Controlling search space materialization in a practical declarative debugger

Ian MacLarty[1] and Zoltan Somogyi[1,2]

[1] Department of Computer Science and Software Engineering,
University of Melbourne, Australia
[2] NICTA Victoria Laboratory
maclarty@csse.unimelb.edu.au zs@csse.unimelb.edu.au

Abstract. While the idea of declarative debugging has been around for a quarter of a century, the technology still hasn't been adopted by working programmers, even by those working in declarative languages. The reason is that making declarative debuggers practical requires solutions to a whole host of problems. In this paper we address one of these problems, which is that retaining a complete record of every step of the execution of a program is infeasible unless the program's runtime is very short, yet this record forms the space searched by the declarative debugger. Most parts of this search space therefore have to be stored in an implicit form. Each time the search algorithm visits a previously unexplored region of the search space, it must decide how big a part of the search space to rematerialize (which it does by reexecuting a call in the program). If it materializes too much, the machine may start to thrash or even run out of memory and swap space. If it materializes too little, then materializing all the parts of the search space required by a debugging session will require too many reexecutions of (parts of) the program, which will take too long. We present a simple algorithm, the *ideal depth strategy*, for steering the ideal middle course: minimizing reexecutions while limiting memory consumption to what is feasible. We show that this algorithm performs well even when used on quite long running programs.

1 Introduction

The aim of the Mercury project is to bring the benefits of declarative programming languages to the software industry. Mercury is designed explicitly to support teams of programmers working on large application programs. It has a modern module system, detects a large fraction of program errors at compile time, and has an implementation that is both efficient and portable. To ensure that programmers can actually enjoy the benefits claimed for logic programs, Mercury has no non-logical constructs that could destroy the declarative semantics that gives logic programs their power.

As part of the project, we have built a declarative debugger for Mercury. We have ample motivation to make this declarative debugger work

well because many parts of the Mercury implementation (including the compiler and most of the declarative debugger itself) are written in Mercury. Using the Mercury declarative debugger to debug the Mercury implementation requires us to confront and solve all the problems that would face anyone attempting to use a declarative debugger to debug large, long-running programs. In previous work, we addressed some usability issues such as how to support the browsing of large (possibly multi-megabyte) terms [3], and implemented search strategies that are effective even for very large search spaces [4].

The problem we address in this paper is how to manage the storage of very large search spaces in the first place. The problem exists because the space searched by a declarative debugger is equivalent to a complete record of every step of the execution of a program, and given today's CPU speeds, describing the actions of a program that runs for just a second or two requires gigabytes of storage. This makes storing the record in its entirety clearly infeasible. We must store only a part, and recreate the other parts on demand. But how should the system decide exactly what parts of the record to materialize when? We give some algorithms for making that decision in sections 4 and 5, together with some experimental evaluation. But first, section 2 introduces the Mercury declarative debugger, and section 3 gives our general approach. We assume familiarity with standard logic programming terminology.

2 Background: the Mercury declarative debugger

When the Mercury compiler is asked to generate debuggable code (with a flag similar to gcc's -g), it includes callbacks to the runtime system at significant events in the program. These events fall into two categories: interface events and internal events. Interface events record transfers of control between invocations of predicates and functions. While Mercury supports functions as well as predicates, the distinctions between them are only syntactic, so we call each predicate or function a *procedure*. (Strictly speaking, it is possible for a predicate or function to have multiple modes of usage, and a procedure corresponds to just one mode of a predicate or function, but this distinction is not relevant to this paper.) There are five kinds of interface events, the first four of which correspond to the four ports of the Byrd box model [2]:

`call` A call event occurs just after a procedure has been called, and control has just reached the start of the body of the procedure.

`exit` An exit event occurs when a procedure call has succeeded, and control is about to return to its caller.

`redo` A redo event occurs when all computations to the right of a procedure call have failed, and control is about to return to this call to try to find alternative solutions.

`fail` A fail event occurs when a call has run out of alternatives, and control is about to return to the rightmost computation to its left that has remaining alternatives which could lead to success.

`excp` An exception event occurs when control leaves a procedure call because that call has thrown an exception.

There are also internal events which record decisions about the flow of control, but these are not important for this paper.

When a Mercury program that was compiled with debugging enabled is run under the Mercury debugger mdb, the runtime system gives the debugger control at each of these events. The debugger can then decide to interact with the user, i.e. to prompt for and accept commands, before giving control back to the program being debugged [14]. The mdb command set provides all the usual debugger facilities, e.g. for inspecting the values of variables and setting breakpoints. It also allows the *retry* of the current call or any of its ancestors. The retry resets the program to the state it had at the time of the `call` event of the selected procedure invocation. This is possible because in Mercury there are no global variables that the call could have modified, and we have I/O tabling [11, 13] to simulate the undoing of any interaction of the call with the outside world.

Retry capability is very useful in its own right, but it is also crucial in the implementation of declarative debugging. Users can give the command to initiate declarative debugging when execution is at an `exit` event that computed a wrong solution, when execution is at the `fail` event of a call that did not compute all the solutions it was expected to compute, or when execution is at the `excp` event of a call that was not expected to throw that exception. In all three cases, the Mercury declarative debugger uses the retry mechanism to reexecute the affected call, but this time the code executed by the runtime system at each event has the task of building a record of all the events of the call. We call this record the *annotated trace* [1]. When execution arrives at the event at which declarative debugging was initiated, the annotated trace is complete, and the system invokes the declarative debugging algorithm.

That algorithm searches a tree called the *evaluation dependency tree* or EDT (That name is from [8], but the tree is an instance of the scheme proposed by Naish [5]). Each node in the EDT corresponds to an `exit`, `fail` or `excp` event in the trace. Each of these nodes also makes an assertion: that the solution represented by an `exit` event is correct, that

the set of solutions returned before a `fail` event is complete, or that the exception thrown at an `excp` event was expected to be thrown. The children of a given node N in the EDT are the `exit`, `fail` and `excp` events generated by child calls made by the procedure invocation represented by node N which could have affected the correctness of the assertion made by N. The declarative debugger searches the EDT for a incorrect node whose children are all correct: such nodes represent bugs.

The declarative debugger constructs the EDT from the annotated trace on demand. The reason why we don't build the EDT directly is that we need to build different EDT fragments for negated goals than for non-negated goals, and the condition of an if-then-else is a negated goal only if the condition fails. We therefore wouldn't know what kind of EDT to build until it is too late. Building a more general data structure such as the annotated trace allows us to avoid this problem [1].

Besides the heap space used by the program under normal conditions, there are two additional memory costs when the annotated trace is being built. One cost is that each node in the annotated trace consumes a few words of memory; the number of words depends on the node type. The other cost comes about because some of our search algorithms need to know the values of procedure arguments. We therefore include copies of the call's input arguments in each `call` node and copies of the call's output arguments in each `exit` node. The copied values may be (and usually are) pointers to the heap. These references prevent the garbage collector from recovering the heap cells reachable through those pointers. This doesn't add to memory consumption *directly*, but the indirect effect on memory requirements is very significant. The exact amount of heap memory retained by e.g. a specific `call` node is impossible to predict, but on average, the amount of heap memory retained by n events is usually linear in n. This is because (1) Mercury programs can only execute straight line code between events, so the amount of memory allocated between two events is bounded for any given program, and the average doesn't even vary very much between programs; and (2) the rate of recovery of heap cells must roughly match the rate of their allocation if the program is not to run out of memory. The memory overhead imposed by collecting the annotated trace is thus broadly linear in the number of nodes and the ratio can be measured for any particular program run. This allows us to control the memory overhead of the annotated trace by controlling the number of nodes in the annotated trace.

```
build_annotated_trace(call_number, end_event, depth_limit) returns trace is

trace := NULL
inside := false
Rewind execution to a call before or equal to call_number
For each executed event e loop
    If e is a call or redo event for call call_number
        inside := true
    If inside
        If depth(e) < depth_limit
            trace := create_annotated_node(e, trace)
        Else if depth(e) = depth_limit and e is an interface event
            trace := create_annotated_node(e, trace)
            If e is an exit, fail or excp event
                trace := mark_as_implicit_root(e, trace)
    If e is an exit, fail or excp event for call call_number
        inside := false
Until the event number of e is end_event
```

Fig. 1. Algorithm for building the annotated trace to a predefined depth limit.

3 Rematerialization on demand

To generate an annotated trace for a call, the call must be reexecuted and the resulting events collected. Not all the events need be collected though. We may collect a subset of the events generated by the call and ask the declarative debugger to try to find a bug in one of these. If the declarative debugger needs to explore events not collected the first time around, then the missing events can always be added by reexecuting the appropriate call. (Reexecuting a deeper call will require less time.)

On each reexecution of a call we require the set of events gathered during that run to form an EDT. For each node in the EDT derived from a generated portion of the annotated trace, we require that either all the children of the node are present in the annotated trace, or none of them are present. If none of them are present then we mark the node as an *implicit root*. An implicit root is the root of a subtree in the EDT whose nodes have not been materialized in the annotated trace.

If the declarative debugger needs to search the nodes in an implicit subtree, the call corresponding to the `exit`, `fail` or `excp` event at the implicit root must be reexecuted. To do this we use the debugger's retry capability to rewind the state of the program to a point just before the `call` event corresponding to the `exit`, `fail` or `excp` event at the root of the implicit subtree. We then proceed to reexecute the program from that point, gathering events into the annotated trace, until we arrive at the `exit`, `fail` or `excp` event at the implicit root.

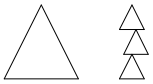

Fig. 2. The EDT corresponding to the whole annotated trace (left), and the EDT fragments corresponding to the parts of the trace materialized on demand (right)

The first version of the algorithm we use to decide which events should be added to the annotated trace on a given run is depicted in figure 1 (this algorithm is from [1]). Here the *depth_limit* parameter controls the depth of each generated portion of the annotated trace, or more precisely, the depth of the EDT represented by the generated portion of the annotated trace. The depth function returns the depth of an event relative to the root of the portion of the EDT currently being materialized. Initially *end_event* will be the event where the user started the declarative debugging session. On subsequent invocations *end_event* will be the event at the root of an implicit subtree we wish to materialize. *call_number* is the call sequence number of the call corresponding to the event at the root of the implicit subtree.

The manipulation of *inside* ensures that we collect events only from inside the call tree of the selected call. Of the events that pass this test, the algorithm includes in the annotated trace all events *above* the depth limit, only interface events *at* the depth limit, and no events *below* the depth limit. Given a large EDT such as the one on the left in figure 2, successive invocations of this algorithm materialize annotated traces that yield EDT fragments whose relationship is shown by the triangles on the right of that figure.

When the event with event number *end_event* is executed, the new annotated trace fragment is complete. If this is the first, topmost fragment, the declarative debugger starts running the search algorithm on it, converting nodes in the annotated trace into nodes of the EDT on demand. If the search algorithm wants to explore a part of the search space beneath an implicit root, it will invoke `build_annotated_trace` again. When it returns, the debugger will link the new fragment into the EDT at the point of the implicit root. Our representation scheme allows the declarative debugger to view the EDT as a single whole tree whose nodes are materialized on demand, not as something stitched together from several fragments.

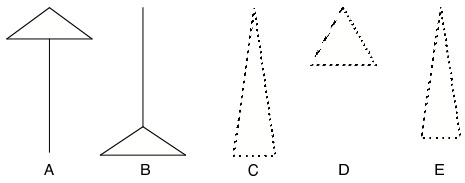

Fig. 3. The shape of the trees for "bigsmall" and "smallbig", and their approximations

4 Ineffective heuristics

The simplest way to control the space-time trade-off is to give a fixed value for *depth_limit* in figure 1. The problem with this solution is that it is impossible for the declarative debugger implementor to give a value for *depth_limit* that yields acceptable results in most cases, let alone all cases. This is because the EDTs of different programs have greatly different average branching factors. It is possible for a given depth setting to cause the declarative debugger to try to collect infeasibly many events for one program, while collecting only a handful of events for another program, requiring a huge number of reexecutions of the program to construct the required parts of the EDT.

A practical algorithm must therefore make *depth_limit* a function of the shape of the implicit tree we wish to materialize. Initially nothing is known about the shape of the search space. We therefore initially give `build_annotated_trace` a small fixed depth limit (currently five), but make it record information in each node at the bottom edge of the first fragment about the implicit subtree rooted there. This allows us to give better depth limits to the invocations that build lower fragments. Our ideal depth limit is one which will cause no more than a specified number of nodes, *node_limit*, to be included in the new trace fragment, since (statistically) this also bounds the memory required by the new fragment. For now, *node_limit* is a parameter; later, we will look at what values of this parameter are sensible.

Average branching factor. It is easy to modify the code in figure 1 to detect when execution enters and leaves an implicit subtree, to record the maximum depth of each subtree (d_{max}) and the numbers of calls (C) and events (E) in each subtree, and to record this data in the root of each implicit subtree. While this info doesn't tell us about the implicit EDT directly, it tells us about a related tree we call the *weighted call tree*. We can think of the annotated trace as a weighted tree where each node corresponds to a `call` event, and the weight of a node is the number of events that have the same call sequence number as the node, including

internal events. Since the sum of the weights of all the nodes equals the number of annotated trace nodes represented in the tree, this tree is useful for modeling memory consumption.

For a weighted call tree with a constant branching factor b, the same weight for each node w, and maximum depth d_{max}, the number of events represented by the tree, N, is given by $N = w \sum_{i=0}^{d_{max}-1} b^i$. In this case, $N = E$, and we can compute the average w as E/C. This makes it easy to solve for b by applying e.g. Newton's method to the monotonic function $f(b) = w \sum_{i=0}^{d_{max}-1} b^i - N$ to find b where $f(b) = 0$. This gives us the average branching factor of the implicit subtree. Assuming the branching factors of most nodes are close to this average, we can calculate the depth limit we need to collect about *node_limit* nodes by calculating the root of the monotonic function $g(d) = w \sum_{i=0}^{d_{max}-1} b^i - node\_limit$.

Unfortunately, our benchmarking shows the assumption is often very far from the truth. We used four synthetic benchmark programs: fib, stick, bigsmall and smallbig, and one real one: the Mercury compiler mmc. For now, we ran all five on data that yielded 1 to 25 million events. Our benchmark machine was a PC with a 2.4 GHz Intel Pentium IV with 512 Mb of RAM running SuSe Linux 8.1. The behavior of the heuristic is ok on the test program whose tree is a thin (predictable) stick and isn't *too* bad for Fibonacci, though for fib it collects 10 to 60 times as many nodes as intended due to the difference between the average branching factor 1.55 and the actual branching factor of 2 for all non-leaf nodes. However things are worse for the other benchmarks. Many programs contain components with characteristics similar to bigsmall or smallbig, whose tree shapes are shown in trees A and B respectively in figure 3. For smallbig, there didn't appear to be any correspondence at all between *node_limit* and the average number of nodes actually constructed per reexecution. With *node_limit* = 20 000, we got about 150 000; with *node_limit* = 100 000, we got about 35 000. Since the tree suddenly gets exponentially large in the bottom few layers, the computed average branching factor needs to be only a little bit too optimistic for the number of nodes constructed to explode. How optimistic the approximation is depends on how close to the widening point of the tree the relevant approximation is taken from. That in turn depends on the depths used for the fragments above in an essentially unpredictable manner. For bigsmall, the computed *depth_limit* is so bad that trying to collect only 100 nodes ran our test machine out of memory, and the same thing happens with mmc with *node_limit* = 2 000.

Biased branching factor. The problem with bigsmall arises because the approximation tree (C in figure 3) is way too deep and narrow: it doesn't

have the same shape as the actual tree, even though it has the same maximum depth and number of nodes. Based on this, the heuristic believes that most nodes are near the bottom and thus it is safe to collect many levels at the top, but those levels contain far more events than expected.

Even though smallbig and bigsmall have very different shapes, they are approximated by the same tree, tree C in figure 3. We could fix this by approximating each implicit tree with a constant-branching-factor tree of the same *average* depth. The smallbig and bigsmall examples would then be approximated by trees D and E respectively in figure 3.

Calculating the average depth of an implicit tree is almost as simple as calculating the maximum depth; we now solve for β in $N = w \sum_{i=0}^{d_{ave}-1} \beta^i$ where d_{ave} is the average depth of the nodes in the tree. This approach does perform better. For stick and fib, the numbers of nodes collected are much closer than the number asked for by *node_limit*, though there are still some significant deviations. However, the new estimates are still far from perfect. The bigsmall and mmc tests still collect far too many nodes, though with this heuristic they run out only with *node_limit* = 2 000 for bigsmall and *node_limit* = 200 000 for mmc. The smallbig tests still suffers from exactly the same problem as before: the numbers of nodes collected still bears little relationship to *node_limit*, though the chaotic pattern is different.

5 An effective strategy

Clearly, approximating an implicit subtree using a tree with a constant branching factor is not a useful approach, since realistic programs do not behave this way. Real programs call all sorts of different predicates. Some are simple recursive predicates which produce stick-like trees; some have long conjunctions in their bodies which produce wide trees with large branching factors. We need a heuristic that works with both these shapes and everything in between. The heuristics of the previous section were also flawed in that their estimates of *depth_limit* could inherently fail in either direction: they could try to collect too many levels as well as too few. While trying to collect too few levels is relatively benign, requiring only a small increase in the number of reexecutions, our benchmarking shows that trying to collect even a few too many levels can require far more memory than is available. We therefore want a heuristic that guarantees that no more than *node_limit* nodes will be added to the annotated trace. The ideal value for *depth_limit* is the highest value that has this property.

Our *ideal depth strategy*, whose algorithm is shown in figure 4, is designed to compute this value directly. When processing events we don't link into the new trace fragment, we don't just record their maximum or

```
build_annotated_trace(call_number, end_event, node_limit, depth_limit)
returns trace is

trace := NULL
inside := false
Initialise the counts array to size ⌊node_limit/2⌋, all zeros
Rewind execution to a call before or equal to call_number
For each executed event e loop
    If e is a call or redo event for call call_number
        inside := true
    If inside
        If depth(e) < depth_limit
            trace := create_annotated_node(e, trace)
        Else if depth(e) = depth_limit and e is an interface event
            trace := create_annotated_node(e, trace)
            If e is an exit, fail or excp event
                ideal_depth := calculate_ideal_depth(counts, node_limit)
                trace := mark_as_implicit_root(e, ideal_depth, trace)
                Reset counts to all zeros
        Else
            depth_in_implicit_subtree := depth(e) - depth_limit
            If depth_in_implicit_subtree ≤ ⌊node_limit/2⌋
                Add 1 to counts[depth_in_implicit_subtree]
    If e is an exit, fail or excp event for call call_number
        inside := false
Until the event number of e is end_event
```

Fig. 4. Algorithm for building the annotated trace using the ideal depth strategy.

average depth; we build a more detailed record. The algorithm does this by building an array, *counts*, that records the number of events at each depth in the tree below the current depth limit for any given implicit subtree. The `calculate_ideal_depth` function scans this array, incrementing the depth and computing the cumulative number of events at or above the current depth until it gets to a depth at which this total exceeds *node_limit*, then returns one less than this depth as *ideal_depth*. (If the subtree contains fewer than *node_limit* nodes, then there is no such depth, and we return a depth that causes all those nodes to be included in the fragment.) We attach the ideal depth of each subtree to the node that acts as the root of that subtree. We specify $depth\_limit = 5$ for the first invocation of `build_annotated_trace`, as before. However, later invocations, whose task is to build an annotated trace fragment from an implicit root at the bottom edge of a previous fragment, will be given as *depth_limit* the recorded ideal depth for the subtree at that node. This guarantees that we collect as many nodes as we can without going over *node_limit*.

node_limit	*exec_count*	*total_created*	$\frac{total\_created}{exec\_count}$	T_U	T_R	RSS	VSZ
100	1124	102 910	91	35.36	51.60	41	49
500	210	101 826	484	10.63	13.67	41	49
1 000	105	102 188	973	7.81	9.40	41	49
5 000	23	112 432	4 888	5.66	6.17	41	49
10 000	12	116 280	9 690	5.45	5.81	41	49
50 000	4	173 876	43 469	5.56	5.80	51	65
100 000	2	165 514	82 757	5.85	6.10	50	57
500 000	1	262 130	262 130	7.28	7.53	59	74
1 000 000	1	524 124	524 124	9.65	9.93	90	99

Table 1. bigsmall: ideal depth strategy.

Since when we materialize the subtree at an implicit root we will wish to collect at most *node_limit* events, it suffices to count events down to a depth of $\lfloor node\_limit/2 \rfloor$. This is because the minimum number of events at each depth is two (a `call` event and its corresponding `exit`, `fail` or `excp` event). We can reuse the same array to calculate the ideal depth for all the implicit subtrees encountered during a particular run. Reserving $\lfloor node\_limit/2 \rfloor$ words of memory for this purpose is not a problem, since we are clearly willing to have the new fragment occupy space linear in *node_limit*. The array just increases the constant factor slightly.

Materializing the subtree of a predicate that may succeed more than once requires a slight variation on our algorithm. Suppose a predicate succeeds twice, producing a `call`/`exit` pair and a `redo`/`exit` pair. The subtree rooted at the second `exit` node can contain events both from between the `call`/`exit` pair and from between the subsequent `redo`/`exit` pair (consider a child call whose result is used in both solutions). The algorithm in figure 4 resets *counts* at each `exit` event, which means the ideal depth limit stored at the second `exit` will be too big because it is based only on the events between the `redo` and second `exit`. To fix this, we can reexecute the call in question using a modified version of the algorithm in figure 4 which doesn't reset the *counts* array and doesn't construct any trace nodes. (Since most calls can succeed at most once, this extra reexecution will be required only rarely.) The usual `calculate_ideal_depth` function at the second `exit` node will then compute the right ideal depth. In practice we have so far found this modification unneccessary, even for programs that use significant backtracking.

Table 1 gives experimental results for the most problematic of our small programs, bigsmall. Tables 2 and 3 do the same for the Mercury compiler. Table 2 shows the compilation of a small module, while table 3 shows the compilation of a large 6 000+ line source file, a process that generates more than 200 million events. Each test simulates a declarative

node_limit	*exec_count*	*total_created*	$\frac{total\_created}{exec\_count}$	T_U	T_R	RSS	VSZ
1 000	83	59 460	716	19.27	21.14	98	126
5 000	42	173 383	4 128	18.76	19.89	106	134
10 000	31	265 019	8 549	18.93	19.83	115	142
50 000	11	507 947	46 177	18.39	18.93	150	176
100 000	7	640 189	91 455	19.48	19.98	157	184
500 000	2	911 521	455 760	19.52	19.97	200	226
1 000 000	1	913 087	913 087	24.19	24.70	246	268

Table 2. Mercury compiler compiling small module, ideal depth strategy.

node_limit	*exec_count*	*total_created*	$\frac{total\_created}{exec\_count}$	T_U	T_R	RSS	VSZ
1 000	56	26 921	480	274.48	277.48	205	233
5 000	31	86 131	2 778	225.00	226.96	201	225
10 000	25	162 232	6 489	214.26	215.92	206	233
50 000	15	583 876	38 925	186.18	187.54	254	284
100 000	13	1 034 614	79 585	186.63	188.03	311	334
500 000	7	2 969 020	424 145	174.98	190.41	477	542
1 000 000	6	5 130 084	855 014	193.08	684.92	443	866

Table 3. Mercury compiler compiling large module, ideal depth strategy.

debugging session using the divide-and-query search strategy [12]. The search starts at the top node of tree (the `exit` node of `main`), and since our testing harness automatically answers 'no' to all questions, it ends at a leaf node. (For our largest test, finding this "bug" required 22 questions.) This is the kind of search that puts the most stress on our algorithm since it requires the most reexecutions of the program.

In each table, the only parameter is *node_limit*, the upper bound on the number of nodes that we want to collect for the annotated trace fragment built by each reexecution. We do not include the initial or final reexecutions in the shown measurements, since the initial reexecution uses a small constant depth limit (since nothing is known about the tree at this time) and the size of the fragment built by the final reexecution is limited by the size of the subtree, not *node_limit*. *total_created* is the total actual number of annotated trace nodes which were produced during the complete debugging session (except for the first and last reexecutions). *exec_count* is the number of reexecutions required to locate the bug (again minus the first and last reexecutions). *total_created*/*exec_count* gives the actual average number of nodes collected per reexecution, which we would like to be less than *node_limit* but otherwise as close to it as possible. The T_U and T_R columns show the user CPU time and the real (wall clock) time required for the tests in seconds; the times were averaged over ten runs. The last two columns show (in megabytes) the total resident set size (RSS) and the total virtual size (VSZ) of the process (including swap

space) at the time when the bug is located, which is when they are at their maximum. All this data is available, in more detail and for more values of *node_limit*, for all our benchmarks and all our heuristics in [3], though we have improved our system since that earlier work.

The results show that the extra calculation required to compute the ideal depth limit (instead of estimating it) is well worth it. For all our benchmarks, including the ones not in the tables, we get more than acceptable performance for a wide range of *node_limit* values, with values in the 10 000-100 000 range generally performing best. Having *node_limit* much lower wastes time in too many reexecutions; having *node_limit* much higher runs the risk of running out of memory. (The last row of table 3 shows the start of thrashing.) We have found *node_limit* = 20 000 to work well for all the programs we have tried. For example, when the Mercury compiler is invoked on that 6 000 line source file, our algorithm needs about three and a half minutes to materialize all the fragments needed to find the "bug" in a leaf node. During this time, the search algorithm asked the oracle 22 questions. If it were the user answering these questions, there would be on average about an 8 to 10 second delay between his/her answer and the next question. Given that the user will certainly take much more than 10 seconds to answer each query, the overhead of search space materialization is not the bottleneck in the search for the bug.

With *node_limit* = 50 000, the sizes of fragments tend to be in the 5-25 Mb range, both for the compiler and some other programs we have looked at. When the cumulative sizes of the fragments materialized so far starts to exceed the available memory, it would be relatively straightforward to release the memory of the least recently used fragment. The EDT nodes constructed from it would remain, and if the search algorithm ever needs the other nodes from that fragment, it could construct the fragment again.

On any given reexecution of part of the program, most events end up being ignored. It is therefore important to optimize the handling of these events. This is why the we use a simple depth cutoff as the criterion for inclusion in a new fragment. Other criteria may lead to fragments that have a higher proportion of nodes useful to the search algorithm (whichever one is being used), but this is unlikely to compensate for the sharply greater cost of evaluating the test of any nontrivial criterion.

6 Related work

Nilsson and Fritzson [6, 7] also propose constructing the program trace piece by piece. They introduce the concept of the *query distance* to a

node. This is the number of questions required to get to the node using a top-down, left-to-right search.

They optimistically materialize nodes and then uses the query distance to decide which nodes should be discarded if memory usage becomes too high. Nodes with higher query distances are the first to be discarded.

This works well for top-down search, since most of the time the next question will be in a materialized fragment of the EDT. This technique doesn't work with the Mercury declarative debugger because it can use multiple search strategies (including a version of Shapiro's divide-and-query [12] algorithm and a search strategy similar to Pereira's rational debugger [9]). With these search strategies [4] the query distance ceases to become a useful heuristic for deciding which nodes to throw away.

Modifying the notion of the query distance to work for different search strategies is not a viable option, since it is unclear how the modified query distance could be efficiently calculated for search strategies like divide-and-query. Also, a node may have a small query distance for one search strategy and a large query distance for another strategy. Since the user may switch search strategies mid-session, the query distance heuristic doesn't help to decide whether to keep such a node.

There is also a penalty to be paid for first creating a node in the EDT and then disposing of that node later when resources become tight (mostly due to the extra work garbage collection must do). Using our method we are able to know *ahead of time* how much of the EDT can be viably generated in a single reexecution, so no nodes are created only to be destroyed later.

Plaisted [10] proposed an efficient method for deciding which nodes in the EDT to materialize. Unfortunately the method only works with the divide-and-query search strategy, and generates questions that are harder for users to answer.

7 Conclusion

During our work on the Mercury declarative debugger we found that we needed a new algorithm to control the resources consumed by the annotated trace, because none of the techniques in the current literature were adequate in the presence of multiple search strategies.

We first tried two variations on a method that tries to guess the shape of the subtree to be constructed from information about its branching factor. These methods don't work, because their implicit assumption that most nodes have similar branching factors is much too far from the truth.

Analyzing the cause of the failure led us to the ideal depth strategy. While this strategy uses a bit more memory, it allows us to calculate *ex-*

actly how much of the search space we can viably materialize each time the search algorithm visits a previously unexplored part of the search space. We have found this algorithm to work very well in practice, so well that users of the Mercury declarative debugger spend much more time answering questions than waiting for rematerialization, even when debugging long running, real programs. Our algorithm thus helps programmers find bugs more quickly.

The techniques we presented are certainly not specific to Mercury. They can be applied to any declarative debugger with a tree that must be searched and an execution replay mechanism that can rebuild previously unmaterialized parts of the tree.

We would like to thank the Australian Research Council and Microsoft for their support.

References

1. Mark Brown and Zoltan Somogyi. Annotated event traces for declarative debugging. Available from http://www.cs.mu.oz.au/mercury/, 2003.
2. Lawrence Byrd. Understanding the control flow of Prolog programs. In *Proceedings of the 1980 Logic Programming Workshop*, pages 127–138, Debrecen, Hungary, July 1980.
3. Ian MacLarty. Practical declarative debugging of Mercury programs. MSc thesis, University of Melbourne, July 2005.
4. Ian MacLarty, Zoltan Somogyi, and Mark Brown. Divide-and-query and subterm dependency tracking in the Mercury declarative debugger. In *Proceedings of AADEBUG 2005*, Monterey, California, September 2005.
5. Lee Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), April 1997.
6. Henrik Nilsson. Tracing piece by piece: affordable debugging for lazy functional languages. In *Proceedings of ICFP '99*, pages 36–47, Paris, France, September 1999.
7. Henrik Nilsson and Peter Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3):337–370, July 1994.
8. Henrik Nilsson and Jan Sparud. The evaluation dependence tree as a basis for lazy functional debugging. *Automated Software Engineering*, 4(2):121–150, April 1997.
9. Luis Moniz Pereira. Rational debugging in logic programming. In *Proceedings of ICLP '86*, pages 203–210, London, England, June 1986.
10. D. A. Plaisted. An efficient bug location algorithm. In *Proceedings of ICLP '84*, pages 151–158, Uppsala, Sweden, July 1984.
11. M. Ronsse, K. de Bosschere, and J.C. de Kergommeaux. Execution replay and debugging. In *Proceedings of AADEBUG 2000*, Munich, Germany, 2000.
12. Ehud Y. Shapiro. *Algorithmic program debugging*. MIT Press, 1983.
13. Zoltan Somogyi. Idempotent I/O for safe time travel. In *Proceedings of the AADEBUG 2003*, Ghent, Belgium, September 2003.
14. Zoltan Somogyi and Fergus Henderson. The implementation technology of the Mercury debugger. In *Proceedings of WPLE '99*, pages 35–49, Las Cruces, New Mexico, November 1999.