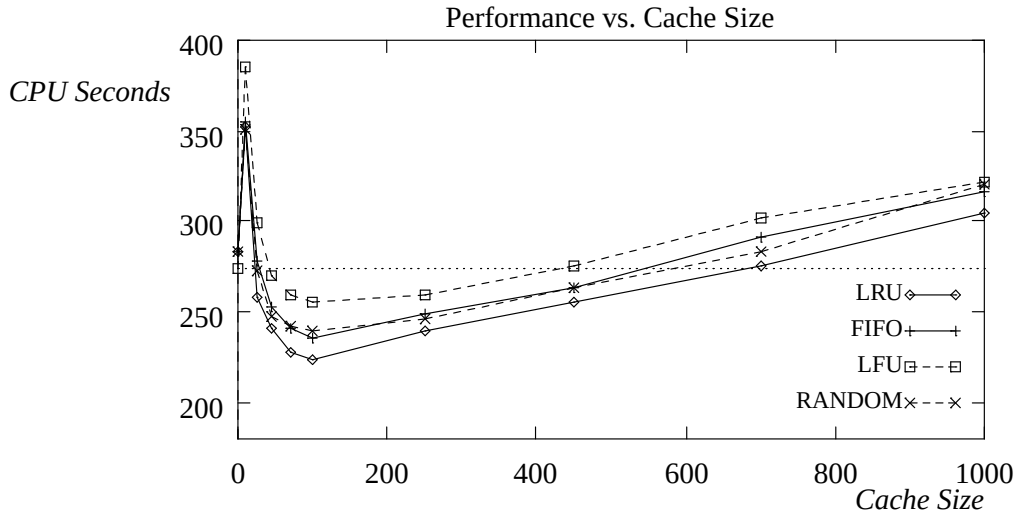

Figure 4: Performance of four bounded-overhead caching schemes as a function of cache size. Performance is measured in terms of cumulative CPU seconds to solve the same 26 situation-calculus problems used in Experiment 1. The horizontal line corresponds to the performance of the non-caching system (273.9 seconds); recall the unlimited-size caching system requires 669.8 seconds to solve all 26 problems.

[11] Note that the use of cumulative CPU times does tend to skew the relative importance of individual problems by emphasizing the larger problems. While there might be other ways of presenting CPU performance data, the use of cumulative CPU times is simple, intuitive, and, most important, consistent with the literature.

size. The second observation is that, while all four of the tested strategies behave in approximately the same fashion, LRU displays slightly better performance than the others. It is not possible to tell whether LRU's edge lies in lower cache overhead costs relative to the other strategies or in some increased beneficial search effect. The answer to this question hinges in part on implementation-specific aspects of the cache. In particular, while cache lookup costs are roughly equivalent for all implementations (modulo differences in actual cache contents, of course), the cost of maintaining the cache itself may differ from one strategy to the next.

We can check both of these informal analyses by once again factoring out the implementation-dependent cache overhead costs and focusing on the implementation-independent search effects. We would like to know, first, how the magnitude of the beneficial search effect changes with cache size, and, second, if the different caching strategies display substantially different beneficial search effects. We again turn to the one-parameter regression model of Equation 5 to factor cache overhead costs out of the analysis. In Figure 5, we plot the value of the regression parameter $log(b)$ against the size of the cache used for all four bounded-overhead cache management strategies listed above. We expect that the smaller cache sizes will have empirically-measured slopes very close to the value obtained for the non-caching system ($log(b) = [1.033 \pm .004]$), while larger cache sizes should approach the slope obtained for the unlimited-size caching system ($log(b) = [0.796 \pm .015]$).

We are now in a position to check the two observations cited earlier. First, we note that not only is the beneficial search effect due to caching increasing monotonically with cache size, but that most of this effect is evident even with relatively small caches. Second, we note that the search performance of different policies is relatively homogeneous, although LRU does show some slight edge for all cache sizes tested. This latter observation means that LRU's slight overall performance edge from Figure 4 is at least to some degree based on search effects rather than only differences in cache overhead.

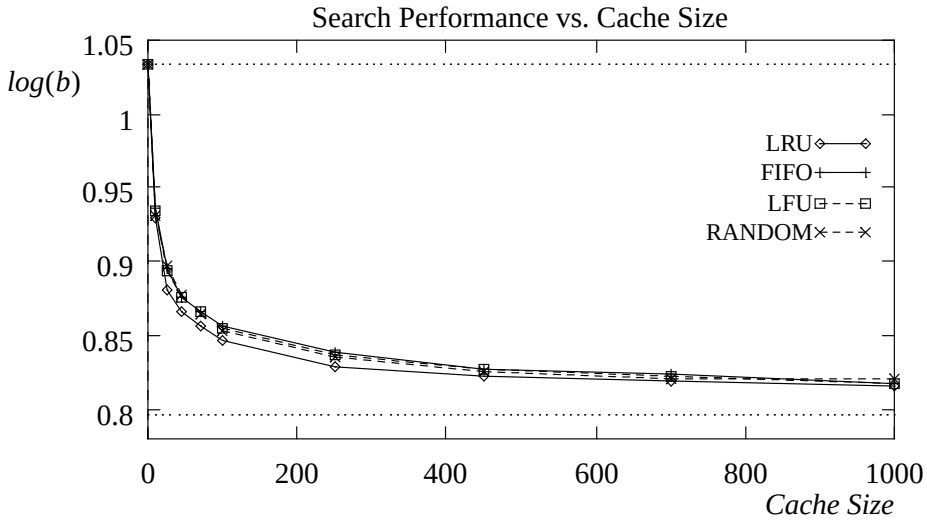

Figure 5: Search performance of a bounded-overhead caching iterative-deepening theorem prover using LRU, FIFO, LFU, and RANDOM cache management strategies. The graph plots the empirically obtained one-parameter regression slope $log(b)$, an indicator of the search space size, against the size of success and failure caches. The horizontal lines correspond to the search performance of the non-caching system ($log(b) = 1.033$) and the infinite-size caching system ($log(b) = .796$).

Notwithstanding LRU's slight edge, the search performance over all four strategies is remarkably uniform. There are two alternative interpretations for this striking similarity in search performance. The first interpretation is that the subgoals explored by a theorem prover do not fit any regular pattern of exploration. If this is true, than a random replacement strategy will provide adequate cache entry replacement guidance. The second interpretation is that the exploration pattern of iterative deepening search strategies does exhibit some analogue to locality of reference, but that we are as yet incapable of exploiting it because we simply do not understand it. This second interpretation leaves open the possibility that one might develop an analytic model of iterative-deepening exploration that will suggest an improved cache management strategy that would eventually outperform LRU. Selecting which of these two competing interpretations is correct is difficult. While we know that a hypothetical optimal

caching system's performance should not exceed the performance of the unlimited caching system ($log(b) = .796$) for this particular problem ordering, we don't really know how it compares with LRU for fixed-size caches.

When designing a hardware cache, one may resort to approximating the performance of an optimal or nearly-optimal caching system by examining page-access traces collected during execution of some benchmark programs. Unfortunately, we cannot rely on this kind of static analysis to predict the performance of a fixed-size cache for theorem proving. The reason is that, unlike paged memory systems where the page access pattern is determined by the program being benchmarked, the pattern of cache accesses is not fixed, but rather changes depending on the cache contents. A cache hit (or lack thereof) changes the search behavior of the system; thus it is simply not possible to use static trace information from one cache configuration to predict system performance with a different cache configuration.

3.4. Experiment 3

In the previous experiment, we tested cache-management policies suggested by analogy to hardware systems. In this section, we begin to explore alternative cache-management strategies based on more theorem-proving specific models of cache entry utility. One would hope that these strategies might more adequately reflect the underlying iterative-deepening search process, resulting in better performance than simple LRU caching.

Traditional paged-memory hardware caching systems generally assume that the cost of a page replacement is independent of the page being replaced. Cache-management policies such as LRU, LFU and FIFO rely at least implicitly on this assumption; a decision to replace a cache element is made based only on its past usefulness rather than on any notion of its original cost. Our success and failure cache

entries are not all of uniform cost.[12] In this experiment, we introduce and test two variants of the LRU cache management policy that do not assume all cache entries are of uniform cost.

The *cheapest least-recently used* policy (CLRU) selects for replacement the least-recently used cache entry whose solution cost (expressed in number of nodes explored) is exceeded by the new cache entry's solution cost. If no cache entry matching this criteria is found, the new entry is simply discarded. In a similar fashion, the *dearest least-recently used* policy (DLRU) looks for the least-recently used cache entry whose solution cost is larger than the new cache entry's solution cost. These two policies explore

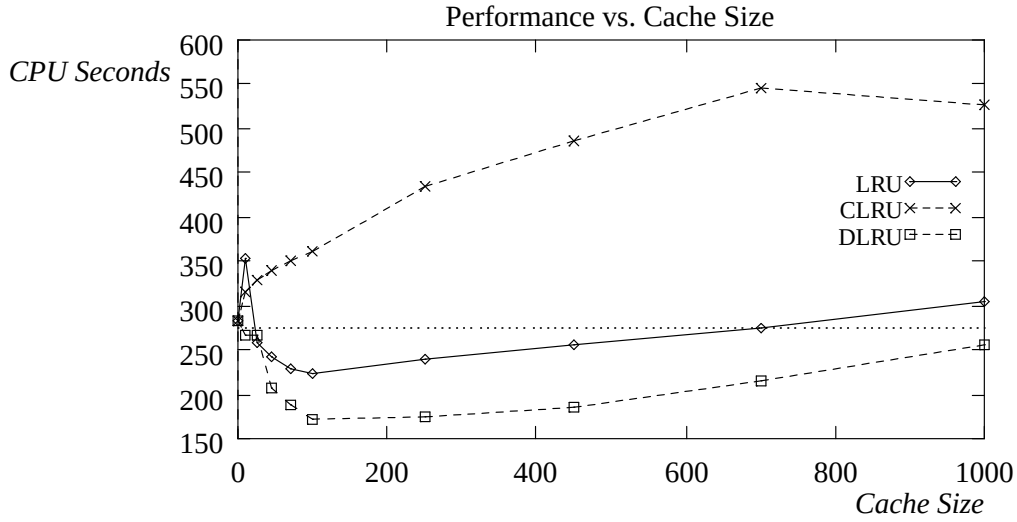

Figure 6: Performance of CLRU and DLRU compared with LRU as a function of cache size. Performance is measured in terms of cumulative CPU seconds to solve the same 26 situation-calculus problems used in Experiment 1. The horizontal lines corresponds to the performance of the non-caching system (273.9 seconds); recall the unlimited-size caching system requires 669.8 seconds to solve all 26 problems. Note that the vertical scale is compressed with respect to Figure 4.

[12] More recent work on caching systems for shared-memory non-uniform memory access machines takes into account the differing latencies of local *vs.* remote data items. However, unlike our theorem prover, these systems generally need only worry about two possible costs: cheaper local access and more expensive remote access.

fundamentally different intuitions about which cache entries are more likely to be useful in solving future problems. CLRU looks for relatively infrequent cache hits that produce large savings, while DLRU strives for more frequent, less dramatic, cache hits.

While still qualifying as bounded-overhead caches, the CLRU and DLRU caches will carry higher cache overheads than the other caches described earlier; in a naive implementation, an unsuccessful cache insertion event may, in the worst case, take time proportional to the size of the cache. All other things being equal, even if more sophisticated implementations are available, the additional bookkeeping required will result in higher cache overheads than, for example, simple LRU.

Figure 6 plots the cumulative CPU time required to solve all 26 problems against cache size for CLRU, DLRU, and LRU. While DLRU significantly outperforms LRU, CLRU's performance is much worse than either of the other two strategies. In fact, even from a qualitative perspective, these three strategies display markedly different performance curves. Unlike LRU, DLRU provides an immediate gain in performance even for very small cache sizes; there is no initial degradation due to the extra cost of operating a cache followed by performance improvement as the beneficial search effects counteract the cache overhead. On the other hand, CLRU's performance degrades immediately and continues to get worse as the cache size increases.

Figure 7 plots the search performance of CLRU, DLRU and LRU as a function of cache size. Given the respective performances of these policies shown in Figure 6, we would expect DLRU to explore the smallest space, followed by LRU and CLRU. While our expectations regarding the relative sizes of the spaces explored by CLRU and LRU are met, DLRU's performance advantage does not, surprisingly enough, seem based on a reduction in search space.

One explanation for CLRU's poor performance is that it might be possible to fully populate the cache with expensive — but useless — tenured cache entries that are never replaced. To test this hypothesis, we implemented a variant of CLRU called *probabilistic cheapest least-recently used* that

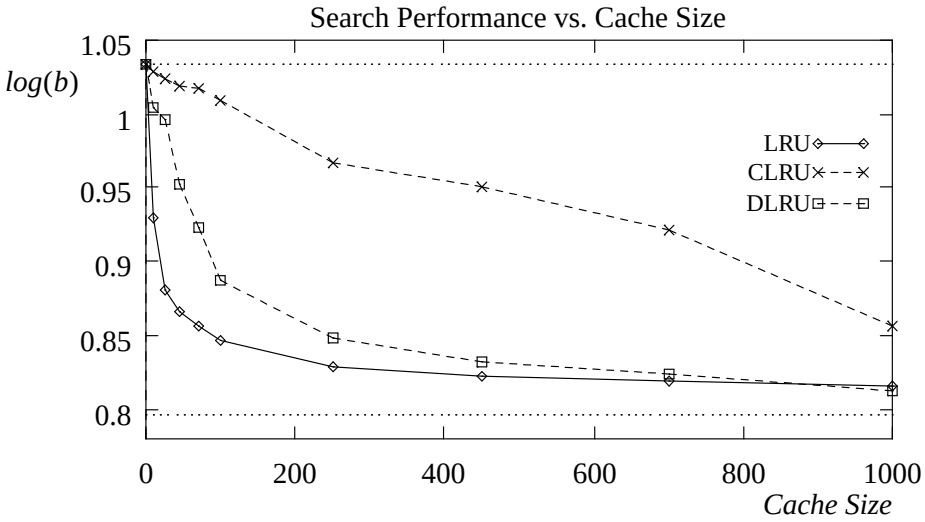

Figure 7: Search performance of CLRU and DLRU compared with LRU as a function of cache size. The graph plots the empirically obtained one-parameter regression slope $log(b)$, an indicator of the search space size, against the size of success and failure caches. The horizontal lines correspond to the search performance of the non-caching system ($log(b) = 1.033$) and the infinite-size caching system ($log(b) = .796$).

allows a new entry to replace a more expensive existing cache entry with probability inversely proportional to cache size. In this fashion, a new cache entry always has at least a chance to replace an existing entry even if the cost-based replacement criteria are not strictly met. Testing of this variant policy supports the tenured cache entry hypothesis, since its search performance (not shown) was found to closely approximate the search performance of LRU.

The behavior of DLRU is somewhat more difficult to explain. On one hand, the overall performance (Figure 6) is better than LRU, but on the other hand, the search performance (Figure 7) is worse than LRU. This implies that DLRU's performance advantage is based on lower cache overhead rather than reduced search. However, we also know that the implementations of DLRU and CLRU are identical, save for the sign of a single arithmetic comparison. Furthermore, we know that for identical cache contents

both DLRU and CLRU carry, by design, cache overhead costs that dominate the overhead cost of LRU. Thus it is difficult to see how DLRU's average node expansion cost can be low enough to more than counteract the increase in nodes searched by DLRU with respect to LRU.

The clue to understanding this inconsistency lies in the relative proportion of success and failure entries within the caches. At the end of the problem suite, about 85% of the DLRU cache is devoted to failure entries, while LRU contains only about 40% failure entries and CLRU contains zero (or at most very few) failure entries. Given that we are performing iterative deepening, and that therefore many failures are due to encountering relatively small resource limits, we would expect that failures, on average, will be less costly than successes. Thus we would expect DLRU to populate its cache, on average, with a larger number of failure entries than CLRU.

This observation also helps to explain the measured difference in cache overhead between DLRU, CLRU, and LRU. Recall we expected DLRU and CLRU to display identical overhead costs for identical cache contents. As the differences in proportion in failure and success entries clearly shows, the cache contents are not identical. Thus if failure entries are inherently cheaper to maintain than success entries, we would expect that DLRU, on average, would display lower cache overhead costs than either LRU or CLRU based on the difference in relative proportion of failure to success entries. We explore this issue in the next experiment.

3.5. Experiment 4

In this experiment we examine the respective contributions of success and failure cache entries. Recall our caching system allows both types of cache entries to coexist in a single cache. Alternative implementations might maintain separate success and failure caches, or may perhaps perform only one kind of caching. Naturally, the relative worth of success and failure caching depends on the domain as well as the implementation, since different types of cache hits may entail a different magnitude of

beneficial search effect and failure and success cache overheads may also differ. In this experiment, we run the same set of 26 blocks world problems in the same random order using both success-only caching and failure-only caching systems. Our intent is to measure the relative contributions of success and failure caching to reducing the search space, as well as to investigate possible implementation-dependent differences in cache overheads. As a basis for comparison, we also include the mixed-mode LRU caching scheme of Experiment 2.

Figure 8 compares the performance of success-only and failure-only caching with the mixed caching system used in the previous experiments. As we predicted, the failure-only system's performance curve matches qualitatively that of DLRU in the last experiment, while the success-only system's curve

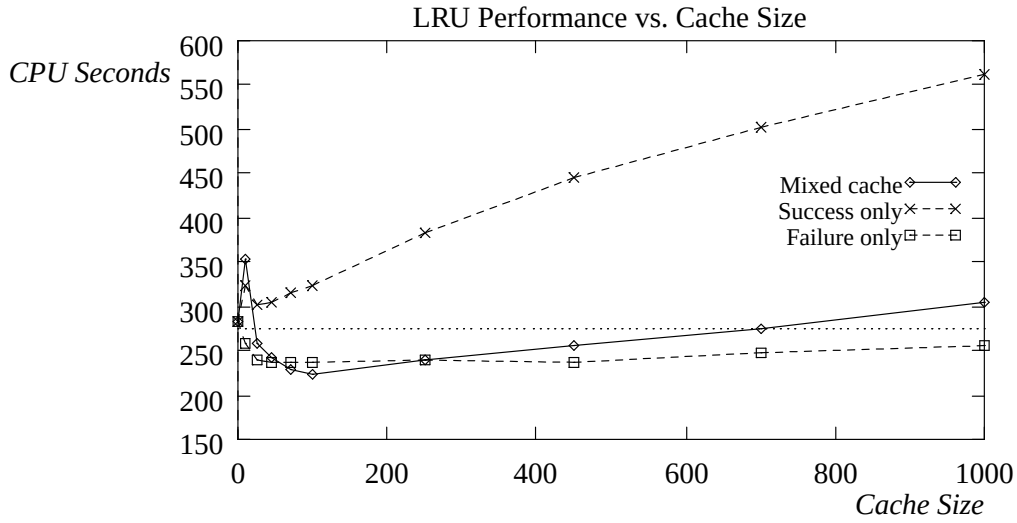

Figure 8: Performance of success-only and failure-only caching systems using an LRU replacement policy as a function of cache size. The mixed cache LRU system of Experiment 2 is included for comparison. Performance is measured in terms of cumulative CPU seconds; the horizontal lines corresponds to the performance of the non-caching system (273.9 seconds); recall the unlimited-size caching system requires 669.8 seconds to solve all 26 problems.

approximates that of the mixed LRU cache. To determine if the root cause is an actual difference in overhead for the two types of cache, Figure 9 plots the search performance of all three strategies. Given that the reductions in search space do not correspond with system performance plotted in Figure 8, we must conclude that per-node exploration costs are far from uniform for equivalent cache sizes. This is in fact easily confirmed via direct inspection of the data; for example, for 100 element caches, the mixed-mode cache explored a total of 18,815 nodes in 262.1 seconds (13.9 msec/node) over the entire test suite. The success-only system explored 41,120 nodes in 323.1 seconds (7.9 msec/node), while the failure-only system explored 41,549 nodes in only 237.4 seconds (5.7 msec/node).

How can we account for these fundamentally different node expansion costs? That such differences

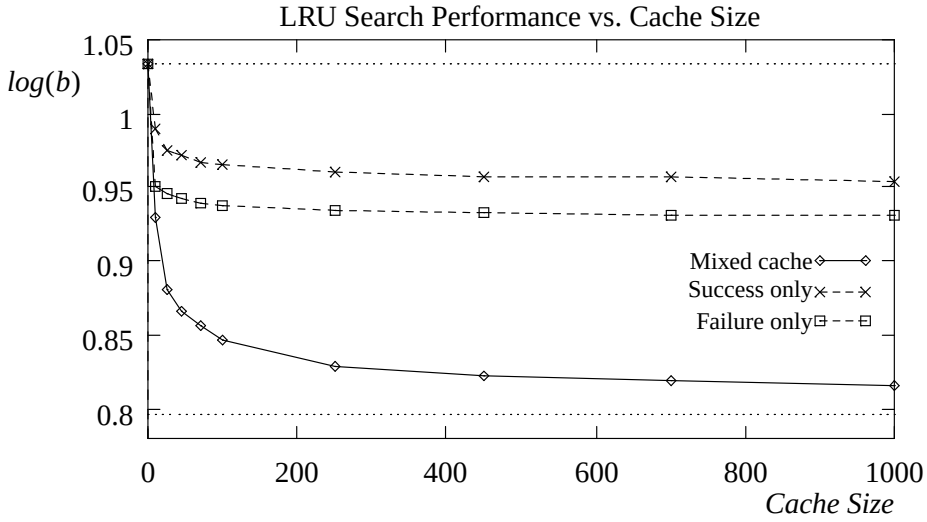

Figure 9: Search performance of success-only and failure-only caching systems using an LRU replacement policy as a function of cache size. The mixed cache LRU system of Experiment 2 is included for comparison. The graph plots the empirically obtained one-parameter regression slope $log(b)$, an indicator of the search space size, against the size of success and failure caches. The horizontal lines correspond to the search performance of the non-caching system ($log(b) = 1.033$) and the infinite-size caching system ($log(b) = .796$).

actually exist shouldn't be surprising; success and failure entries are fundamentally different sorts of things. While the precise magnitude of the difference is undoubtedly specific to this implementation, any implementation would almost necessarily exhibit some difference in overhead cost for manipulating success and failure entries.

Given the relative node expansion costs, one might question the utility of caching success entries in this implementation. With the exception of 70 and 100 element caches, failure-only caches outperform mixed-mode caches for all other tested cache sizes (success-only caching performed poorly for all tested cache sizes). Even in the region where mixed-mode caching is faster than failure-only caching, the difference is not very large, and one might argue that the added performance does not warrant the additional implementation complexity. However, repeating this same experiment using a DLRU policy (the policy with the best measured performance in Experiment 2) supports quite a different conclusion.

Figure 10 plots the performance of the same three cache configurations as Figure 8, but with DLRU cache management as opposed to LRU cache management. In this plot the relative performance of the three configurations differs significantly from the relative performance of the LRU systems of Figure 8. Here, the mixed-mode cache system operating with the DLRU policy always outperforms the comparable failure-only system; the 100 element cache using a DLRU policy displays the best performance of any system tested in this paper on this suite of problems. In addition, we note that the success-only system performs better than failure-only and the mixed-mode systems on very small cache sizes. We conclude that one should not discount the importance of success cache entries to the overall performance of the system.

3.6. Experiment 5

Given that we have established the importance of both success and failure cache entries, we next turn our attention to the best relative proportion of these two types of cache entries. Should success and

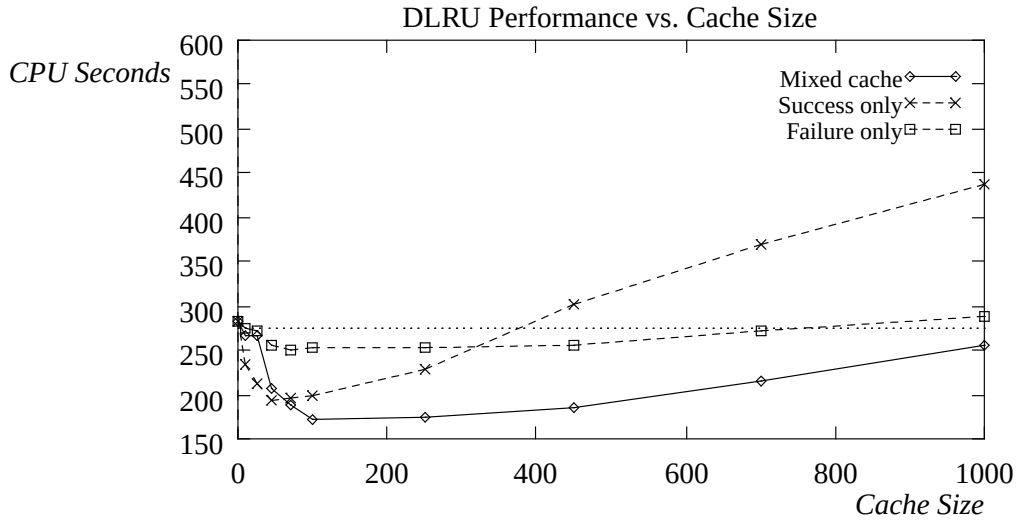

Figure 10: Performance of success-only and failure-only caching systems using a DLRU replacement policy as a function of cache size. The mixed cache DLRU system of Experiment 3 is included for comparison. Performance is measured in terms of cumulative CPU seconds; the horizontal lines corresponds to the performance of the non-caching system (273.9 seconds); recall the unlimited-size caching system requires 669.8 seconds to solve all 26 problems.

failure entries be managed separately in two smaller, separate, caches, or should they be allowed to intermingle in a single cache? If managed separately, what relative sizes should be chosen for the two caches?

In our previous tests using a mixed-mode DLRU cache (Experiment 3), we note that the success/failure ratio measured at the completion of the trial varied from 0/100 to 26/74 percentage of total cache size. For the smaller cache sizes (10 and 25 elements), no success entries were retained at all. The largest percentage of success entries (26%) occurred with a 250 element cache, and tapered off to 13% on the 1000 element cache trial. Here, we will test an alternative dual-cache implementation against the mixed-mode cache system. We run 4 new DLRU trials for each cache size, fixing the ratios of success to failure cache sizes to 20/80, 40/60, 60/40, and 80/20 percentage of total cache size. These results are

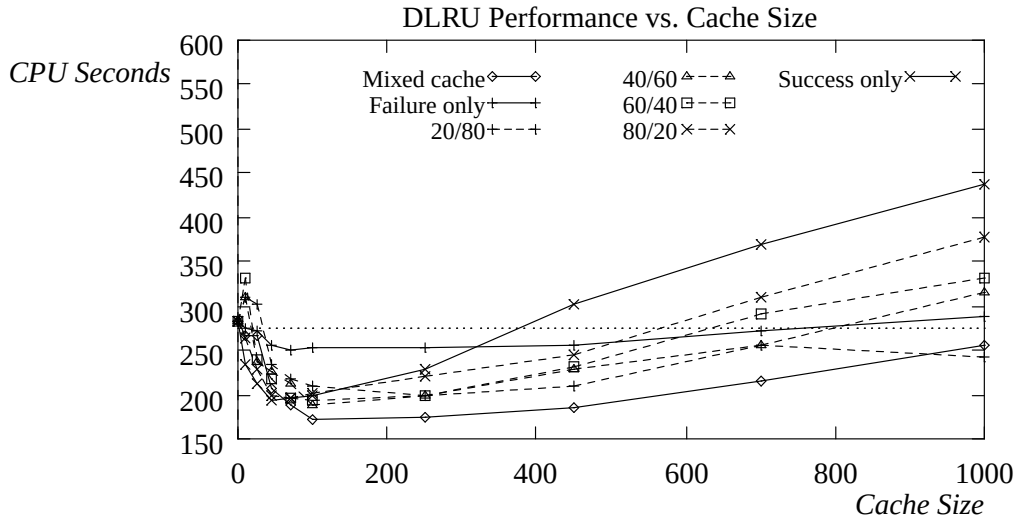

Figure 11: Performance of an assortment of dual-cache implementation trials compared to success-only, failure-only, and mixed-mode caching systems using a DLRU replacement policy as a function of cache size. Performance is measured in terms of cumulative CPU seconds; the horizontal lines corresponds to the performance of the non-caching system (273.9 seconds); recall the unlimited-size caching system requires 669.8 seconds to solve all 26 problems.

compared to the failure-only (*i.e.*, 0/100 success/failure ratio) and success-only (*i.e.*, 100/0 success/failure ratio) systems from Experiment 4 as well as the mixed-mode DLRU performance of Experiment 3.

Figure 11 plots the performance of all seven tested systems. This plot is consistent with several previously mentioned observations. First, it is clear that a mixture of success and failure entries generally outperforms a system that only performs one of success or failure caching (an exception is made for very small cache sizes, where success-only caching performs quite well). Second, we note that when the caches are managed separately, a larger proportion of failures to successes generally entails better performance. One might suspect that part of this effect may be due to the lower overhead costs associated with manipulating failure entries.

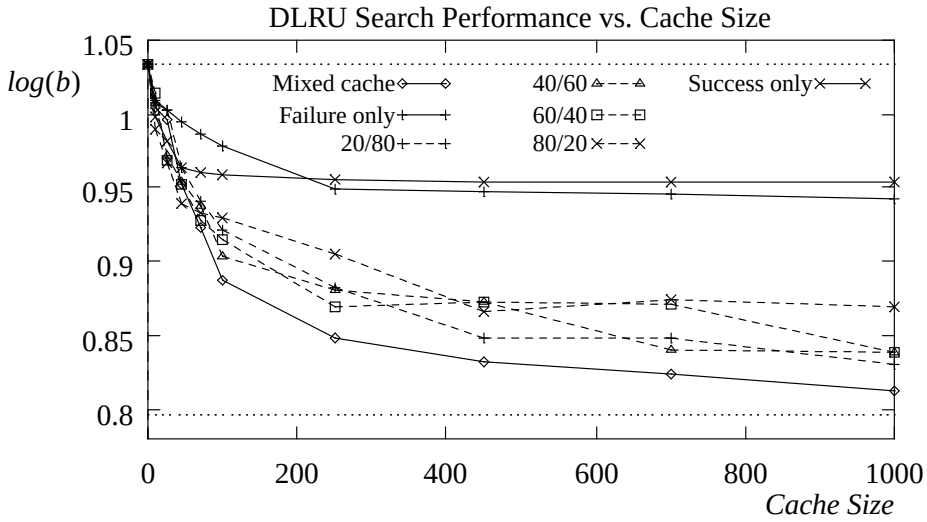

Figure 12: Search performance of a selection of dual-cache systems compared to success-only, failure-only, and mixed-mode caching systems using a DLRU replacement policy as a function of cache size. The graph plots the empirically obtained one-parameter regression slope $log(b)$, an indicator of the search space size, against the size of success and failure caches. The horizontal lines correspond to the search performance of the non-caching system ($log(b) = 1.033$) and the infinite-size caching system ($log(b) = .796$).

Figure 12 shows the search performance of the same seven systems. We note that all of the fixed-proportion systems produce roughly the same amount of search reduction; thus we conclude that the increased performance observed with a larger proportion of failures is probably due to differing relative overhead costs between success and failure entries. On the other hand, it is equally clear that the dynamically-managed mixed-mode cache gets at least some of its performance advantage from actual reductions in search space rather than simply differences in relative cache overhead. It would certainly appear — at least for this cache management strategy and test domain — that forcing success and failure entries to coexist and fight for survival on a uniform basis is the best policy over a broad range of cache sizes, resulting in greater search reduction and better overall performance.

3.7. Experiment 6

In this experiment, we examine the effect of redundant cache entries on the performance of the system. A redundant entry is an entry that is either identical to or subsumed by a different cache entry. Redundant cache entries arise due to the imposition of cache hit generality constraints and also as a result of iterative deepening and failure caching. They reduce the performance improvements obtained with bounded-overhead caching by occupying a portion of the cache with redundant — and therefore useless — information. In addition, the presence of redundant cache entries may interfere with the cache replacement policy; if multiple entries exist for a given query, the usefulness of each of the entries may seem artificially low.

To see how multiple entries can arise, consider a subgoal $q(?x)$ where candidate success entries $q(a)$, $q(b)$, and $q(c)$ are already present in the cache. Since the cache entries are less general than the subgoal, these entries are not allowed to cause a cache hit. If the theorem prover eventually solves the $q(?x)$ subgoal while binding $?x$ to a, a new success entry $q(a)$ is added to the cache, which already contains a copy of this entry. Alternatively, if the theorem prover manages to solve the $q(?x)$ subgoal in its most general form, the new success cache entry $q(?x)$ renders the existing entries $q(a)$, $q(b)$, and $q(c)$ obsolete.

A second source of redundant cache entries is the natural interplay between iterative deepening and resource-limited failure caching. Consider a subgoal $q(a)$ that almost matches a candidate failure entry $q(a)$ in the cache, where the problem is that the resource annotation on the cache entry is smaller than the resources currently available for proving the subgoal. If the prover fails to prove $q(a)$ within the larger current resource limit, we can simply update the resource annotation on the original failure entry (if the original failure entry is still in the cache at failure time). Alternatively, we can simply add a new failure entry with the larger resource annotation and trust the cache replacement policy to discard the other, less general, entry at some time in the future.

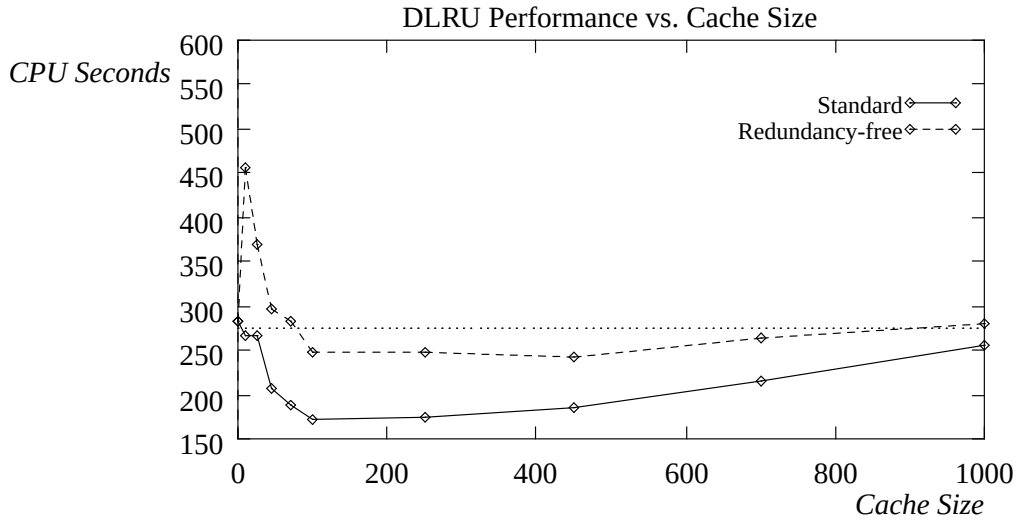

Figure 13: Performance of DLRU caching both with and without redundant entries allowed as a function of cache size. As in previous experiments, the horizontal line corresponds to the performance of the non-caching system (273.9 seconds); recall the unlimited-size caching system requires 669.8 seconds to solve all 26 problems.

There are two approaches for dealing with this problem. The first approach to check for redundant entries whenever a new cache entry is made, at some additional overhead cost.[13] The extra overhead may be more than outweighed by increased cache efficiency. For example, a redundancy-free infinite size caching system requires 480.7 seconds to solve all 26 problems, leaving 4,216 entries in the cache, 697 of which provided cache hits at some point during the trial. When compared with the 10,753 entries and 669.8 seconds required by the standard infinite size caching system, it is clear that the extra redundancy check pays off. Note, however, that even with redundancy checking the infinite size cache does not achieve the level of performance obtained by some of the bounded-overhead cache implementations.

[13] Sophisticated indexing techniques may allow the redundancy check to occur as a side-effect of cache insertion. Nevertheless, while the magnitude of the additional overhead may be limited, some amount of additional overhead is inevitable.

The second approach is particular to fixed size caches. The idea is to ignore the problem and trust the cache management policy to eventually reclaim space allocated to redundant entries. This approach requires a cache retrieval algorithm that guarantees the same entry is retrieved on identical successive queries regardless of any redundant entries that may be lurking within the cache. Management policies such as LRU that are based on the notion of recency have this property; the RANDOM cache replacement policy does not. Note that this approach requires no additional overhead: we simply let the cache take care of itself.

In this experiment, we again rely on the same set of 26 situation-calculus problems used in the previous experiments. A version of the mixed-mode DLRU caching system is altered so that an extra cache lookup is performed at cache insertion time in order to check for redundant cache entries. This system is compared with the same mixed-mode DLRU caching system of Experiment 3. The standard DLRU caching system sorts candidate entries so that older entries are preferred over newer ones, ensuring that redundant entries are never responsible for cache hits.

Figure 13 plots the performance of the two tested systems. Clearly, the additional overhead required to censor redundant entries overwhelms any added search benefit. This is strictly an implementation-dependent result: different implementations will have different overhead characteristics and thus may produce different overall performance. We can, however, obtain some estimate of how much search reduction benefit can be expected when censoring redundant entries. Figure 14 plots the search performance of the two systems. As expected, the standard system requires a larger cache to attain the same search performance as the redundancy-free system, although the search performance advantage of the redundancy-free system does not appear to be terribly large. Of course, the decision to censor redundant entries can only be made in an implementation-specific manner by fully investigating the cache overhead/search performance tradeoff for a particular system. Implementing a more efficient scheme for censoring redundant entries that reduces the associated overhead will naturally tilt this decision in favor of

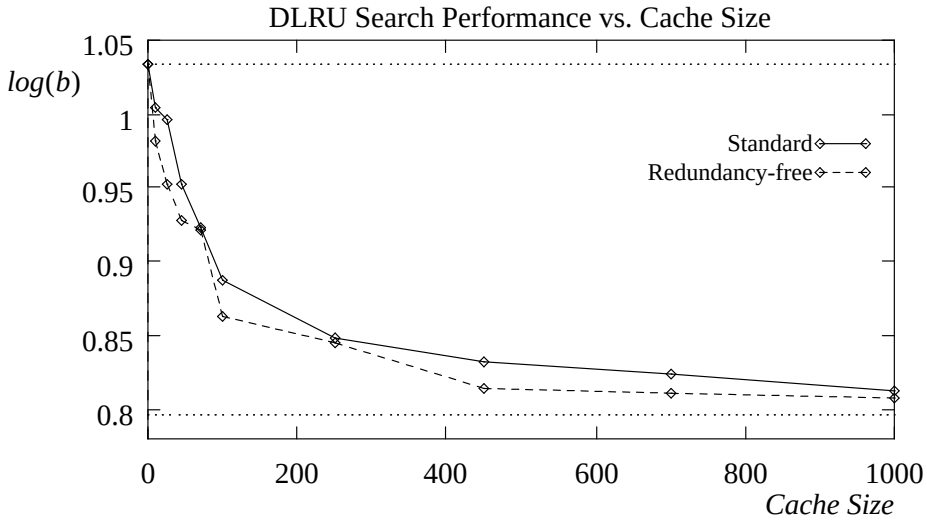

Figure 14: Search performance of DLRU caching both with and without redundant entries allowed as a function of cache size. The graph plots the empirically obtained one-parameter regression slope $log(b)$, an indicator of the search space size, against the size of success and failure caches. The horizontal lines correspond to the search performance of the non-caching system ($log(b) = 1.033$) and the infinite-size caching system ($log(b) = .796$).

censoring redundant entries.

4. Conclusion

Caching is not a new idea. Certainly the most extensive use of fixed-size caches to improve search behavior is within the computer chess community, where even some of the earliest chess playing programs used caches to store previously-computed board strength estimates [Slate77]. More recent work in this area also exploits the use of bounded-overhead caches using cache management policies derived by analogy with hardware caching. For example, Hitech uses a hash table as a fixed-size cache to store previously-generated portions of the search tree. The cache is partitioned into two fixed-size parts. One part is managed using a FIFO management policy, while the other is managed using a cost-based policy that allows replacement of cheaper entries with more expensive entries only (a performance evaluation of

this particular cache configuration can be found in [Ebeling87]). Nevertheless, fixed-size caches have not been traditionally considered appropriate for theorem-proving applications.

Truth-maintenance systems [Doyle78, McAllester82, McAllester80, McDermott91] have in the past been used in lieu of caching to reduce the amount of search required by problem-solving systems. Like caches, truth-maintenance systems record information about previous successes and failures. They are, however, noticeably different from caches. First, truth-maintenance systems are usually allowed to record an unbounded number of facts. This implies the overhead associated with using a TMS will continue to grow with the size of the TMS. Second, truth-maintenance systems are comparatively eager in evaluating constraints which result in failures, rather than allowing the problem solver itself to discover the failure and simply recording the result. Third, truth-maintenance systems generally deal with absolute failures rather than resource-limited failures. Finally, caches tend to entail simpler implementations than truth-maintenance systems. Notwithstanding these significant differences, some solutions used by truth-maintenance systems (*e.g.*, McAllester's hash-consing technique) may easily be transferred to analogous problems in caching (*i.e.*, efficient detection of redundant cache entries).

In this paper, we advocate the use of bounded-overhead caching for definite-clause theorem proving systems. We have justified the use of such caches on the basis of a particular application context: the use of a theorem prover to solve many related problems drawn from a single problem distribution. We believe this application context is a realistic one for many real applications in artificial intelligence, deductive retrieval, and logic programming, and have shown how the traditional approach to caching for theorem proving, that is, the use of unlimited-size caches, is inappropriate in one instance of this general application context.

We have described some of the design decisions inherent in the construction of bounded-overhead caches, focusing on the design of cache-management strategies, the relative contributions of success and failure cache entries, and handling the problem of redundant cache entries. Relying on a reasoned

experimental methodology that allows us to separate implementation-dependent overhead effects from implementation-independent search effects, we have presented a set of experiments designed to explore our intuitions about bounded-overhead caches. Based on our experimental study, we have proposed a new bounded-overhead cache management policy we call *dearest least-recently used*, and have shown how this policy outperforms other, perhaps more obvious, cache management policies in at least one implementation and in one test domain. In summary, for this particular application context and this particular theorem prover, a 100-element DLRU mixed mode success/failure cache provides the best overall performance as measured by smallest total CPU time to solve all the problems in the test suite.

How well do these empirical results scale? There are at least three aspects to this important question; questions that should be asked of *any* experimental study. First, one might ask whether the results obtained with these particular theorem prover and cache implementations are indicative of results obtained with other implementations. We have been quite careful to distinguish between implementation-dependent results (such as the cumulative CPU curves of Figures 1, 2, 4, 6, 8, 10, 11, and 13) and implementation-independent results (such as the search performance curves of Figures 3, 5, 7, 9, 12, and 14). Thus some results, such as the recommendation to forego redundancy-free caching in Experiment 6, depend on aspects of the implementation: in this case, the exact tradeoff between the added cache overhead and the extra search performance edge due to redundancy-free caching. In a similar fashion, the choice of cache size and management policy for best performance are implementation-dependent results. Other results, such as the relative search performance of the different caching strategies, are independent of the implementation altogether.

A second concern is whether the results scale from small problems to larger problems within the same domain. The good news is that our experiments give better reason to believe the results scale across problem size than do most other experiments. Because of the experimental methodology employed, we can extrapolate from small problems to large with the full faith and confidence we have in the underlying

model. It's hard to argue with such a simple mathematical model; a model that just acknowledges that search cost grows exponentially with problem difficulty. Some may, in fact, argue that this model is too simple: note, however, that we do have at least some supporting evidence that the model fits the data quite well (see Figure 1). That is not a guarantee, of course, but it is all any experimental study can offer, and more than most other studies do.

Finally, one might ask if the results obtained in this problem domain can be expected to generalize to other problem domains. Like for problem size, no experimental evaluation can ever guarantee that the results obtained extrapolate over free parameters. The domain theory is a free parameter: simply repeating the experiments in other domains does not suffice as proof that the results are more general, since some untested domain may differ in a critical way that invalidates the results. Until one can posit an underlying model of domain theories to support extrapolation across domains like the model of theorem proving as search supports extrapolation across problem size, our results remain strictly domain-dependent. Thus one should take these results as indicative of what can be achieved rather than a promise of what will be achieved. When it comes to generalizing across problem domains, this is the only guarantee any experimental evaluation has to offer.

We are continuing our experiments with bounded-overhead caching for definite-clause theorem proving in a variety of domains. We are particularly interested in combining bounded-overhead caching with other speedup techniques to form adaptive inference systems. The intuition is that multiple speedup techniques, when brought to bear in concert against problems drawn from a fixed (possibly unknown) problem distribution, can provide better performance than any single speedup technique [Segre92a, Segre93a]. We are also investigating the use of cache information to guide speedup learning algorithms such as EBL* [Segre90], and to guide a heuristic antecedent reordering strategy. We are also looking at the effects of caching for different iterative-deepening strategies such as conspiratorial iterative-deepening [Elkan89] and iterative broadening [Ginsberg92]. Finally, we hope to develop an analytic model of

iterative-deepening search with the intent of exploiting more effective cache management strategies based on this analytic model. It is our hope that an analytic study, when combined with carefully constructed experiments, will lead not only to a better understanding of the mechanics of caching, but also to more effective caching systems for definite-clause theorem proving.

Acknowledgements

Paul Stodghill implemented a preliminary version of the caching code used for the experiments reported in this paper. Gene Ressler suggested the creeping cache approximation of the least-frequently used cache-management policy. Bard Bloom, Wilfred Chen, Robert Constable, Charles Elkan, Mark Stickel, and an anonymous reviewer provided helpful comments on several early drafts of this paper, and Murray Campbell provided pointers to the computer chess literature. Support for this research was provided by the Office of Naval Research grant N00014-90-J-1542, and through an equipment gift from the Hewlett-Packard Corporation.

References

[Astrachan91]
O.L. Astrachan and M.E. Stickel, "Caching and Lemmaizing in Model Elimination Theorem Provers," Technical Note 513, SRI International, Menlo Park, CA 94025-3493 (December 1991).

[Dietrich87]
S.W. Dietrich, "Extension Tables: Memo Relations in Logic Programming," *Proceedings of the IEEE Symposium on Logic Programming* (August 1987), pp. 264-273.

[Doyle78]
J. Doyle, "Truth Maintenance Systems for Problem Solving," Technical Report 419, MIT Artificial Intelligence Laboratory, Cambridge, MA (1978).

[Ebeling87]
C. Ebeling, *All the Right Moves*, MIT Press, Cambridge, MA (1987).

[Elkan89]
C. Elkan, "Conspiracy Numbers and Caching for Searching And/Or Trees and Theorem-Proving," *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (August 1989), pp. 341-348.

[Elkan90]
C. Elkan, "Incremental, Approximate Planning," *Proceedings of the National Conference on Artificial Intelligence* (July 1990), pp. 145-150.

[Ginsberg92]
M.L. Ginsberg and W.D. Harvey, "Iterative Broadening," *Artificial Intelligence* **55**:2-3, North Holland (June 1992), pp. 367-383.

[Green69]
C. Green, "Application of Theorem Proving to Problem Solving," *Proceedings of the First*

International Joint Conference on Artificial Intelligence (August 1969), pp. 741-747.

[Hennessy90]
J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, CA (1990).

[Korf85]
R. Korf, "Depth-First Iterative Deepening: An Optimal Admissible Tree Search," *Artificial Intelligence* **27**:1, North Holland (1985), pp. 97-109.

[McAllester82]
D. A. McAllester, "Reasoning Utility Package User's Manual, Version One," Memo 667, MIT Artificial Intelligence Laboratory, Cambridge, MA (April 1982).

[McAllester80]
D. A. McAllester, "An Outlook on Truth Maintenance," Memo 551, MIT Artificial Intelligence Laboratory, Cambridge, MA (August 1980).

[McDermott91]
D. McDermott, "A General Framework for Reason Maintenance," *Artificial Intelligence* **50**:3, North Holland (August 1991), pp. 289-330.

[Michie68]
D. Michie, "'Memo' Functions and Machine Learning," *Nature* **218** (April 1968), pp. 19-22.

[Minton90]
S. Minton, "Quantitative Results Concerning the Utility of Explanation-Based Learning," *Artificial Intelligence* **42**:2-3, North Holland (March 1990), pp. 363-392.

[Mostow85]
J. Mostow and D. Cohen, "Automating Program Speedup by Deciding What to Cache," *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (August 1985), pp. 165-173.

[Plaisted88]
D. Plaisted, "Non-Horn Clause Logic Programming Without Contrapositives," *Journal of Automated Reasoning* **4**:3, Kluwer Academic (1988), pp. 287-325.

[Segre87]
A.M. Segre, "Explanation-Based Learning of Generalized Robot Assembly Plans," Ph.D. Thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL (January 1987).

[Segre88]
A.M. Segre, *Machine Learning of Robot Assembly Plans*, Kluwer Academic, Boston, MA (March 1988).

[Segre91a]
A.M. Segre, "Learning How to Plan," *Robotics and Autonomous Systems* **8**:1-2, North Holland (November 1991), pp. 93-111.

[Segre92a]
A.M. Segre, "On Combining Multiple Speedup Techniques," *Proceedings of the Ninth International Machine Learning Conference* (July 1992), pp. 400-405.

[Segre91b]
A.M. Segre, C. Elkan, and A. Russell, "Technical Note: A Critical Look at Experimental Evaluations of EBL," *Machine Learning* **6**:2, Kluwer Academic (March 1991), pp. 183-196.

[Segre90]
A.M. Segre and C.P. Elkan, "A Provably Complete Family of EBL Algorithms," Working Paper, Department of Computer Science, Cornell University, Ithaca, NY (November 1990).

[Segre91c]
A.M. Segre, C.P. Elkan, G.J. Gordon, and A. Russell, "A Robust Methodology for Experimental Evaluations of Speedup Learning," Working Paper, Department of Computer Science, Cornell University, Ithaca, NY (October 1991).

[Segre93a]
A.M. Segre, C.P. Elkan, D. Scharstein, G.J. Gordon, and A. Russell, "Adaptive Inference," in *Foundations of Knowledge Acquisition*, Vol. 2, A. Meyrowitz and S. Chipman (Eds.), Kluwer Academic, Boston, MA (January 1993), pp. 43-81.

[Segre93b]
A.M. Segre and J. Turney, "Planning, Acting, and Learning in a Dynamic Domain," in *Machine Learning Methods for Planning*, S. Minton (Ed.), Morgan Kaufmann, San Mateo, CA (July 1993), pp. 125-158.

[Segre92b]
A.M. Segre and J. Turney, "SEPIA: A Resource-Bounded Adaptive Agent," *Proceedings of the First International Conference on Artificial Intelligence Planning Systems* (1992), pp. 303-304.

[Slate77]
D. Slate and L. Atkin, "Chess 4.5 — The Northwestern University Chess Program," in *Chess Skill in Man and Machine*, P. Frey (Ed.), Springer Verlag (1977), pp. 82-118.

[Sussman73]
G.J. Sussman, "A Computational Model of Skill Acquisition," Technical Report 297, MIT Artificial Intelligence Laboratory, Cambridge, MA (1973).

[Tamaki86]
H. Tamaki and T. Sato, "OLD Resolution with Tabulation," *Third International Conference on Logic Programming*, Springer Verlag (July 1986), pp. 85-97.

[Turney89a]
J. Turney and A.M. Segre, "SEPIA: An Experiment in Integrated Planning and Improvisation," *Working Notes of the 1989 AAAI Symposium on Planning and Search* (March 1989), pp. 59-63.

[Turney89b]
J. Turney and A.M. Segre, "A Framework for Learning in Planning Domains with Uncertainty," Technical Report 89-1009, Department of Computer Science, Cornell University, Ithaca, NY (May 1989).

[Warren92]
D.S. Warren, "Memoing for Logic Programs," *Communications of the Association for Computing Machinery* **35**:3 (March 1992), pp. 94-111.

Appendix A

Blocks world domain theory and randomly-ordered problem set used for the experiments reported herein. The domain theory describes a world containing 4 blocks, *A*, *B*, *C*, and *D*, stacked in various configurations on a *Table*. It consists of 11 rules and 9 facts: there are 26 sample problems whose first solutions range in size from 4 to 77 nodes and vary in depth from 1 to 7 inferences deep.

Facts

$holds(on(A, Table), S_0)$ $holds(on(B, Table), S_0)$ $holds(on(C, D), S_0)$
$holds(on(D, Table), S_0)$ $holds(clear(A), S_0)$ $holds(clear(B), S_0)$
$holds(clear(C), S_0)$ $holds(empty(), S_0)$ $holds(clear(Table), ?s)$

Rules

$$holds(and(?x, ?y), ?s) \leftarrow holds(?x, ?s) \wedge holds(?y, ?s))$$
$$differ(?x, ?y) \leftarrow ?x \neq ?y$$

$$holds(holding(?x), do(pickup(?x), ?s)) \leftarrow holds(empty(), ?s) \wedge holds(clear(?x), ?s) \wedge differ(?x, table)$$
$$holds(clear(?y), do(pickup(?x), ?s)) \leftarrow holds(on(?x, ?y), ?s) \wedge holds(clear(?x), ?s) \wedge holds(empty(), ?s)$$
$$holds(on(?x, ?y), do(pickup(?z), ?s)) \leftarrow holds(on(?x, ?y), ?s) \wedge differ(?x, ?z)$$
$$holds(clear(?x), do(pickup(?z), ?s)) \leftarrow holds(clear(?x), ?s) \wedge differ(?x, ?z)$$

$$holds(empty(), do(putdown(?x, ?y), ?s)) \leftarrow holds(holding(?x), ?s) \wedge holds(clear(?y), ?s)$$
$$holds(on(?x, ?y), do(putdown(?x, ?y), ?s)) \leftarrow holds(holding(?x), ?s) \wedge holds(clear(?y), ?s)$$
$$holds(clear(?x), do(putdown(?x, ?y), ?s)) \leftarrow holds(holding(?x), ?s) \wedge holds(clear(?y), ?s)$$
$$holds(on(?x, ?y), do(putdown(?z, ?w), ?s)) \leftarrow holds(on(?x, ?y), ?s)$$
$$holds(clear(?z), do(putdown(?x, ?y), ?s)) \leftarrow holds(clear(?z), ?s) \wedge differ(?z, ?y)$$

Problem Set

$holds(and(on(C, B), on(B, A)), ?s)$ $holds(on(C, Table), ?s)$ $holds(and(on(B, D), on(A, C)), ?s)$
$holds(and(clear(D), on(A, B)), ?s)$ $holds(and(on(D, B), on(B, C)), ?s)$ $holds(and(on(A, B), on(B, C)), ?s)$
$holds(on(D, A), ?s)$ $holds(and(on(D, B), on(C, A)), ?s)$ $holds(clear(D), ?s)$
$holds(and(on(C, D), on(A, C)), ?s)$ $holds(and(on(A, B), on(B, D)), ?s)$ $holds(and(on(D, A), on(A, C)), ?s)$
$holds(and(on(B, A), on(C, B)), ?s)$ $holds(and(on(C, A), on(D, B)), ?s)$ $holds(on(A, D), ?s)$
$holds(and(on(A, B), clear(D)), ?s)$ $holds(and(on(A, D), on(D, B)), ?s)$ $holds(and(on(A, C), on(C, B)), ?s)$
$holds(and(on(B, C), on(A, B)), ?s)$ $holds(on(A, C), ?s)$ $holds(on(D, C), ?s)$
$holds(and(on(B, D), on(C, A)), ?s)$ $holds(and(on(A, C), on(C, D)), ?s)$ $holds(and(on(A, B), on(D, C)), ?s)$
$holds(and(on(D, C), on(C, B)), ?s)$ $holds(and(on(C, B), on(D, C)), ?s)$