Deep profiling: engineering a profiler for a declarative programming language

Thomas C. Conway and Zoltan Somogyi

{conway,zs}@cs.mu.OZ.AU
Department of Computer Science and Software Engineering
University of Melbourne, 3010 Victoria, Australia
Phone: +61 3 8344 9100, Fax: +61 3 9348 1184

Declarative programs differ from imperative programs in several respects, the main ones being their heavy use of recursion, of various forms of polymorphism, and of higher order. Existing profilers tend not to produce very useful information in the presence of these constructs. We present a new profiling technique we call *deep profiling* that yields detailed and useful information about programs even in the presence of these constructs, information that is significantly richer than the output of other profilers.

The heart of deep profiling is a source-to-source transformation. We have implemented this transformation and its associated infrastructure in the compiler for Mercury, a purely declarative logic programming language. While our measurements of this implementation show that deep profiling has slightly greater overhead than some other profiling techniques, the wealth of information it provides makes this extra overhead worthwhile. The deep profiling algorithms themselves are applicable to most other language styles, including imperative, object-oriented, and functional languages.

1 Introduction

The quote "premature optimization is the root of all evil", often attributed to Knuth, contains good advice. Once programmers have a performance-aware design, i.e. one that does not needlessly perform expensive operations, it is usually best if they forget about performance while writing code, returning to the topic only when the program has been fully debugged.

Worrying about performance during coding is usually counter-productive, for several reasons. One reason is that code that has been hand-optimized is harder to debug; "debug and then optimize" is easier work than "optimize and then debug". However, "debug and then don't optimize" is easier still, and often it is the appropriate path to take. Most parts of most programs are not worth optimizing; they contribute so little to the cost of running the program that even making them run infinitely fast would not be worthwhile. Only the parts of the program that do contribute a significant fraction of the cost of running the program (the "hot spots") are worth optimizing.

Experience has shown that programmers, when asked to guess where the hot spots are in code that they have written, usually guess wrong: many of the places they guess to be hot spots turn out not to be, and they miss many hot spots that do exist. They therefore need software tools (*profilers*) that will find out where the program is actually spending its time when processing some typical inputs.

Over the years, researchers and software engineers have designed and implemented many different profiling systems. The variety of existing profiling systems is driven mainly by the intense conflict at the heart of profiler design: a good profiler should present to the programmer as much information relevant to the performance of the program as possible, but the more information a profiler gathers, the more the gathering process itself interferes with the performance of the program, and thus the less valid the gathered information is. This conflict has driven designers to build profilers that gather only the minimum necessary information, no more. Profiler designs differ because what information is necessary depends on many factors, including programming language, programming style and program size.

Declarative programming languages pose new challenges for profilers because they introduce new programming styles. These include extensive use of recursion including mutual recursion, intensive reliance on various forms of polymorphism, and frequent use of higher order constructs. All these existed or at least could be simulated in imperative or object-oriented languages, but as long as they were not typical and were rarely used in performance critical code, profiler designers could mostly ignore them. They could design profilers as if (mutual) recursion, polymorphism and higher order calls did not exist, and make the minimum concession to their existence in the implementation: they could ensure that the profiler does not crash on programs containing such code without making the profiler gather the information that the programmer would need to optimize program fragments containing such code. Designers of profilers for declarative programming languages do not have that option. Since declarative languages do not have iteration, no program fragment can be performance critical without being recursive (possibly mutually recursive), and the pervasive use of polymorphism and higher order code implies that performance critical code will likely use those constructs as well. A profiler for a declarative language must therefore make sure that the information it gathers and presents to the programmer will allow hot spots to be located and diagnosed even in the presence of (mutual) recursion, polymorphism and higher order code.

In this paper, we present a new profiling technique called *deep profiling* that we have designed to meet this requirement. It gets its name from its chief design feature, the association of a deep context (effectively the entire list of ancestors of the current function or predicate) with each profiling measurement. Recording this much information at an acceptable cost was a considerable engineering challenge, as was finding a way to display this much information without confusing and distracting the programmer. Actually, we have found that the information gathered by the profiler can be used not only for performance improvement, but also for elucidating the typical behaviour of the program to maintainers, programmers who are unfamiliar with the structure of a program either because they were not involved in its initial implementation or because the program has changed significantly since the last time they looked at it. In addition, our profiler (like many others) can be used for coverage testing: finding parts of the program that have not been executed at all and in which therefore undetected bugs may still be lurking.

Our proof-of-concept implementation of deep profiling is `mdprof`, a deep profiler for Mercury. Mercury is a new, purely declarative logic programming language designed for the construction of large, reliable software systems by teams of programmers [2, 10, 18], the kinds of programs that typically need to be profiled. Mercury's syntax is similar to the syntax of Prolog, but Mercury also has strong module, type-, mode- and determinism systems, which catch a large fraction of programmer errors and enable the compiler to generate fast code. Thus programming in Mercury feels very different from programming in Prolog, and much closer to programming in Haskell or in safety-oriented imperative languages such as Ada or Eiffel.

We assume familiarity with the basic concepts of logic and functional programming, and with Prolog syntax, which is also Mercury syntax. We do not assume familiarity with other aspects of Mercury beyond the following.

– Mercury is strongly typed. Its type system is Hindley-Milner, and similar to the type systems of Haskell and ML.
– Mercury supports functions as well as predicates. Every predicate and every function has one or more *modes*, which say, for every argument, whether it is input or output. (The mode system is more sophisticated than that, but that is all that matters for this paper.) We call each mode of a predicate or function a *procedure*.
– Mercury associates with every procedure a *determinism*, which expresses upper and lower bounds on the number of solutions the procedure can have. Procedures that are guaranteed to have exactly one solution have determinism *det*. Procedures that are may have one solution for one set of inputs and zero solutions for other inputs have determinism *semidet*. Procedures that may have any number of solutions have determinism *nondet*.

The structure of this paper is as follows. Section 2 lists the main questions that the designer of a profiling system must answer, together with the answers used in existing profilers. Section 3 lists the requirements imposed on a profiler by the design of the Mercury language and by the structure of typical Mercury programs. These issues are shared not only by other logic programming languages but also by functional languages, and, to a lesser extent, by some object-oriented language as well. Sections 4 and 5 are the core of the paper: they give a detailed description of deep profiling, the profiling system we have designed to meet the requirements listed in section 3. It should be adaptable to any language that has the same list of requirements (since lazyness adds extra requirements, adaptation to lazy functional languages would require further work). Section 4 shows how deep profiling instruments the program to associate a deep context with each measurement at acceptable cost. It also discusses tradeoffs in our chosen approach, and outlines possible approaches we have decided not to take. Section 5 shows how data gathered during profiling runs is postprocessed and presented to the programmer. Section 6 presents experimental data on the costs of deep profiling: the overhead it imposes on the programs being profiled and the cost of running the profiling tool itself. Section 7 then compares deep profiling with other profiling systems described in the literature, including cost centre stack profiling [11] and call path profiling [8, 9].

2 Dimensions of profiler design

Many different profiling systems exist for a variety of languages and platforms. These systems can be usefully compared by examining the following characteristics.

What is being measured. The most common forms of profiling measure how much time is spent in each part of the program. However, profiling may also measure memory-related statistics, such as the number of memory allocations in each part of the program, the amount of memory allocated by each part of the program, or the amount of memory live during the execution of each part of the program. It may also count the number of times each part of the program is executed, either focusing only on some kinds of constructs (e.g. branches taken, exceptions thrown, locks contended for, or goals failed (in logic programming languages)) or on a universal basis, at various levels of granularity (e.g. function calls, statement executions). (Some of this information is as useful for coverage testing as for profiling.) Most profilers measure more than one aspect of the program, although few measure *all* these aspects, at least not at the same time.

How much context each measurement has. In the simplest case, each measurement is independent of context; e.g. the profiler may count the number of times function `f` is called, but will not record where each call came from. The profiler may add context to each measurement by recording what function each call to `f` came from. That context can be refined further by noting not just that a call to function `f` came from function `g`, but also that it came from a specific call site (e.g. line 120, column 10). The context can be made more detailed by recording, for each measurement, not only the identity of the parent caller function or call site, but also the identities of *all* the ancestor caller functions or call sites.

How measurements are taken. The two most common ways of collecting profiling data are to instrument the code (which can be done by modifying the source program, using the compiler to generate instrumented code, or by postprocessing the output of the compiler), and to periodically interrupt the program and take samples of the desired data. The sampling approach suffers from the obvious disadvantage that its measurements are not deterministic, but errors resulting from the uncertainty are usually significant only for measurements in infrequently executed parts of the code, and the programmer usually does not care about the performance of such code. The two techniques are often combined in a hybrid approach, in which some measurements are taken by instrumentation and other measurements are taken by sampling. The usual choice is that time is measured by sampling and everything else is measured by instrumentation. (Since most CPU types lack timers with sufficient resolution, which currently means in the region of nanoseconds, a profiler that tried to measure time via instrumentation would not be portable.)

The relative runtime overhead of profiling. Different profiling techniques have different impacts on the running time and memory usage of the program. Some kinds of measurement (e.g. CPU time) are distorted by these overheads. One kind of distortion results from an uneven distribution of the overheads over the measured aspects program execution: for example, the addition of a fixed number of instructions to each function will produce a proportionately bigger increase in the times recorded for small functions than for big functions. Another kind of distortion occurs because profiling overhead alters the interaction of the program with the virtual memory system, with caches, with branch target buffers, etc. In theory, both these kinds of effects can be compensated for. Unfortunately, given the complex pipelined and superscalar architecture and multilevel memory hierarchy of modern machines, and the dearth of published information on their design at the required level of detail, such compensation is usually far from feasible. The only practical way to minimize distortions of the data presented to programmers is to keep the overheads of the profiling process as low as possible.

When the runtime costs of profiling are paid. In many profilers, the costs of profiling are paid only at the events that profiling measures. For example, the cost of counting a procedure call is paid at the procedure call, and the cost of recording a sample for CPU time measurement is paid at the time the sample is taken. However, in profilers that put each measurement in context, the work of setting the contexts may be performed when the contexts change, even if there is no measurement to be made at that point. This reduces overall overhead if the average number of measurements per context change is more than one.

When the information is collated. Before being presented to the user, the raw data collected during the profiling process must be organized and compacted. Some systems just record the raw data during the execution of the profiled data, leaving the data reduction to a substantial postprocessing stage; others perform data reduction during program execution, leaving the postprocessing stage to do little more than formatting the data. The latter kind of system incur more runtime overhead and thus distort the profile more, but in return they do not require storage of the raw data in toto at any time. Given that the raw data can be quite big (tens or hundreds of megabytes), this is a substantial advantage. Hybrid systems preprocess the raw data during runtime but still require some postprocessing. A good tradeoff is to perform processing tasks that reduce the volume of the data at runtime, and to do all other processing tasks after runtime.

When the programmers specify what information they are interested in. In many profilers, programmers never specify this at all; the profiling tool produces a file containing all its data, and programmers look at whatever parts of this file they are interested in. As the number of measurements and the size of the context of each measurement increases, this approach becomes unworkable, because programmers find themselves facing a sea of data they have to summarize themselves. Some profiling tools are therefore interactive; they digest a database of measurements produced by some profiling runs, point out to the programmer the most interesting parts of the database, and then allow programmers to ask queries on the database. Some go even further, and allow or even require programmers to state what parts of the program they are interested in even *before* compiling the program, and perform measurements only on that part. This reduces overhead and therefore distortions in the resulting data, at the cost of requiring recompilation and rerun of the program if the programmers find that they need information about a previously non-profiled part of the program after all.

3 Requirements for Mercury profiling

In this section, we list the requirements placed on a profiler by the characteristics of the Mercury language and Mercury programs. We expect most other eager declarative languages and even many imperative object-oriented languages to have the same or very similar requirements, which means that our profiler design should also be well suited for them. Laziness imposes a significant additional requirement, but we believe deep profiling can be adapted to handle lazy languages as well.

Recursion as a basic language construct. The most obvious issue for profiling declarative languages is that they depend far more heavily on recursion than imperative languages do; in most declarative languages (including Mercury), recursion is the only way to implement repetition. A profiler for an imperative language can usually get away with reporting that e.g. "of the 1000 calls to function `f`, 200 were from function `g`, 100 from function `h`, and 700 from function `f` itself" without also reporting e.g. how many recursive calls each call from `g` gives rise to, or how long the average call to `f` *from g* took. The reason is that most functions written in such languages are not recursive, so the probability that such missing information will cause programmers difficulty in speeding up a performance-critical function is often low enough to be acceptable. Profilers for such languages can thus treat recursion as an exception, in whose presence the correct attribution of costs is a nice bonus but not essential. On the other hand, in declarative languages a function or predicate *cannot* be performance critical without being recursive. Profilers for such languages therefore must treat recursive calls properly, which in almost all cases means treating them as belonging to the same profiling unit as their recursive ancestor. (We discuss an important exception below.)

Mutual recursion. Programs in declarative languages often operate on complex data structures, a significant proportion of which are defined in terms of mutually recursive types. The procedures that process such data structures are therefore often also mutually recursive. Mutually recursive procedures act as each other's extensions, with each procedure containing part of a repetitively executed piece of code. Calls between mutually recursive procedures should therefore be treated like self-recursive calls in most ways, the exception being that we do still want to associate costs incurred directly by a procedure with that procedure, and not just with the clique of mutually recursive procedures as a whole.

Polymorphism of predicates. Tables storing key-value pairs are some of the most common data structures. In most imperative programming languages, a program that needs several different kinds of tables would have a different function for searching each kind of table, either written by the programmer or generated automatically by the compiler (e.g. when using C++ templates). When written in a programming language (such as Mercury) that supports and encourages the use of parametric polymorphism, a single function can perform searches on *all* tables of key-value pairs in the program, regardless of the types of the keys and values.

This poses a problem for profilers. Whereas programmers would typically find it useful to know e.g. the average cost of a call to a function that searches a binary search tree containing e.g. environment variable name - value pairs, they typically would *not* find it useful to know the average cost of a call to a function that searches *all* binary search trees in the program. Even though that function appears to be a single unit in the source code, it is actually an amalgam of several distinct units for the programmer, and the profiler must help the programmer decompose the costs reported for the function into the costs of its distinct uses.

Polymorphism of callers. Unfortunately, it is not sufficient for the profiler to assume that two uses of a function are distinct if and only if the function is called from two different sites. To see why not, consider the Mercury compiler, which is written in Mercury itself. The compiler has several hundred data structures that are tables of key-value pairs, many temporary, some permanent. All these tables are implemented as different instances of the same polymorphic type. This type, `map(K,V)` is an abstract type; its actual definition is private to the `map` module, to allow its definition to be changed without impacting any other modules. Currently, maps are implemented as 2-3-4 trees, and predicates in the map module just call their counterparts in the `tree234` module. (The Mercury compiler and library contain many such "forwarding" predicates; the abstraction they provide makes it easier to revise old design decisions.)

One of the most expensive predicates in the Mercury compiler is `tree234__search`, which performs lookups in 2-3-4 trees. This predicate is expensive because it performs searches on all those several hundred kinds of tables. Despite this fact, it is called from only one place, `map__search`. Clearly, the profiler must use more detailed context information, not just the immediate caller, if it is to correctly distinguish the different uses of `tree234__search`.

Polymorphism within callers. Most profilers make no distinction between different calls to function `f` from function `g`, i.e. they treat the call graph as having just one arc from `g` to `f`. This effectively makes the assumption that all the different calls from `g` to `f` (if there is more than one) have similar enough performance characteristics that the programmer will not mind having their data amalgamated. For programs written in traditional imperative languages, this assumption is usually correct.[1] However, this assumption is much less likely to be correct for programs written in languages that support some form of polymorphism, whether the class-based polymorphism of object oriented languages or the parametric polymorphism of declarative languages.

Consider a program that deals with sets of items (e.g. LR(1) items) with each set assigned a set identifier. Such a program would probably use two maps, one mapping identifiers to sets of items and another one mapping sets of items back to their identifier, so that the program can detect when the result of e.g. a set union is a set that has been seen previously. The procedure that handles insertions into this data structure would call the map insertion function (e.g. `map__insert` in Mercury) twice, once to update each of the two maps. If the maps are implemented as a search tree of some kind, then insertions into the map will require repeated key comparisons, which will be much more expensive for the map in which the keys are sets of items than for the map in which the keys are identifiers. The two calls to `map__insert` will therefore have quite different performance characteristics, which means that the programmer will almost certainly want to be given separate information about each call.

Frequent use of higher order constructs. Programs written in declarative languages use higher order constructs significantly more frequently than programs written in imperative languages. It is therefore much more important for profilers of declarative languages to handle such constructs properly. At the minimum, this means that the profiler must track caller-callee relationships even when a particular caller-callee pair does not appear in the source code. Two desirable extensions are distinguishing first order and higher order calls in the profiler output, and, for higher order calls, distinguishing between closures constructed at different sites that happen to call the same function.

Nested use of higher order constructs. In a program that makes extensive use of higher order constructs, it is likely that some calls to a higher order function will be from descendants of that same higher order function. For example, consider a program working with a genealogical database whose task is to print, for each parent in the database, the parent's name along with the names of all their children. The main program may use `foldl` to call `handle_parent` for each parent in the database, while `handle_parent` itself may use `foldl` to print the names of all the parent's children.

A simplistic profiler would consider `foldl` and `handle_parent` to belong to the same strongly connected component of the call graph, since each calls the other. However, programmers are unlikely to view the program that way; they are much more likely to view the two uses of `foldl` as distinct entities. The two uses cannot be distinguished by type, since both loop over lists of people; they can be distinguished only by the identity of the closure passed to `foldl` or by the location of the top-level call to `foldl` (since the code there determines what closure is passed to `foldl`). What the programmer usually wants is for the profiler to treat the program not as it is, but as it would be if all top-level calls to `foldl` were specialized, i.e. if each call to `foldl` was replaced by a call to a distinct new procedure which is a renamed apart copy of `foldl`. (Some optimizing compilers, including the Mercury compiler, perform exactly this specialization in an attempt to transform higher order calls into more efficient first order calls.)

Bidirectional foreign language interfaces. Like most other non-mainstream languages, Mercury has a foreign language interface (FLI). This interface is bidirectional. On the one hand, the programmer may write some C code instead of some clauses as the implementation of a Mercury predicate or function. On the other hand, the programmer may define a predicate or function in Mercury code and then request that the Mercury compiler generate a C function whose body just calls that predicate or function. Programmers frequently use the foreign language interface to link Mercury code with existing libraries.

One important class of existing libraries are libraries implementing graphical user interfaces. Most of these libraries (e.g. Tk) are based on an event loop that performs callbacks to code

[1] The main reason for this is that programs written in traditional imperative languages tend to have fairly shallow call graphs, and their functions tend to have a single kind of usage, either because the languages lack polymorphism or (e.g. in the case of C++) because template instantiation creates separate copies of functions for different usages.

provided by the client. A Mercury program using the Mercury-Tk interface would start out by constructing some Tk objects. These objects would have associated actions, and these actions would be implemented as Mercury predicates and/or functions which are exported to C code using the FLI. The program would then enter Tk's event loop, which is of course written in C. The event loop would then respond to user actions (key strokes, mouse clicks) by invoking the relevant callbacks on the appropriate objects. These callbacks may cause execution to reenter Mercury code.

Ideally, the profiler for Mercury would understand C code as well, and would be able to treat the Mercury and C parts of the program in a uniform manner. However, given the large differences between the two languages, even deciding what such a profiler *should* do would be quite a complex undertaking. Building such a bilingual profiler would not be cost effective for us, given that we prefer to write all our new code in Mercury, and only use the foreign language interface to link to third party libraries that we have no wish to modify or even (in most cases) to understand beyond the interface level.

However, even a profiler that only understands Mercury must do something sensible with the costs incurred by C code. The only sensible thing to do with such costs is to charge them against the nearest enclosing Mercury procedure that used the foreign language interface to enter C code. Similarly, calls to Mercury procedures from C code should be counted as calls made directly by the nearest enclosing Mercury predicate that used the foreign language interface to enter C code. A Mercury predicate implemented in C can thus be viewed as similar to a predicate that makes higher order calls, in that both have descendants in the call graph that do not appear syntactically in their definition.

4 The deep profiling transformation

This section describes our design for gathering the data that one needs in order to satisfy the requirements we identified in the previous section. We start with a basic system that meets three of those requirements: polymorphism of predicates, polymorphism of callers and polymorphism within callers, but does not handle recursion properly and does not handle higher order calls and foreign language code at all. Section 4.1 describes the data structures of this system, while section 4.2 shows how we build them. The later subsections then extend the system to handle more of our requirements. Section 4.3 shows a technique we could use to handle self recursion while section 4.4 shows a technique we could use to handle mutual recursion within one module. Our profiler doesn't actually use either of those techniques, preferring instead the approach presented in section 4.5 which handles all forms of recursion, even mutual recursion across modules. Section 4.6 shows how we preserve tail recursion in procedures that are not nondeterministic. Section 4.7 then shows how we handle higher order calls, and section 4.8 shows how we avoid unwanted interactions between recursion and higher order calls. Section 4.9 describes our treatment of foreign language code, while section 4.10 describes our handling of exceptions. Section 4.11 describes how we gather profiling measurements, and section 4.12 shows how we save the resulting profiling data. The final system meets all of the requirements we listed in section 3.

4.1 Introducing the main data structures

Our initial profiling system is driven by its demanding requirement of resolving all forms of polymorphism. As we have seen in section 3, giving programmers enough information to let them measure distinct uses of a procedure separately requires associating a significant amount context with each measurement, more than most profilers do. At the very least, it requires associating the caller procedure with each measurement of the callee, but many circumstances of practical importance require additional context in the form of the specific call site within the caller procedure and/or the identity of more distant ancestors.

In theory, we could use a chain of ancestor call sites as the sole context of each measurement. However, call sites don't call other call sites; they call procedures, which in turn contain other call sites. We have therefore found it significantly simpler to use an alternating chain of call site nodes and procedure nodes as context: call site nodes link to procedure nodes and vice versa.

Suppose the program contains five functions: `f`, `g`, `h`, `i` and `j`, where `f` calls `g` and `h` (at call sites `f1` and `f2` respectively), `g` and `h` both call `i` (at call sites `g1` and `h1` respectively), and `i` calls `j` (at call site `i1`). We will want to record measurements for the call chain `f->f1->g->g1->i->i1->j` separately from the measurements for the call chain `f->f2->h->h1->i->i1->j`. This means that the profiler will have to build up a data structure that contains more than one instance of the call site `i1`. In most programs, there will be many procedures (and thus call sites within those procedures) that will appear many times in the tree.

There are several ways one can identify a call site. One can specify them by file name, line number, and column number; one can specify them by file name and byte offset within the file; one can specify them by file name, procedure name, and call number within the procedure. All these identifiers are big enough that we don't want the tree built during a profiling run to contain many redundant copies of them. We therefore build a static data structure for each call site: this data structure describes the call site, and its address serves as a compact identifier of the call site. The set of call sites associated with a procedure is also static information, and thus it is also worthwhile to avoid storing it multiple times. We therefore split procedure nodes into static and dynamic components just as we do with call site nodes.

The main data structure of the profiler is thus a tree (later to be a graph) containing four kinds of nodes, which we call CallSiteStatic, ProcStatic, CallSiteDynamic and ProcDynamic. For now, the fields of these nodes are the following. (Later we will modify the definitions of some fields and add other fields.)

CallSiteStatic CallSiteStatic structures are created by the compiler. There is one CallSiteStatic structure for each call site in the source code.

- `css_id` This field uniquely identifies the call site, and links it back to the source file for the programmer. Any of the representations above will suffice. The choice of representation does not affect the profiler except with respect to details of its user interface, which we do not discuss in that level of detail in this paper.
- `css_callee_id` For now, we will consider this field to contain a pointer to the ProcStatic structure of the procedure called at this call site. In later sections we will also allow it to indicate that the call site represents a higher order call or a typeclass method call (section 4.7) or entry into foreign code (section 4.9).

ProcStatic ProcStatic structures are created by the compiler. There is one ProcStatic structure for each procedure in the source code.

- `ps_call_sites` This field is an array containing the CallSiteStatic structures of the call sites within the procedure.
- `ps_num_call_sites` This field gives the number of call sites within the procedure, and thus the size of the array in the `ps_call_sites` field.

CallSiteDynamic CallSiteDynamic structures are created by the instrumented program during a profiling run. There will be one or more CallSiteDynamic structures for each call site through which the program actually performs a call during the profiling run. For a given call site, there will be distinct CallSiteDynamic structures for each distinct context in which those invocations take place.

- `csd_self_profile` This field contains all the measurements collected for the invocations of the called procedure from the context represented by this CallSiteDynamic structure and all its ancestors, which will be an alternating chain of ProcDynamic and CallSiteDynamic nodes. This chain is linked together with downward pointers in order to allow the graph to be traversed from the root at the end of execution, when the information it contains is saved to a file.
- `csd_callee_ptr` This field contains a pointer to the ProcDynamic node for an invocation of the called procedure, to be used as part of the context for some profiling measurements in the callee's descendants.

ProcDynamic ProcDynamic structures are created by the instrumented program during a profiling run. There will be one or more ProcDynamic structures for each procedure which is called during the profiling run. For a given procedure, there will be distinct ProcDynamic structures for each distinct context in which those calls take place.

`pd_proc_static` This field points to the ProcStatic structure of the procedure that this ProcDynamic structure represents an invocation of.

`pd_call_site_ptrs` This field contains a pointer to an array, that contains a pointer to a CallSiteDynamic structure for each call site in the procedure. The size of the array is given by the `ps_num_call_sites` field of the ProcStatic structure pointed to by the `pd_proc_static` field, and the call site represented by each entry in this array is identified and linked back to the source code by the corresponding entry in the `ps_call_sites` array. If a given procedure invocation does not make a call through a given call site, then the element in the array pointed to by the `pd_call_site_ptrs` field corresponding to that call site will contain a null pointer. Given the precise contexts we use, we expect this will be true for a large fraction of all call sites in each context, which is one reason why we use an array of pointers to structures instead of an array of the structures themselves. However, the main reason is that we want to be able to link to more than one CallSiteDynamic structure from elements of the array that correspond to call sites representing higher order calls, method calls and entry into foreign language code, as we will explain in sections 4.7 and 4.9.

4.2 Building the initial version of the call graph

As we mentioned in section 2, profilers make some of their measurements by sampling. For example on Unix, profilers implement this by asking the kernel to send the program a SIGPROF signal periodically, and recording the measurements in the signal handler function. These measurements should be associated with the context of the code that is being executed at the time the signal arrives. Since the program has no control over the arrival of the signal, the program must maintain at all times a data structure that tells the signal handler what the current context is. The data structure we use in the Mercury profiler is a global variable called `current_call_site_dynamic`, which points to the CallSiteDynamic structure associated with the call site of the procedure invocation currently being executed. To minimize the measurement distortions caused by the program instrumentation we are about to describe, the signal handler ignores profiling signals that arrive while the global boolean variable `inside_instrumentation` is true. Our system sets `inside_instrumentation` to true during the execution of all pieces of instrumentation code except those for which setting and resetting `inside_instrumentation` would be more time consuming than the instrumentation code itself. However, to simplify our exposition and save space we will not show the assignments to `inside_instrumentation`. We will also delay discussion of taking the profiling measurements themselves until section 4.11.

The `current_call_site_dynamic` global variable must be updated whenever control passes from one procedure to another. The standard box model for the execution of logic programs [3] classifies control transfers into four categories: call, exit, redo and fail. (Exit and fail represent successful and unsuccessful returns respectively, while redo represents backtracking back into a call.) Each of these methods of control transfer is called a *port* in logic programming debuggers. [2] To give the profiling system control at each of these ports, the compiler transforms the bodies of all procedures when profiling is enabled in a fashion very similar to what it does when debugging is enabled, except that for profiling, the form of transformation it uses is source-to-source. (For reasons discussed in [17], the debugging transformation is implemented in the code generator.)

Internally, the Mercury compiler considers each procedure to be defined by a single goal; multiple clauses are converted into explicit disjunctions very early on. The general transformation therefore starts with a procedure definition such as

[2] Mercury adds a fifth port as well: control can leave a procedure invocation by raising an exception. However, we will delay consideration of exceptions until section 4.10.

```
p(Args) :-
    Body.
```

and converts it into code of the following form:

```
p(Args) :-
    call_port_code(p_proc_static_ptr, TopCSD, MiddleCSD),
    (
        Body',
        ( exit_port_code(TopCSD, MiddleCSD)
        ; redo_port_code(TopCSD, MiddleCSD), fail
        )
    ;
        fail_port_code(TopCSD, MiddleCSD), fail
    ).
```

Each procedure invocation is concerned with the CallSiteDynamic structures representing the calls to its parent and to its children as well as the call to itself. Each procedure invocation refers to those CallSiteDynamic structures as TopCSD, BottomCSD and MiddleCSD respectively. However, the identity of the call site that e.g. TopCSD refers to depends on which procedure invocation's perspective we adopt. `call_port_code` and `redo_port_code` each shift the perspective from the caller to the callee, while `exit_port_code` and `fail_port_code` shift the perspective from the callee back to the caller.

Each of `call_port_code`, `exit_port_code`, `redo_port_code` and `fail_port_code` updates the global variable `current_call_site_dynamic`. Since `call_port_code` and `redo_port_code` are executed when control transfers from parent to child, they set `current_call_site_dynamic` to point to MiddleCSD, the CallSiteDynamic structure of the call site which created the child procedure invocation (that call site will be in the parent invocation); since `exit_port_code` and `fail_port_code` are executed when control transfers from child to parent, they set `current_call_site_dynamic` to point to TopCSD, the CallSiteDynamic structure of the call site which created the parent procedure invocation (that call site will be in the grandparent invocation). This of course requires the child invocation to know not only the address of the CallSiteDynamic structure representing its own invocation, but also the address of the CallSiteDynamic structure representing its parent's invocation. Obviously, the child invocation does not know either call site; it must be passed that information by the parent. Our scheme passes these two addresses in two global variables. The first, `next_call_site_dynamic`, is new: it points to the CallSiteDynamic structure of the child invocation, which the caller would call BottomCSD and the callee calls MiddleCSD. The second, `current_call_site_dynamic`, points to the CallSiteDynamic structure of the parent invocation, which the caller calls MiddleCSD and the callee calls TopCSD. We delegate the task of assigning the proper value to `next_call_site_dynamic` to a service function in the Mercury runtime system, `prepare_for_normal_call`, and transform the bodies of all procedures, inserting a call to `prepare_for_normal_call` before every call in the original code. (This is the difference between *Body* and *Body'* above.) The values of these global variables are then picked up by `call_port_code`, which returns their original values in Mercury variables. The values of these local variables will then be stored in the stack frame of the child procedure invocation until they are needed by one of `exit_port_code`, `redo_port_code` and `fail_port_code`. Figure 1 shows the code of `prepare_for_normal_call`, while figure 2 shows the code of `call_port_code`.

The code of `prepare_for_normal_call` starts by checking whether a call has already been made through this call site in the current ancestor context. If yes, `prepare_for_normal_call` reuses the CallSiteDynamic structure allocated by the previous call for the new procedure invocation. With the scheme we have presented so far, this can only happen in the presence of backtracking, when a call is reexecuted after a goal to its left has succeeded for the second or later time. However, the first time a call is made through this call site in the current context, the call site will not have a CallSiteDynamic structure yet in the current procedure invocation's ProcDynamic structure. `prepare_for_normal_call` will therefore create one, and will link it into its position in the call

```
function prepare_for_normal_call(call_site_num) is
    local pd
    pd := current_call_site_dynamic->csd_callee_ptr
    if pd->pd_call_site_ptrs[call_site_num] is null then
        next_call_site_dynamic := allocate_call_site_dynamic()
        next_call_site_dynamic->csd_self_profile := zero_profile()
        next_call_site_dynamic->csd_callee_ptr := null
        pd->pd_call_site_ptrs[call_site_num] := next_call_site_dynamic
    else
        next_call_site_dynamic := pd->pd_call_site_ptrs[call_site_num]
    end
end
```

Fig. 1. `prepare_for_normal_call`

```
function call_port_code(proc_static_ptr, out TopCSD, out MiddleCSD) is
    local pd
    TopCSD := current_call_site_dynamic
    MiddleCSD := next_call_site_dynamic
    current_call_site_dynamic := next_call_site_dynamic
    current_call_site_dynamic->csd_self_profile.call_count += 1
    if current_call_site_dynamic->csd_callee_ptr is null then
        pd := allocate_proc_dynamic()
        pd->pd_proc_static := proc_static_ptr
        pd->pd_call_site_ptrs := allocate_csd_ptr_array(proc_static_ptr->ps_num_call_sites)
        current_call_site_dynamic->csd_callee_ptr := pd
    end
end
```

Fig. 2. `call_port_code`

graph as a child of the invoking procedure's ProcDynamic structure after initializing its fields. `prepare_for_normal_call` initializes the `csd_callee_ptr` field to null to tell the callee that this is the first call to the callee in the current context. The `call_port_code` of the callee (shown in figure 2) will check for whether this is the case and if it is, it will allocate a ProcDynamic structure, fill in its fields, and link it into the call graph in turn. The `pd_proc_static` field will be filled in with a pointer to the procedure's statically allocated ProcStatic structure, and the `pd_call_site_ptrs` field with a pointer to a newly allocated array of pointers, all of whose elements are initialized to null. In the presence of backtracking, `prepare_for_normal_call` will have found that the callee already has a CallSiteDynamic, created by a previous call; that previous call must also have filled in `current_call_site_dynamic->csd_callee_ptr` with a pointer to the ProcDynamic structure of the callee. In such cases, the `call_port_code` of the callee will use that instead of allocating a new ProcDynamic structure.

The general transformation above wrapping procedure bodies is often unnecessarily wasteful. The reason is that in most programs, the overwhelming majority of procedures (e.g. about 99% of the procedures in the Mercury compiler) are guaranteed to succeed *at most* once, and a large fraction of those (e.g. about 78% of the procedures in the Mercury compiler) are guaranteed to succeed *exactly* once. How the compiler transforms each procedure therefore depends on its determinism. If the procedure is nondet or multi (i.e. it can succeed more than once), it uses the general transformation above. If the procedure is semidet (i.e. it can succeed at most once), it doesn't insert code to get control on redo, since a redo will always step *past* and cannot step *into* such procedures:

```
p(Args) :-
    call_port_code(p_proc_static_ptr, TopCSD, MiddleCSD),
    (
        Body',
        exit_port_code(TopCSD, MiddleCSD)
    ;
        fail_port_code(TopCSD, MiddleCSD), fail
    ).
```

This transformation avoids the creation of a nondet stack frame (Mercury's equivalent to Prolog choice points) and thus reduces the overhead of profiling. Even more important, unlike the general transformation, it does not violate the invariant of the Mercury execution algorithm [18] that procedures which cannot succeed more than once do not leave behind any nondet stack frames when they succeed. (Mercury, unlike Prolog, automatically prunes the remaining disjuncts in a semidet disjunction if a disjunct succeeds, so that if *Body'* succeeds, `fail_port_code` will never be executed.)

For det procedures (i.e. those that succeed exactly once), even the code to handle failures is unnecessary, since such procedures cannot fail. We therefore use an even simpler transformation:

```
p(Args) :-
    call_port_code(p_proc_static_ptr, TopCSD, MiddleCSD),
    Body',
    exit_port_code(TopCSD, MiddleCSD).
```

There are three reasons why we use two service functions, `prepare_for_normal_call` and `call_port_code` to handle calls instead of doing everything in `prepare_for_normal_call`. First, we want to update `current_call_site_dynamic` only when execution has entered the callee. If the code to update `current_call_site_dynamic` were in `prepare_for_normal_call`, then any profiling signals arriving between the return from `prepare_for_normal_call` and the call to the callee, i.e. while the program executes the instructions in the caller that move the input arguments of the call to the registers where the callee expects them, would be incorrectly charged against the callee. Second, by avoiding the requirement to know the address of the callee's ProcStatic structure when compiling the caller, the split makes it much easier to handle higher order calls, method calls and entries to foreign code, since in those cases the identity of the callee is not known until runtime. Third, we need to reset `current_call_site_dynamic` when the call returns, and as long as we wrap `call_port_code`, `exit_port_code`, etc around procedure bodies rather than around call sites, we need something like `call_port_code` to pick up TopCSD and MiddleCSD.

Wrapping calls would have the advantage that procedures that don't make nondeterministic calls need only remember the address of their own CallSiteDynamic structure; they need not remember the address of their parent's CallSiteDynamic structure or of their children's. Procedures that do make nondeterministic calls need to remember the addresses of the CallSiteDynamic structures of their nondeterministic children, so that they can reset `current_call_site_dynamic` when execution backtracks into such a child, but this is a small cost given the rarity of such procedures. Wrapping calls can therefore be expected to reduce memory consumption on the stacks. The reason why we wrap procedure bodies instead is that calls outnumber procedure bodies by a significant margin; in the Mercury compiler, the average number of call sites per procedure is about 5.5. Wrapping procedure bodies therefore adds much less code to the program, and we expect the performance gain from the relative improvement in code locality to outweigh the improvement from lower stack usage.

Linking CallSiteDynamic and ProcDynamic structures upward as well as downward would allow us to avoid using any local variables at all to remember CallSiteDynamic addresses. From `current_call_site_dynamic`, a procedure invocation could always find the CallSiteDynamic structure of its parent as well as the CallSiteDynamic structures of all its children, and therefore `call_port_code`, `exit_port_code`, `redo_port_code` and `fail_port_code` would need only the current value of `current_call_site_dynamic` to find out what its next value should be. The cost would be an extra field in both the CallSiteDynamic and ProcDynamic structures. These structures are effectively permanent; once they are allocated, their memory is not reclaimed until the end of execution. Stack space is of course reclaimed much earlier, on the success or failure of a call (depending on the determinism of the procedure in question). This is a sufficient reason for us eschew upward links and accept the cost in stack usage. Another, more compelling reason is that navigating the tree using upward links is incompatible with the technique we use to handle recursion.

4.3 Handling self-recursion

As described thus far, the structure we create is a tree, and it has a call site node for almost every call made by the program.[3] This is obviously not practical: the amount of memory needed to store such a tree would be infeasible for all programs but those with trivially short runtimes. Even if we could store such trees, the information in it would be too sparse to be of any use without significant postprocessing. The most important postprocessing one would want to do is to merge together the nodes that result from self-recursive invocations of a procedure. In a language that supports recursion but not iteration, performing this merging at runtime causes a dramatic reduction in the size of the tree. There are two reasons for this. The obvious reason is that the recursion is often quite deep, and using a single node to represent all the recursive calls eliminates the nodes that would have been there for all the recursive calls after the first. The less obvious but far more important reason is that if more than one recursive invocation makes a call through a call site within the recursive procedure, then the subtrees of those call sites will be merged as well. The sizes of those subtrees, and therefore the savings, can be huge, particularly for calls near the top of the call tree.

Merging the nodes of self-recursive calls at runtime requires treating self-recursive calls differently from other calls. The compiler will continue to insert calls to `prepare_for_normal_call` before all other calls. However, before self-recursive calls it will insert a call to `prepare_for_recursive_call` instead. This function behaves almost the same as `prepare_for_normal_call` except that if and when it initializes the `csd_callee_ptr` field of the CallSiteDynamic it has allocated, it will make it to point to the ProcDynamic structure of the current invocation instead of filling it with null. This will cause the `call_port_code` of the new recursive invocation to recognize that this call does not create a new context, but is part of an existing context instead.

4.4 Handling mutual recursion within a module

As we discussed in section 3, self-recursion is not the only kind of recursion a Mercury profiler must handle. For example, in the Mercury compiler many performance-critical procedures, and some of the procedures near the top of the call tree, are members of cliques of mutually recursive procedures. Unless we merge together the nodes corresponding to invocations of procedures within a clique, the call graph constructed during a profiling run will continue to be infeasibly large for many interesting programs.

Some of these cliques consist of procedures that are all in the same module; some contain procedures from more than one module. Originally we thought that merging nodes involved in intramodule mutual recursion without also merging nodes involved in intermodule mutual recursion would make feasible the profiling of a large enough class of programs that we need not complicate the implementation any further. We therefore implemented a scheme that did this.

The basic idea of this scheme is to have the compiler identify groups of mutually recursive procedures within the module it is compiling, and to treat each of those groups as a single profiling unit. Each of these groups corresponds to a strongly connected component of the part of the call

[3] The exceptions are calls that are reexecuted after a goal to their left has succeeded for the second or later time. Given the rarity in most Mercury programs of procedures that can succeed more than once, this will be a small minority of calls.

graph visible from the module, and we therefore call them SCCs. (Every procedure that is not mutually recursive with another procedure in the same module has its own SCC.)

The scheme replaces ProcStatic structures with SccStatic structures and ProcDynamic structures with SccDynamic structures. Instead of generating a ProcStatic structure for each procedure in the module, the compiler generates an SccStatic structure for each SCC in the module. SccStatic structures have the same fields as ProcStatic structures do, with one difference: the `ps_call_sites` array has an entry for all the call sites in all the procedures in the SCC. Similarly, at runtime procedure prologues fill in `current_call_site_dynamic->csd_callee_ptr` with a pointer to an SccDynamic structure, not a ProcDynamic structure. The `pd_proc_static` field of the SccDynamic structure points to the SccStatic structure of the SCC, while the `pd_call_site_ptrs` field now points to an array which has an element for every call site in the SCC, just like the `ps_call_sites` array it parallels. This time, the compiler will use `prepare_for_recursive_call` to preface not only self-recursive calls but also any call to a procedure within the current SCC. This is sufficient to eliminate the distinction between self-recursion and mutual recursion within a module.

Having produced an implementation that did this, we found that this approach did indeed reduce the size of the call graph quite substantially, reducing the memory requirement for a profiling run of the Mercury compiler to the point where it just fit into the memory of the machine we had available at the time (128 Mb). The Mercury compiler is a pretty big program: it is over a quarter of a million lines of Mercury code, and its runtime is usually five seconds to a minute. However its normal memory consumption is usually around 20 Mb; that's when the garbage collector kicks in. We felt that having profiling increase memory consumption about five fold was excessive, and we wanted to be able to profile programs that are bigger and longer-running than the Mercury compiler. We therefore decided that we needed a method that could reuse existing nodes of the call graph even in the presence of recursion that crosses module boundaries.

4.5 Handling mutual recursion across modules

From the point of view of the profiler, the ideal way to handle recursion across modules would be to give the compiler access to the complete call graph of the program even when it is compiling a single module, and then use the scheme described in section 4.4.

Unfortunately, giving the compiler access to the complete call graph of the program is far from trivial. The reason is that the links in the call graph are subject to change as programmers edit modules. Suppose the system has the complete call graph of the program stored somewhere, and the programmer then edits two modules, modules A and B, in ways that leave their interfaces the same. The standard Mercury compilation system simply compiles first one module (say module A), then the other (module B). However, when the compiler compiles module A, it does not have access to an accurate call graph: the links in the call graph that represent calls made in module B are obsolete. The only way to fix this problem is to change the order in which the compiler performs its tasks. First, it should update the two modules' portions of the call graph. Since the identity of the called procedure in a call may depend on type and mode information as e.g. the compiler resolves overloading, this requires performing static analysis on both modules. Only then can the compiler generate code. If we (simplistically) divide the compiler into a front end which does static analysis and a back end that does everything else, then the old order was front end on module A, back end on module A, front end on module B and back end on module B, while the required new order is front end on module A, front end on module B, back end on module A and back end on module B. Dividing the compiler in this fashion without making compilation much more time consuming would require a considerable amount of reengineering.

Actually, the situation is even more complicated. Suppose there is one group of mutually recursive procedures in modules A and B, and another group in modules C and D, where some procedures of the second group call procedures in the first group, but not vice versa. A modification to e.g. module A may introduce a call that merges the two previously separate SCCs into a single SCC. This will change the status of some calls in modules C and D: what were previously non-recursive calls now become recursive calls. Since the compiler must insert different code before these two kinds of calls, the change to module A will require modules C and D to be recompiled, even if the interface of module A has not changed.

Given that detecting cross-module cycles at compile time is infeasible, we have no choice but to detect them at runtime. Since our approach detects cycles without regard for whether they are cross-module or not, or even for whether they involve self-recursion or mutual recursion, we do not need the mechanisms we presented in sections 4.3 and 4.4. The starting point of our implementation is therefore the scheme we presented in section 4.2, which handles all calls uniformly. The schemes we presented in section 4.3 always reused the context (CallSiteDynamic and ProcDynamic structures) of the outermost invocation in a sequence of self-recursive calls, we now want a scheme that always reuses the context of the outermost invocation in *any* sequence of recursive calls, even if the stack frames of those recursive calls are interspersed with the stack frames of other procedures, which may be from other modules.

Our approach is based on the definition of a recursive cycle: a call is part of a recursive cycle if the called procedure is one of its own ancestors. At the time of a call, we could check whether the called procedure is one its own ancestors by walking the stacks (Mercury has two) upwards, at each step going from callee to caller, looking for the address of the outermost already active stack frame of the called procedure. Since the stacks can be very deep, searches can be quite expensive. This is especially bad for nonrecursive calls, which all involve an unsuccessful search which must scan the stacks all the way back to `main/2`. (Successful searches can be short-circuited by adding extra links between stack frames.) However, even the cost of searching a few stack frames would be greater than our objective of keeping overheads small relative to the program itself would allow us to tolerate. We therefore detect cycles *without* searching the stacks. Instead, we add new fields to each ProcStatic structure that tell us whether execution is currently inside a call to the corresponding procedure or not. If so, these fields must also tell us where the ProcDynamic structure of the outermost invocation is.

The reason why we need to know the address of the ProcDynamic structure of the outermost invocation has to do with why we want to detect cycles in the first place. Suppose the `call_port_code` of a procedure finds, in a given invocation, that `current_call_site_dynamic->csd_callee_ptr` is null, i.e. this is the first call to this procedure in this precise context. Previously, `call_port_code` always created a new ProcDynamic structure for the call, even if an invocation of this procedure was already active among the ancestors. From now on we will avoid allocating a new ProcDynamic structure and instead reuse the ProcDynamic structure of the outermost invocation of this procedure if there is one and if `current_call_site_dynamic->csd_callee_ptr` is initially null. This merges the profiling information of this call with the outermost call of this procedure. This is an appropriate thing to do because consumers of profiling information do not care *which* iteration of a recursive loop incurs a given cost; they require only that the cost be properly charged to the loop. Making *all* calls to a given procedure in a recursive loop use the same ProcDynamic structure even if the loop extends over more than one module reduces the amount of memory needed for the call graph quite substantially, for the reasons we discussed in section 4.5.

We obviously need a field in ProcStatic structures (which we call `ps_outermost_proc_dynamic`) that points to the ProcDynamic structure of the outermost existing invocation of the corresponding procedure, if there is one. We also need a mechanism that tells us whether there is such an invocation. There are two obvious mechanisms we could use for detecting this.

The first mechanism (which we call 'save/restore') requires the `ps_outermost_proc_dynamic` field to be valid at all times, containing a null pointer if and only if there is no active invocation of this procedure in the chain of ancestors, and thus no outermost invocation. The `call_port_code` of each procedure stores the original value of the `ps_outermost_proc_dynamic` field in a stack slot before checking whether this value is null. If it is, then this is the outermost call to this procedure: `call_port_code` therefore allocates a ProcDynamic structure, and overwrites the `ps_outermost_proc_dynamic` field with its address after filling it in. If the initial value of the `ps_outermost_proc_dynamic` field is not null, then `call_port_code` does not allocate a ProcDynamic structure, and uses the one pointed to by the `ps_outermost_proc_dynamic` field instead. If the procedure can succeed more than once, `call_port_code` will then store the final value of the `ps_outermost_proc_dynamic` field in another stack slot. The `exit_port_code`

```
function call_port_code(proc_static_ptr, out TopCSD, out MiddleCSD, out OldOuterMost) is
    local pd
    TopCSD := current_call_site_dynamic
    MiddleCSD := next_call_site_dynamic
    current_call_site_dynamic := next_call_site_dynamic
    current_call_site_dynamic->csd_self_profile.call_count += 1
    OldOuterMost := proc_static_ptr->ps_outermost_proc_dynamic
    if current_call_site_dynamic->csd_callee_ptr is null then
        if proc_static_ptr->ps_outermost_proc_dynamic is null then
            pd := allocate_proc_dynamic()
            pd->pd_proc_static := proc_static_ptr
            pd->pd_call_site_ptrs :=
                allocate_csd_ptr_array(proc_static_ptr->ps_num_call_sites)
            current_call_site_dynamic->csd_callee_ptr := pd
            proc_static_ptr->ps_outermost_proc_dynamic :=
                current_call_site_dynamic->csd_callee_ptr
        else
            current_call_site_dynamic->csd_callee_ptr :=
                proc_static_ptr->ps_outermost_proc_dynamic
        end
    else
        if proc_static_ptr->ps_outermost_proc_dynamic is null then
            proc_static_ptr->ps_outermost_proc_dynamic :=
                current_call_site_dynamic->csd_callee_ptr
    end
end
```

Fig. 3. `call_port_code` for the save/restore approach

and `fail_port_code` can then restore the field to the value it had before entry to this call, while `redo_port_code` (if it exists) can restore it to the value it had after entry to this call. We can dispense with the second stack slot if the procedure in question can succeed at most once. This scheme therefore requires between one and two stack slots per stack frame; for programs that are very close to deterministic, such as the Mercury compiler, the average number will be very close to one. Figure 3 shows `call_port_code` for procedures that succeed at most once; the version for procedures that can succeed more than once differs only in also returning the final value of `proc_static->ps_outermost_proc_dynamic` in an extra output argument, for use at the redo port (which procedures that succeed at most once do not have).

The second mechanism (which we call 'activation counting') involves adding a second field to ProcStatic structures, an activation counter whose value gives the number of times the corresponding procedure appears in the chain of ancestors starting at the current call. Since `call_port_code` can use the value of this counter to check whether the call is an outermost call, the `ps_outermost_proc_dynamic` field need not contain any particular value if this counter contains zero. The activation counter is of course initialized to zero, and is maintained simply by incrementing it in `call_port_code` and in `redo_port_code`, and decrementing it in `exit_port_code` and in `fail_port_code`. The point of this counter is that `exit_port_code` and `fail_port_code` need not restore the contents of the `ps_outermost_proc_dynamic` field. If the call is not an outermost call, then the `call_port_code` will not have changed the value of the `ps_outermost_proc_dynamic` field and therefore there is no change to be undone, while if the call is an outermost call, then the previous contents of the `ps_outermost_proc_dynamic` field was garbage, and the value it has now is as good 'garbage' as any to reset it to. We therefore need not save the old contents of the `ps_outermost_proc_dynamic` field in a stack slot in the first place. For nondeterministic procedures, we do still need a stack slot holding the new value of the `ps_outermost_proc_dynamic` field. To see why, consider a nondeterministic procedure `p` that contains two conjoined calls to a

```
function call_port_code(proc_static_ptr, out TopCSD, out MiddleCSD) is
    local pd
    TopCSD := current_call_site_dynamic
    MiddleCSD := next_call_site_dynamic
    current_call_site_dynamic := next_call_site_dynamic
    current_call_site_dynamic->csd_self_profile.call_count += 1
    if current_call_site_dynamic->csd_callee_ptr is null then
        if proc_static_ptr->ps_activation_counter = 0 then
            pd := allocate_proc_dynamic()
            pd->pd_proc_static := proc_static_ptr
            pd->pd_call_site_ptrs :=
                allocate_csd_ptr_array(proc_static_ptr->ps_num_call_sites)
            current_call_site_dynamic->csd_callee_ptr := pd
            proc_static_ptr->ps_outermost_proc_dynamic :=
                current_call_site_dynamic->csd_callee_ptr
        else
            current_call_site_dynamic->csd_callee_ptr :=
                proc_static_ptr->ps_outermost_proc_dynamic
        end
    else
        if proc_static_ptr->ps_activation_counter = 0 then
            proc_static_ptr->ps_outermost_proc_dynamic :=
                current_call_site_dynamic->csd_callee_ptr
    end
    proc_static_ptr->ps_activation_counter += 1
end
```

Fig. 4. `call_port_code` for the activation counting approach

nondeterministic procedure `q`. If the ancestors of a given call to `p` do not include `q`, then both calls to `q` will be outermost calls. When the second call fails and execution backtracks to the first call, the `redo_port_code` in the first call would not only have to increment the activation counter from 0 to 1, it would also have to change the `ps_outermost_proc_dynamic` field of `q`'s ProcStatic structure, to make it point to the ProcDynamic structure of not the second call but the first. This can only be done if every invocation of `q` saves a pointer to the ProcDynamic structure it uses (i.e. the new value of the `ps_outermost_proc_dynamic` field in `q`'s ProcStatic structure) in a stack slot, so that it copy the value in this stack slot back to the `ps_outermost_proc_dynamic` field in the `redo_port_code`. Figure 4 shows `call_port_code` for this approach, again for a procedure that succeeds at most once.

Our implementation can use either approach, depending on a configuration option; we will discuss the performance implications of the choice in section 6. Until then, we will assume the use of the activation counters.

Regardless of which mechanism the profiler uses to detect recursive calls, it will find such calls only if the program makes them. If the code of procedure `p` contains calls to procedure `q` and vice versa, but the calls to `p` in procedure `q` are in execution paths that happen not to be taken during a particular run of the program, then the nodes of calls to procedures `p` and `q` will not belong to a cycle in the call graph constructed during that run. What the mechanisms we just described detect is therefore dynamic, actual recursion, not the static, potential recursion that the approach in section 4.4 was based on.

4.6 Preserving tail recursion

The high-level transformation we use to instrument programs we have just described works well for many programs, but adding `exit_port_code` after procedure bodies defeats last call optimization

```
p(...) :-
    (
        ...
    ;
        % call site #j
        p(...), % not tailcall
        ...
    ;
        ...,
        % call site #k
        p(...) % tailcall
    ).
```

Fig. 5. Doubling transformation: before

```
p(...) :-                            inner_p(...) :-
    (                                    (
        ...                                  ...
    ;                                    ;
        % call site #j                       % call site #j
        p(...), % not tailcall               p(...), % not tailcall
        ...                                  ...
    ;                                    ;
        ...,                                 ...,
        % call site #k                       % call site #k
        inner_p(...) % was tailcall          inner_p(...) % tailcall
    ).                                   ).
```

Fig. 6. Doubling transformation: after

and tail recursion. For procedures that would otherwise be tail-recursive this results in $O(n)$ stack-space usage (where n is the depth of the recursion) rather than $O(1)$. This is clearly a problem for many programs; for example, for programs that use would-be tail recursive procedures to read in large input files, the sizes of the stacks limit the sizes of the input files that can be used during profiling runs.

We therefore propose that for procedures that are initially tail recursive and can succeed at most once, the profiler should use a modified transformation that preserves tail recursion. (Preserving tail recursion in procedures that can succeed more than once is not worthwhile, because the stack frames of such procedures cannot be reclaimed on success anyway; they can only be reclaimed on failure.) We restrict our attention to self-tail-recursion. We have looked into extending the technique to mutual recursion that is visible within a module, but did not find any way to do so without incurring high levels of overheads. This is not a very significant limitation, given that self-tail-recursion is much more common than mutual tail recursion. Also, turning mutual tail recursion into self-tail-recursion is usually an optimization, so performance-conscious programmers will probably not mind doing it if they need to do so in order to be able to profile their programs with realistic input data.

To preserve tail recursion, we must move `exit_port_code` out of its position at the end of a procedure, so that the tail call can be the last thing executed in the procedure body. Instead, we must perform the work of `exit_port_code` somewhere else, and that somewhere else can only be in the caller. However, we do not wish to require the compiler to know, when generating a call to a procedure, whether that procedure is tail recursive or not, since such a requirement would conflict with separate compilation.

```
p(...) :-
    call_port_code(p_proc_static_ptr, TopCSD, MiddleCSD),
    (
        (
            ...
        ;
            ...
            prepare_for_normal_call(j),
            p(...),
            ...
        ;
            ...,
            save_recursion_depth_N(trcsn_1, ..., trcsn_N,
                SavedDepth_i, ..., SavedDepth_N),
            prepare_for_tail_call(k),
            (
                inner_p(...),
                restore_recursion_depth_exit_N(MiddleCSD, trcsn_1, ..., trcsn_N,
                    SavedDepth_i, ..., SavedDepth_N),
            ;
                restore_recursion_depth_fail_N(MiddleCSD, trcsn_1, ..., trcsn_N,
                    SavedDepth_i, ..., SavedDepth_N),
                fail
            )
        ),
        exit_port_code(TopCSD, MiddleCSD)
    ;
        fail_port_code(TopCSD, MiddleCSD), fail
    ).
```

Fig. 7. `p` after the doubling and deep profiling transformations

Our solution to this conflict starts with a "doubling" transformation, whose steps are all guaranteed to preserve the semantics of the program. Given a tail-recursive procedure `p`, this transformation

- invents a new predicate name, say `inner_p`, that does not occur in the rest of the program,
- defines `inner_p` as a new predicate that is identical to `p` in everything but its name,

```
inner_p(...) :-
    (
        ...
    ;
        ...,
        prepare_for_normal_call(j),
        p(...),
        ...
    ;
        ...,
        prepare_for_tail_call(k),
        inner_p(...)
    ).
```

Fig. 8. `inner_p` after the doubling and deep profiling transformations

– changes all the self-tail-recursive calls in the body of `p` to call `inner_p` instead, leaving all other calls (whether to `p` or to other predicates) unchanged,
– updates the body of `inner_p` to be identical to the new body of `p`.

We say that `inner_p` is the inner procedure of `p`, while `p` is the outer procedure of `inner_p`.

The effect of the doubling transformation is shown in figures 5 and 6. Some of the recursive calls in the original procedure `p`, such as the one at call site k, are tail recursive, and are thus redirected to call `inner_p` instead. Some, such as the one at call site j, are not, and are thus preserved unchanged.[4]

Procedures that are affected by the doubling transformation are handled specially by the deep profiling transformation. The differences from the standard transformation described in the previous sections are as follows.

– In the inner procedure, the deep profiling transformation does not add calls to `call_port_code`, `exit_port_code` or `fail_port_code`.
– In both the inner and the outer procedures, the deep profiling transformation prefaces calls to the inner procedure with a call to the new service function `prepare_for_tail_call` instead of the usual call to `prepare_for_normal_call`. This compensates for the absence of `call_port_code` in the inner procedure.
– In the outer procedure, the deep profiling transformation wraps all calls to the inner procedure in calls to `save_recursion_depth_N`, `restore_recursion_depth_exit_N` and (if the call can fail) `restore_recursion_depth_fail_N` as shown in figure 7, with N being the number of self-tail-recursive call sites in the original procedure (and hence the number of calls to `inner_p` in `p` and `inner_p`), and `trcsn_1`, ... `trcsn_N` being the call site numbers of those self-tail-recursive call sites. (Therefore in our example, k will be a member of the set {`trcsn_1`, ... `trcsn_N`}, but j will not be.) This additional code compensates for the absence of `exit_port_code` and `fail_port_code` in the inner procedure.

These are the only differences from the standard transformation. For example, call sites in the bodies of both the inner and outer procedure that are not calls to the inner procedure still have a call to the usual service procedure (`prepare_for_normal_call`, `prepare_for_higher_order_call` etc) inserted before them.

In cases where the original program would have had a call from (say) `s` to `p` followed by m self-tail-recursive calls from `p` to `p`, the transformed program will have a call from `s` to `p`, a call from `p` to `inner_p`, and $m - 1$ self-tail-recursive calls from `inner_p` to `inner_p`. This keeps the complexity of stack memory consumption at O(1), although of course the constant is now higher. since the stack has one more frame, and the deep profiling transformation makes each frame bigger too.

Figures 9, 10 and 11 give the code of the three new service routines; the code of `restore_recursion_depth_fail_N` is identical to the code of `restore_recursion_depth_exit_N` except for updating the `fail_count` field instead of `exit_count`. The `prepare_for_tail_call` function performs the work of the `call_port_code`that is missing from `inner_p` as well as the work of `prepare_for_normal_call`. The functions `restore_recursion_depth_exit_N` and `restore_recursion_depth_fail_N` have the task of performing the work of the calls to `exit_port_code` and `fail_port_code`respectively that are missing from `inner_p`. `save_recursion_depth_N` exists to make the arrangement reentrant. All these service routines use a new field of the CallSiteDynamic structure, the `tail_recursion_depth` field, which is initialized to zero when the structure is allocated. Our algorithm only needs this field to be present and initialized in CallSiteDynamic structures corresponding to self-tail-recursive call sites, but of course it is simpler to make it part of all CallSiteDynamic structures.

Let's walk through what happens during a chain of tail recursive calls. Assume for now that

[4] Since we are assuming that `p` can succeed at most once, the body of `p` is cannot be an arbitrary disjunction but must be either a switch (a disjunction in which each disjunct unifies the same input argument with a distinct function symbol) or a set of nested switches.

```
function prepare_for_tail_call(call_site_num) is
    local pd, callee_csd
    pd := current_call_site_dynamic->csd_callee_ptr
    if pd->pd_call_site_ptrs[call_site_num] is null then
        callee_csd := allocate_call_site_dynamic()
        callee_csd->csd_self_profile := zero_profile()
        callee_csd->csd_callee_ptr := pd
        callee_csd->tail_recursion_depth := 0
        pd->pd_call_site_ptrs[call_site_num] := callee_csd
    else
        callee_csd := pd->pd_call_site_ptrs[call_site_num]
    end
    callee_csd->tail_recursion_depth++
    current_call_site_dynamic := callee_csd
end
```

Fig. 9. Service functions for preserving tail recursion: `prepare_for_tail_call`

```
function save_recursion_depth_N(MiddleCSD, trcsn_1, ..., trcsn_N,
        out SavedDepth_1, ..., out SavedDepth_N) is
    local pd, callee_csd
    pd := MiddleCSD->csd_callee_ptr
    for i := 1 to N do
        callee_csd := pd->pd_call_site_ptrs[trcsn_i]
        if callee_csd is null then
            SavedDepth_i := 0
        else
            SavedDepth_i := callee_csd->tail_recursion_depth
            callee_csd->tail_recursion_depth := 0
        endif
    done
end
```

Fig. 10. Service functions for preserving tail recursion: `save_recursion_depth_N`

- the chain of tail recursive calls is not interrupted by any non-tail calls,
- `p` has only one self-tail-recursive call site, and
- this is the first call to `p` in the current ancestor context.

The chain will start with a call to `p` from some other procedure, e.g. `s`. The `call_port_code` of `p` will update `current_call_site_dynamic` to point to the call site from which `s` called `p`. This will still be the value of `current_call_site_dynamic` when execution reaches the call to `save_recursion_depth_N`. As per our assumption, `N` will be 1, and `trcsn_1` will be k, the call site number of the only self-tail-recursive call site in the original `p`. Since `pd->pd_call_site_ptrs[k]` is initially null, `save_recursion_depth_N` will return zero in `SavedDepth_1`. `prepare_for_tail_call` will then allocate a new CallSiteDynamic structure, and make `pd->pd_call_site_ptrs[k]` point to it, just as `prepare_for_normal_call` would. `prepare_for_normal_call` would initialize the `csd_callee_ptr` field of the new CallSiteDynamic to null, and let `call_port_code` fill it in with a valid pointer; since `inner_p` has no `call_port_code`, `prepare_for_tail_call` initializes the field with the value `call_port_code` would have assigned, the address of the ProcDynamic structure of `p` itself. (Reusing `p`'s ProcDynamic structure for `inner_p` is OK because inner/outer procedure pairs have identical structures.) Future tail recursive calls to `inner_p` will also all reuse this same ProcDynamic structure.

Every call to `inner_p` will increment the `tail_recursion_depth` field of the CallSiteDynamic structure at this ProcDynamic's k'th call site. Eventually, when the chain of tail recursive calls

```
function restore_recursion_depth_exit_N(MiddleCSD, trcsn_1, ..., trcsn_N,
        out SavedDepth_1, ..., out SavedDepth_N) is
    local pd, callee_csd
    pd := MiddleCSD->csd_callee_ptr
    for i := 1 to N do
        callee_csd := pd->pd_call_site_ptrs[trcsn_i]
        if callee_csd is not null then
            callee_csd->csd_self_profile.call_count += callee_csd->tail_recursion_depth
            callee_csd->csd_self_profile.exit_count += callee_csd->tail_recursion_depth
            callee_csd->tail_recursion_depth := SavedDepth_i
        endif
    done
end
```

Fig. 11. Service functions for preserving tail recursion: `restore_recursion_depth_exit_N`

comes to an end, that `tail_recursion_depth` field will contain a count of all the calls to `inner_p`, the first call being from `p` and the rest from `inner_p` itself. Given a sequence of tail calls, either they all succeed or they all fail; having the caller succeed while the callee fails or vice versa is possible only for non-tail calls. This is why `restore_recursion_depth_exit_N`, which is called if the call from `p` to `inner_p` succeeds, is justified in incrementing the exit count of the call site by the number of calls through that call site, and why `restore_recursion_depth_fail_N` is similarly justified in doing the same to the fail count field. The `exit_port_code` or `fail_port_code` of `p` itself will then record the success or failure of the call to `p`.

However, when `restore_recursion_depth_exit_N`, `restore_recursion_depth_fail_N`, `exit_port_code` and `fail_port_code` are called, `current_call_site_dynamic` still points to the call site of the deepest call to `inner_p`, since `inner_p` never resets `current_call_site_dynamic`. This is why `p` passes MiddleCSD explicitly to both `restore_recursion_depth_exit_N` and `restore_recursion_depth_fail_N`, and why `restore_recursion_depth_exit_N`, `restore_recursion_depth_fail_N`, `exit_port_code` and `fail_port_code` all use MiddleCSD rather than `current_call_site_dynamic` to locate the CallSiteDynamic structure of the current call. The `exit_port_code` or `fail_port_code` of `p` will also reset `current_call_site_dynamic` to TopCSD, so the only time when `current_call_site_dynamic` does not point to the call site of the currently executing procedure invocation is during the interval from the exit or failure of the innermost call to `inner_p` and the `exit_port_code` or `fail_port_code` of `p`. The difference affects only which CallSiteDynamic structure is charged with a clock tick if and when a profiling interrupt arrives during the interval and does not affect which user-visible *procedure* is charged with the clock tick. Since which invocation of `p` or `inner_p` the profiling interrupt interrupts during a sequence of tail calls to `inner_p` is effectively random anyway, the difference does not distort the profile in any significant way.

What happens if we relax the third assumption, and suppose that this is not the first call to `p` in the current ancestor context? Mostly the same thing. The main difference is that the previous invocations of `p` will have filled in some of the slots in the array of `p`'s ProcDynamic structure, and this may have included the call site of the call to `inner_p`. If it has, then `prepare_for_tail_call` does not need to allocate a new CallSiteDynamic structure, and can reuse the existing one instead. Its fields will contain the right values: the `csd_callee_ptr` field still points to `p`'s ProcDynamic structure, and while the `tail_recursion_depth` field has been updated since its initialization, it has been reset to contain zero by `restore_recursion_depth_exit_N` or `restore_recursion_depth_fail_N`. The `csd_self_profile` field may of course have different contents, but the service functions just increment its various subfields; they do not otherwise depend on its initial content.

What happens if we relax the second assumption, and suppose that `p` (and thus `inner_p`) contains more than one call site calling `inner_p`? Then it is possible that the calls to `inner_p` in the tail-recursive chain involved more than one call site and thus more than one slot in the ProcDynamic structure's array. The various calls to `prepare_for_tail_call` will ensure that every call site that `inner_p` was called through will have non-null CallSiteDynamic pointers in their slots, and that the `tail_recursion_depth` field of each such structure records the number of times the corresponding call site was called through. `restore_recursion_depth_exit_N` or `restore_recursion_depth_fail_N` will then increment the `exit_count` or `fail_count` fields of these structures by the values of their respective `tail_recursion_depth` fields. They will then reset the `tail_recursion_depth` fields of all the filled-in slots to zero to prepare for other calls to `p`.

What happens if we relax the first assumption, and suppose that `inner_p` makes some non-tail calls? If none of these non-tail calls call `p` or `inner_p`, then the operation of `p` and `inner_p` is not affected. Since all the calls to `inner_p` are in `p` and `inner_p` by construction and all calls from `inner_p` to `inner_p` are tail calls, the only potential problem is a call in `inner_p` either directly to `p` or to another procedure, say `q`, which then calls `p`. Since the original call to `p` from `s` is still active, that call to `p` from `inner_p` or `q` will reuse the ProcDynamic structure of the call to `p` from `s`. However, if `p` calls `inner_p` through *any* of its call sites, the calls to `save_recursion_depth_N` and either `restore_recursion_depth_exit_N` or `restore_recursion_depth_fail_N` will together ensure that when this recursive invocation of `p` returns to its caller, the `tail_recursion_depth` fields of *all* the call sites that call `inner_p` in the ProcDynamic structure shared by `p` and `inner_p` are restored to the values they had at the time of the recursive call to `p`. The only change that the call to `p` can do to that array is to replace a null pointer with a pointer to a CallSiteDynamic structure whose `tail_recursion_depth`field (at the time of return) contains zero. Since a null pointer is just an optimized representation of such a CallSiteDynamic structure, this is not a semantic change at all. Therefore calls to `p` from within `inner_p` are invisible to the service functions.

4.7 Handling higher order calls

So far we have assumed that all calls through a given call site are to the same procedure. While this is true for call sites that correspond to first order calls, which are the vast majority of call sites in most programs, it is not true for call sites that correspond to higher order calls. The Mercury profiler supports three kinds of higher order calls: calling through procedure-valued variable, calling a typeclass method, and calling the generic unification and comparison procedures.[5]

The `pd_call_site_ptrs` field of each ProcDynamic structure points to an array that has one element for each call site in the procedure. We said in section 4.1 that the elements are pointers to CallSiteDynamic structures. For first order call sites, this is true. For higher order call sites, it is not. The element for a higher order call site conceptually contains a table mapping the identity of the called procedure to a CallSiteDynamic structure. In practice, we currently implement this table as a linked list of key-value pairs, with the values being pointers to CallSiteDynamic structures.

As its name suggests, we insert a call to `prepare_for_normal_call` only before first order calls. Before calls through procedure-valued variables, we insert a call to the procedure `prepare_for_higher_order_call` (see figure 12) instead. It is `prepare_for_higher_order_call` that searches the table for a CallSiteDynamic structure that is appropriate for the current call, and creates one if it does not exist yet.

This scheme takes advantage of the weakly typed nature of C (the language in which the runtime system of Mercury is written) to store pointers to CallSiteDynamic structures and to list nodes in different elements of the same array. Each element of the `ps_call_sites` field of a ProcStatic structure has a component that specifies which kind of pointer the corresponding element in the `csd_callee_ptr` field of a ProcDynamic structure for the same procedure contains, so that the code that traverses the call graph at the end of execution in order to save its information content in a file will know what to do.

[5] The fourth kind of higher order call supported by the Mercury compiler, calling a procedure implemented in the Aditi2 deductive database system (the successor of Aditi [19]), is not supported by the Mercury profiler, since that would require profiling the C++ code of Aditi2 itself.

```
function prepare_for_higher_order_call(closure, call_site_num) is
    local pd, key, list_ptr
    pd := current_call_site_dynamic->csd_callee_ptr
    key := extract_key_from_closure(closure)
    list_ptr := pd->pd_call_site_ptrs[call_site_num]
    while list_ptr is not null do
        if list_ptr->key = key then
            next_call_site_dynamic := list_ptr->value
            return
        end
        list_ptr := list_ptr->next
    end

    next_call_site_dynamic := allocate_call_site_dynamic()
    next_call_site_dynamic->csd_self_profile := zero_profile()
    next_call_site_dynamic->csd_callee_ptr := null
    list_ptr := allocate_list_node()
    list_ptr->key := key
    list_ptr->value := next_call_site_dynamic
    list_ptr->next := pd->pd_call_site_ptrs[call_site_num]
    pd->pd_call_site_ptrs[call_site_num] := list_ptr
end
```

Fig. 12. `prepare_for_higher_order_call`

There are several reasonable choices for what one may use as the key when calling through a procedure-valued variable, which is implemented as a closure:

- The most obvious choice is the address of the called procedure. Since the program is about to call through it, using it as the key requires no additional overhead.
- We can also use the source location of the code that constructed the closure as the key, since this also uniquely identifies the called procedure.[6] If a particular higher order call site happens to be performance critical because a procedure called from it is expensive, the user may discover that more than one place in the program creates closures that, invoked at this call site, call the same expensive procedure. He or she may then want to know whether the performance characteristics of the calls resulting from the different closures differ, and if so, what those performance characteristics are. If it turns out that calls using closures constructed in one source location are always expensive but calls using closures constructed somewhere else are not, then the first source location is worth investigating but the second is not. Using source location as the key provides the information programmers need to do this.
  Closures in the Mercury implementation already include a pointer to a compiler generated static structure (called the closure's *layout* structure) that describes the types of the values stored inside the closure, since without that information the closure could not be copied. Adding an extra field to closure layout structures that uniquely identifies the location in the source where the closure was created costs a trivial amount of memory and no time. Using the address of the layout structure (which now identifies the source location that constructed the closure) as the key costs an extra load instruction, but since Mercury stores the pointer to the layout structure right next to the address of the called procedure, the probability that this load will cause an extra cache miss is reasonably small.
- The most specific choice is the address of the closure itself. This choice allows the profiler to distinguish between different closures constructed at the same source location. However, since

[6] This requires us to disable a small and rarely used optimization which reuses the contents of an old closure when constructing a new one. The non-optimizated code in such cases wraps up the old closure as a hidden argument in a new closure that calls a new procedure created specifically for that closure construction site.

there is no obvious method that the programmer who looks at a profile can usefully optimize some but not all closures constructed at a source code site or can even distinguish them in a meaningful fashion, we have not found the extra prevision worth the cost. The cost itself is substantial, because using closure addresses as keys puts pointers to the Mercury heap into the call graph. This means that the garbage collector must scan the call graph, which grows to be quite a large data structure; it can increase the peak resident set size of the Mercury compiler from around 20 Mb to about 34 Mb (see section 6.1). If we don't use closure addresses as keys, then the call graph does not contain any pointers into the Mercury heap, which means that we can put it into memory that is not scanned by the collector. Since this speeds up garbage collection by a large factor, doing without this optimization is a heavy cost.

The Mercury profiler uses the source location of the code that constructed the closure as the key, since this option gives the best tradeoff between preserving the most *useful* information and incurring the least overhead.

We handle calls to type class methods very similarly to calls through procedure-valued variables. The only difference is that the procedure `prepare_for_method_call`, to which we insert a call before type class method calls, gets passed the address not of a closure, but of a *dictionary*, a data structure that contains pointers to the entry points of the procedures implementing the methods of the type class [12]. The code of `prepare_for_method_call` itself is identical to the code of `prepare_for_higher_order_call`, except that it uses different code to extract a key from its first argument.

The choices for keys for method calls are very similar to the choices for keys for calls through procedure-valued variables: the address of the called procedure, the id of the source location that constructed the dictionary, and the address of the dictionary itself. The issues are very similar as well, despite the more complex structure of dictionaries (they contain pointers to the dictionaries of superclasses, something for which there is no equivalent in closures), and our choice is therefore the same: we use the id of the source location that constructed the dictionary as the key.

While in Mercury, unification and comparison are defined on values of every type by default, in languages such as Haskell, they require an explicit "deriving (Eq,Ord)". Therefore it is not surprising that we handle calls to the generic unification and comparison procedures the same way as we handle typeclass method calls. Their service function, `prepare_for_generic_call`, works the same way as `prepare_for_method_call`, the only difference is that it extracts the key from a dictionary structure that happens to have different format from ordinary typeclass dictionaries.

4.8 Handling recursion through higher order calls

Higher order calls often add an arc to the call graph that was not visible at compile time. These arcs can lead to cycles, which we handle in the way we described in section 4.5. The following program fragment contains a cycle that includes an arc created by a higher order call. We call such a cycle a *higher order cycle*.

```
:- type tree ---> leaf(leaf_data) ; interior(list(tree)).

:- pred traverse(tree, T, T).
:- mode traverse(in, in, out) is det.

traverse(leaf(LeafData), Acc0, Acc) :-
    process_leaf(LeafData, Acc0, Acc).
traverse(interior(Children), Acc0, Acc) :-
    foldl(traverse, Children, Acc0, Acc).

:- pred foldl(pred(T, U, U), list(T), U, U).
:- mode foldl(pred(in, in, out) is det, in, in, out) is det.
```

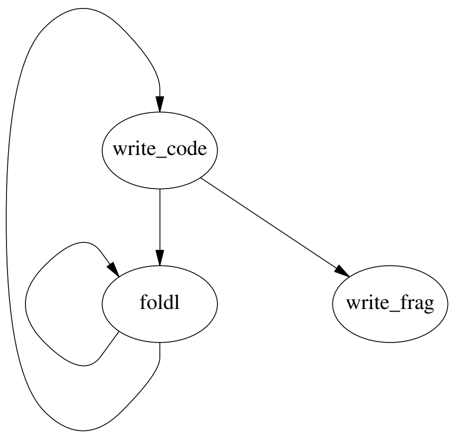

Fig. 13. A higher order cycle

```
foldl(_, [], Acc, Acc).
foldl(Pred, [Head | Tail], Acc0, Acc) :-
    Pred(Head, Acc0, Acc1),
    foldl(Pred, Tail, Acc1, Acc).
```

The higher order cycle exists because `traverse` calls `foldl` at a first order call site while `foldl` calls `traverse` at a higher order call site. The self-recursive call in `foldl` creates a second cycle in this program fragment, but this second cycle is not a higher order cycle because it does not involve any higher order call sites. Both cycles are visible in figure 13, which contains the call graph created for the above program fragment by the profiler mechanisms we have described in sections 4.5 and 4.7. This graph corresponds to the programmer's expectations of what the call graph should look like.

Unfortunately, this is not always the case for the call graphs created by the profiler mechanisms we have described so far. For some programs, these mechanisms create higher order cycles where the programmer doesn't want them. As an example, consider the following program fragment, which is a simplified version of a part of the Mercury code generator.

```
code_gen(conj(Goals), ...) :-
    foldl(goal_gen, Goals, ...).
code_gen(call(Call), ...) :-
    call_gen(Call, ...).

call_gen(Call, ...) :-
    ..., foldl(save_var, Vars, ...), ...
```

The programmer undoubtedly meant the call to `foldl` in `call_gen` to be just a convenient way of avoiding having to write yet another predicate to loop over a list, such as `save_vars`:

```
call_gen(Call, ...) :-
    ..., save_vars(Vars, ...), ...

save_vars([], ...)
save_vars([Var | Vars], ...) :-
    save_var(Var, ...),
    save_vars(Vars, ...).
```

In the programmer's mind, the version of `call_gen` calling `foldl` and the version of `call_gen` calling `save_vars` are semantically identical, and, modulo the overhead of the higher order calls

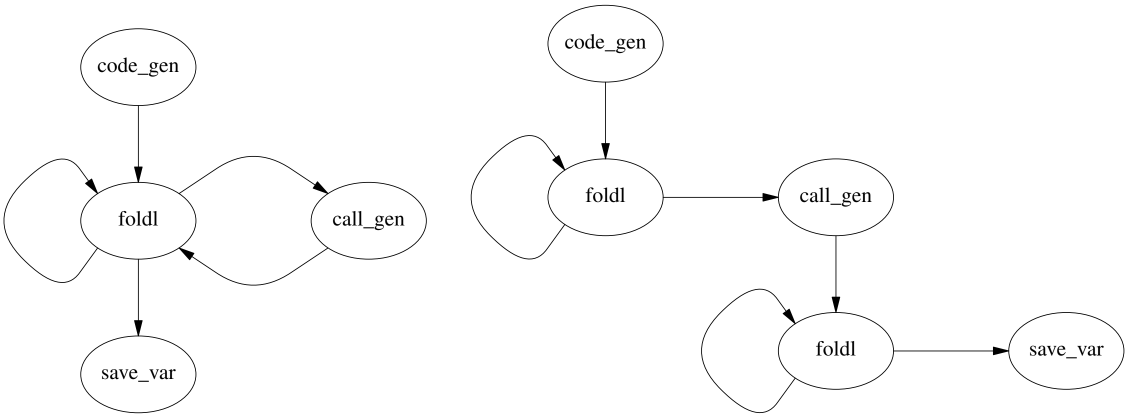

Fig. 14. A spurious higher order cycle (*left*) and the programmer's intended interpretation (*right*)

made by `foldl`, they should have the same performance and result in the same profile. However, with the mechanisms we have described so far, the version calling `foldl` will result in the call graph shown in the left half of figure 14, while the version calling `save_vars` will result in the call graph shown in the right half of that figure. The figure on the left is clearly not what the programmer would expect.

In general, the profiler cannot be sure what the programmer would expect, since it is possible to come up with programs for which two programmers may reasonably disagree on what they expect. However, we believe that the profiler and its designers may reasonably expect consensus in a fairly large and important set of cases: the cases where higher order predicates are being used as parameterized control structures. In these cases, the programmer uses a single higher order predicate with a procedure-valued argument as a parameter instead of writing a whole collection of first-order predicates that each do the same thing for one particular value of the parameter. The `traverse` and `code_gen` program fragments above are both examples of this; both use a higher order predicate as a parameterized control structure. In fact, `foldl` and `map` over lists, and many other higher order predicates and functions that are commonly found in the libraries of declarative languages, are almost always used as parameterized control structures. Calls to predicates and functions such as these account for a large majority of higher order calls in most programs, so it is important for the profiler to do what programmers expect in these cases. This is to handle the program as if each instantiation of each parameterized control structure were implemented as a distinct predicate. If the transformed program fragment has a cycle, as is the case for the `traverse` example above, then we say that it has *intrinsic* recursion through a higher order call; while if the transformed program fragment does not have a cycle, as is the case for the `code_gen` example above, then we say that it has *accidental* recursion through a higher order call. We want the profiler to produce a cycle for program fragments exhibiting intrinsic recursion but not for program fragments exhibiting accidental recursion.

In fact, the Mercury implementation has long had a transformation that, given the call `foldl(save_var, Vars, ...)`, would replace it with a call such as `save_vars(Vars, ...)` after defining `save_vars` as above (although the automatically generated name of the new predicate is not as meaningful as `save_vars`). The original purpose of this optimization was to eliminate the overhead of a higher order call: whereas the original code had a higher order call to `save_var`, the code after higher order specialization has a first order call to `save_var`. However, by switching on this transformation, and one other, when generating profiling executables, we can make the call graph constructed during a profiling run contain cycles only for cases of intrinsic and not accidental recursion, at least in the absence of some rarely used programming techniques (returning higher order values from procedures and putting higher order values into data structures). The other transformation pushes the code following a branched control structure (if-then-else, disjunction or switch) into each branch if the branch instantiates a higher order variable. This turns code such as

```
p(...) :-
    ( ... ->
        P = union
    ;
        P = intersection
    ),
    <code using P>.
```

into

```
p(...) :-
    ( ... ->
        P = union,
        <code using P>
    ;
        P = intersection,
        <code using P>
    ).
```

in which each copy of the "code using P" can be effectively specialized for the relevant value of P.

One practical problem with this approach to discriminating between intrinsic and accidental recursion is that the compiler can perform the required transformation, i.e. the creation of a specialized version of the higher order predicate (which is `foldl` in both the `traverse` and `code_gen` examples) only if it has access to the source code of the higher order predicate. This is a problem, because the definition of the higher order predicate and the call to that predicate that we want to specialize are very unlikely to be in the same module, and the Mercury compiler normally has access to the source code of only one module at a time. We do have a mechanism, which we originally designed for supporting intermodule optimization, that saves some parts of the source code of a module in an optimization file associated with the module, and makes the information in that file available to the compiler when it is compiling another module. The parts of a module that are selected in this way consist of predicates that can be profitably inlined as well as higher order predicates that can be profitably specialized. Normally, a higher order predicate whose definition is above a certain threshold in size cannot be specialized *profitably*, because the increase in runtime from instruction cache misses outweighs the decrease in runtime from the elimination of the higher-order call's overhead. Since deep profiling wants to specialize higher order predicates in order to get the right call graph, it does not matter if the specialization is profitable in the sense of reducing the program's runtime, so when deep profiling is enabled, the optimization file of a module will contain the source code of *all* the higher order predicates in the module, not just the ones below the given size threshold. Doing this effectively solves the problem of distinguishing intrinsic from accidental higher order cycles for higher order predicates being used as parameterized control structures.

The problem with this solution is that it interferes with separate compilation. Normally, a change to a module requires the programmer to recompile only that module, unless the change alters the interface of the module, in which case any other module that depends on that interface must be recompiled as well. When intermodule optimization is switched on, the contents of a module's optimization file effectively becomes part of its externally visible interface. Since changes to module interfaces are relatively rare but changes to the bodies of predicates whose source code appears in optimization files are reasonably frequent, this greatly increases the average time required to rebuild the executable after a change.

Whether this is a problem depends on how many times you need to rebuild the executable with deep profiling enabled. Normally, programmers concern themselves with profiling only after the program is fairly stable, so if profiling reveals only a small number of possibilities for improvement then it is reasonably likely that the recompilation cost is acceptable. (The number of opportunities for performance improvement identified by profiling during initial development tends to have an inverse relationship with the thoroughness with which the program's designer(s) considered performance during the program's design phase.) On the other hand, if profiling reveals many possibilities for improvement, which is likely to be the case during the maintenance phase even if the original design was sufficiently performance-aware, then the cost of recompilation is likely to prove burdensome. We have therefore looked into other methods for distinguishing intrinsic and accidental higher order cycles.

One method we have looked at, which we call the signature method, aims to simulate at runtime the effect that higher order specialization would have on the call graph. The idea is that if a procedure has higher-order arguments, then we want to have separate copies of the fields of its ProcStatic structure that are used for cycle detection for each distinct combination of values of the higher-order arguments; we call each of these distinct combinations a *signature*. That way, a call to `foldl` that gives `goal_gen` as its first argument and a call to `foldl` that gives `save_var` as its first argument are guaranteed to use different ProcDynamic structures even if one call occurs inside the other.

This method incurs the overhead of searching the list of signatures on every call to a procedure with a higher-order argument which finds `current_call_site_dynamic->csd_callee_ptr` to be null. The good news about this is that recursive calls will find `current_call_site_dynamic->csd_callee_ptr` to be non-null. Unfortunately, there are two items of bad news. The first is that this cost must be incurred even for predicates that make no higher order calls but only pass around higher-order data. This is necessary to create a precise enough context to avoid the creation of unwanted cycles in the call graph; if it isn't done, then the calls `foldl(goal_gen, ...)` and `foldl(save_var, ...)` will use the same ProcDynamic structure, which will cause the profiler to build the call graph shown on the left side of figure 14. The second, more serious problem is that the cost of checking which signature, if any, in the ProcStatic structure corresponds to the signature of the current call can be quite high, because one cannot usefully construct a signature for a call using only the outermost code address in each closure; one must take into account all the code addresses inside the hidden arguments of closures as well.

Consider programs that use parser combinators [6]. Some of the closures that such programs often pass to e.g. the Kleene star combinator (which is similar to `foldl`) are simple closures with no higher order hidden arguments, e.g. a simple closure that parses a given terminal symbol. More often, though, they pass closures that do contain higher order hidden arguments. For example, the parser combinator for alternation takes two closures (parsers that parse e.g. different nonterminals), and yields a closure that tries first one closure and then the other. The closures of those two parsers are hidden higher order arguments in the closure representing their alternation. One call to the Kleene star combinator may have as its higher order argument a closure representing the alternation of nonterminals NT1 and NT2, while for another call the argument may be a closure representing the alternation of nonterminals NT3 and NT4. If the signature comparison looks at only the outermost code address in each higher order argument, then these two calls will be considered to have the same signature and will therefore share the same copy of the ProcStatic fields for detecting cycles. If one of those calls is inside the other, this means the two calls will be considered to be mutually recursive, which is obviously undesirable if in reality they are not. For example, an XML parser we have written using parser combinators contains about two dozen calls to the Kleene star combinator, of which about half use the same combinator as its higher order argument, but only three of these represent real cycles in the program. If we want to avoid creating artificial cycles for all the other cases, we must consider a signature to be defined by *all* the code addresses in higher order arguments, even the ones inside their hidden arguments. The problem of course is that if signature extraction requires scanning hidden arguments in their entirety, it will obviously take an unbounded and often large amount of time.

A potential solution to this problem is to store the signature of each closure in the closure itself. The signature will be a kind of cryptographic hash that is a function of all the code addresses in the closure and their positions within the closure, since we want to distinguish the closure `p(q, r)` from the closure `p(r, q)`. Associating each copy of the cycle detection fields in ProcStatic structures with a list of these signatures, one from each higher order argument, would reduce the cost of searching for the right copy to a manageable level. On the other hand, it would increase the cost of constructing the closures in the first place. This is a step in the right direction, because each closure is constructed just once but tends to be passed around and called more than once. Whether it is enough to make the approach practical, on the other hand, depends on the size of the extra cost imposed on closure construction, which is proportional to the sizes of hidden arguments. As we have said before, these are unbounded in size and often large, and therefore cost is as well, which means that this "solution" is not sufficient. (We could distribute the cost throughout the program by recording a signature in *every* term of the program except first-order constants, a signature-canonicalizing instead of address-canonicalizing variant of hashconsing [5], but this would change the performance characteristics of the program being profiled beyond the bounds we consider acceptable.)

Another potential solution is to limit the scan of hidden arguments to their outermost higher-order layers. By this we mean checking the type of each hidden argument. If it is not a higher order value, then ignore it; if it is a typeclass dictionary or typeinfo structure [12], then consider its code addresses to be part of the signature; and if it is a closure, then consider its code address to be part of the signature and scan *its* hidden arguments recursively. In many programs, including the XML parser, this technique will find all the code addresses in closures that a full scan would find, but at a much lower cost. This technique can be combined with the previous one by using this algorithm to create a signature for each closure when that closure is constructed. However, for programs that create deeply nested higher order terms (the nesting of closures in our XML parser is more than 150 levels deep) and/or have lots of procedures that pass around higher order terms, the cost of even this combined method is likely to be too high.

We have therefore developed a third method, which also does not yield the right call graph for all programs, but has guaranteed low overhead and avoids interference with separate compilation. We call it the zeroing approach. The approach is based on taking the heuristic "a higher order call is very likely be to a lower SCC" seriously. Taking the heuristic as a hard-and-fast rule would require zeroing the activation counters of the current procedure and all the ancestors when making a higher order call, and restoring all those activation counters after the higher order call. The requirement for unbounded memory to store the saved activation counters and for a means to traverse the call graph upward (see section 4.2) makes this difficult to implement and definitely not low overhead. However, zeroing the activation counter of the current procedure is sufficient to produce the right call graph in many cases, and zeroing the activation counter of the current procedure and its immediate parent will produce the right call graph in even more cases. The reason is that zeroing the activation counter of a procedure has an effect only if the higher order call can lead to call to that procedure, and in many programs the only cycles that go through higher order calls involve only the procedure that makes the higher order call and possibly its parent but not its more remote ancestors. Both the "zeroing the caller" and the "zeroing the caller and parent" variants require only fixed amounts of storage per higher-order call, and neither requires upward links in the call graph: the caller's ProcStatic structure is reachable through the `current_call_site_dynamic` global variable in `call_port_code`. One can also imagine 'zeroing the callee" and "zeroing the callee and caller" variants as well, but these would require a simple and fast mechanism for finding the ProcStatic structure of the procedure being called, and at the moment closures and typeclass dictionaries do not contain that information.

Zeroing is compatible with the save/restore approach to recursion detection as well as the activation counter method, although in that case the field of the ProcStatic structure being saved, zeroed (i.e. set to null) and restored is the `ps_outermost_proc_dynamic` field, not the `ps_activation_counter` field.

Of the three methods to handle accidental recursion: forced cross-module higher-order specialization, signature-canonicalization and zeroing, we have implemented the first and the third. Zeroing is the default because it does not change the program being profiled.

4.9 Handling foreign language code

As we discussed in section 3, we want to charge all the costs incurred by foreign language code against the nearest enclosing Mercury procedure that used the foreign language interface to enter the foreign language. To achieve this objective, we don't have to do anything special; the mechanisms we have discussed so far will do the right thing.

The Mercury-calls-foreign part of the bidirectional Mercury foreign language interface allows a programmer to define a Mercury procedure by giving its code in an imperative language supported by the Mercury implementation, typically C, like this:

```
:- pragma foreign_proc("C", math__sin(X::in) = (Sin::out),
                [will_not_call_mercury, thread_safe],
"
        Sin = sin(X);
").
```

The first three arguments of the foreign_proc declaration describe the language in which the definition is written, the procedure being defined and its arguments, and some attributes of the definition; the last gives the code that implements the procedure.

The Mercury compiler does not handle procedures defined by a foreign_proc declaration particularly specially, beyond requiring that they must not be also defined by other such declarations or by clauses. Instead, the Mercury compiler considers foreign language code fragments such as the one above to be an atomic goal type, just as unifications and calls are. Foreign language code goals may occur inside any procedure, not just those defined by foreign_proc declarations. Of course, only procedures defined by foreign_proc declarations start out containing such goals; indeed, the compiler considers foreign_proc declarations to be a kind of clause whose right hand side contains a single foreign language code goal. However, foreign language code goals may then migrate into other procedures as well through inlining and related source-to-source transformations. Most of these transformations do not care what kinds of goals procedure bodies contain.

The transformations we have described for wrapping `call_port_code`, `exit_port_code` and possibly `fail_port_code` and `redo_port_code` around a procedure body are also oblivious to whether the procedure body contains foreign language goals or not. They set up `current_call_site_dynamic` to point to the call site from which the currently executing procedure was called, regardless of the language in which that procedure is implemented. If the procedure is implemented in foreign language code, this will cause that call site to be charged for all the costs incurred by the foreign language procedure body and by any other foreign language code called from the procedure body.

Most procedures implemented in foreign language code do not call back into Mercury code. A few, however, do. As we discussed in section 3, we must handle such callbacks by considering them to be calls made by the nearest enclosing procedure that used the foreign language interface to enter foreign language code; in other words, by the procedure whose call site `current_call_site_dynamic` points to when the callbacks are made. Such callbacks resemble higher order calls in that the identity of the called procedure is not apparent from the "call site", which in this case is not a higher order call but a foreign language goal. However, they differ from higher order calls in that the Mercury compiler cannot insert a call to `prepare_for_higher_order_call` before them. Not only is the callback made by foreign code which the Mercury compiler cannot meaningfully modify, but the foreign language code doing the callback may not even occur inside the Mercury source files seen by the Mercury compiler; typically, it is in another source file (e.g. tcl.c) seen only by the compiler for the foreign language. Callbacks also differ from higher order calls in that a single foreign language code goal can make more than one call to Mercury procedures before it finishes executing.

We therefore handle callbacks slightly differently from higher order calls, and split the task of updating the call graph for callbacks into two parts. The first part consists of inserting a call to a new service function before every foreign language code goal that may cause callbacks to Mercury code. (The annotations on each such goal say whether this may happen or not.) This service function, prepare_for_callback, sets the new global variable current_callback_site to point to the element in the current procedure's pd_call_site_ptrs array reserved for the "call site" represented by the foreign language code goal, but does not modify current_call_site_dynamic itself. That is left for the second part of our approach, the wrapper around the Mercury procedure called from foreign language code.

Such wrappers are an integral component of the foreign-calls-Mercury part of the bidirectional Mercury foreign language interface. This allows a programmer to take a Mercury procedure, e.g.

```
:- pred tcl_status_ok(tcl_status::in) is semidet.
tcl_status_ok(tcl_ok).
```

and make it accessible from foreign language code by providing an *export declaration* such as the following. (Currently, these are allowed only for procedures that cannot succeed more than once.)

```
:- pragma export(tcl_status_ok(in), "mtcltk_tcl_status_ok").
```

Export declarations cause the Mercury compiler to define a wrapper function in a foreign language (currently C) whose name is given by the second argument of the export declaration and whose code consists of a call to the Mercury procedure identified by the first argument of the export declaration. This wrapper function handles the marshalling of the input and output arguments, compensating for the different argument passing conventions of the languages.

The second part of our approach to handling callbacks is to get the Mercury compiler to insert extra code into wrapper functions when deep profiling is enabled. The extra code we insert before the call to the Mercury procedure searches the list of CallSiteDynamic nodes starting at current_callback_site for a CallSiteDynamic node that corresponds to callbacks to this Mercury procedure. If it finds such a CallSiteDynamic node, it sets next_call_site_dynamic to point to it; if it doesn't, it creates a new CallSiteDynamic node, initializes it, links it into the current_callback_site list, and then sets next_call_site_dynamic to point to it. This code is very similar to the code of prepare_for_higher_order_call, differing only in that we use the identity of the procedure being called as the key in the list of key-value pairs, and in using the global variable current_callback_site instead of current_call_site_dynamic->csd_callee_ptr->pd_call_site_ptrs[call_site_num] to point to the memory location where that list is stored.

If a Mercury procedure called from C code itself contains calls to C code, then it can modify the value of current_callback_site. The wrapper function therefore starts and ends with code that saves and restores the value of current_callback_site.

4.10 Handling exceptions

Exceptions represent an unplanned return from a series of calls, starting from the procedure that throws the exception and continuing through zero or more other procedures until we come to a procedure that catches the exception. In Mercury, there is only one procedure that ever catches exceptions: the procedure builtin_catch. This procedure is the basic building block upon which the Mercury standard library procedure try and its variants (e.g. try_io) are built; user programs in turn must call try or its variants if they wish to make a call and catch any exceptions thrown by that call.

When a program compiled with debugging enabled throws an exception, the debugger synthesizes exception events for all the procedure calls that are aborted by the exception. The debugger knows what calls to synthesize exception events for by walking the stacks and creating such an event for each stack frame until it comes to the stack frame of builtin_catch, which it recognizes by the known return address in its callee. The deep profiler can do something similar. For each call aborted by the exception, the runtime system can do whatever exit_port_code or fail_port_code would do to restore the value current_call_site_dynamic had before the call and to undo whatever change call_port_code may have made in the called procedure's ProcStatic structure. However, instead of incrementing the counter for the number of exits (as `exit_port_code` would do) or the counter for the number of failures (as `fail_port_code` would do), we increment a counter for the number of times that control has left the procedure via an exception.

However, simulating the wrapping of procedure bodies by the deep profiling transformation is not enough; one must simulate the wrapping up of individual calls as well, for goals which are wrapped by the deep profiling transformation. This means calls that are the entry points of tail-recursive procedures (as shown in figure 7), and higher order calls when their implementation uses zeroing.

4.11 Gathering statistics

So far, we have talked exclusively about the infrastructure needed to collect profiling information during a profiling run. In this subsection we will describe what profiling information we collect.

The Mercury profiler attaches a lot more context to each measurement than most other profilers. The measurements themselves, though, are mostly conventional: counts of program counter samples, counts of memory cells allocated, counts of words of memory allocated, and they are collected by conventional means: recording program counter samples in the signal handler for the profiling signal delivered periodically to the process by the OS, and instrumenting the memory allocation mechanism. The only aspect that may be novel to some readers is that the Mercury profiler can count not only the number of times the `call_port_code` of procedure is executed in a given context, but also the number of executions of `exit_port_code`, `fail_port_code`, and `redo_port_code` as well. The reason is of course that in Mercury, some procedures can fail and some can succeed more than once, so the number of calls and the number of successful exits are not necessarily equal. The Mercury version of the invariant stating that the number of times execution enters a procedure in a given context is the same as the number of times execution leaves that procedure in that context is somewhat more complicated:

```
#call + #redo = #exit + #fail + #excp
```

Each of those five variables gives the number of times that control passes through the corresponding port. For procedures whose determinism indicates they cannot fail, `#fail` will be zero, while for procedures whose determinism indicates they cannot succeed more than once, `#redo` will be zero. (At the moment, `#excp` will *always* be zero, because the runtime system does not yet handle exceptions properly when deep profiling is enabled, and thus any program that throws an exception will get a runtime abort.)

Actually, given that the above equation is guaranteed to hold, any one of the five counts in it can be computed from the other four. Our implementation takes advantage of this fact and does not count calls; the lines of code to this effect in e.g. figure 4 were only for exposition.

`exit_port_code`, `fail_port_code` and `redo_port_code` do each increment their own counter in `current_call_site_dynamic->csd_self_profile`, and when implemented, the counter for exceptions will be incremented as well when control leaves a procedure via an exception. When deep profiling is enabled, the memory allocator increments two fields in `current_call_site_dynamic->csd_self_profile` at each memory allocation: it increments the number of cells allocated by one, and it increments the number of words allocated by the size of the cell in words. (The Mercury implementation never allocates fractions of words.) The signal handler for the profiling timer also increments its own field in `current_call_site_dynamic->csd_self_profile`. Thus none of the profiling measurements incurs any significant cost beyond the cost of maintaining the call graph itself.

Both the port counts and the memory allocation accounting are performed synchronously and precisely; multiple runs of the program with the same input will yield exactly the same results for them. By contrast, in our profiler as in most others, the timing data is collected asynchronously and the precise location of the timing interrupts will depend on factors external to the program. As a result, multiple runs of the program that gather timing statistics will produce slightly different results. However, provided that the program runs long enough, these variations will not be significant.

4.12 Creating the profiling data file

At the end of a profiling run, just before the program exits, the runtime system turns off profiling interrupts and writes out the call graph to a file. This means all the CallSiteDynamic and ProcDynamic nodes in the graph constructing during the run, and the ProcStatic and CallSiteStatic nodes corresponding to all the procedures and call sites in the program. During the profiling run, all these nodes referred to other nodes by their addresses. Since those addresses won't be the same in the profiling tool that will read in the generated profiling data file, in the process of writing out all those nodes the runtime system assigns an integer node id to all nodes and writes out pointers as the corresponding node ids. Each kind of node has its own kind of node id and a table mapping addresses to node ids, which the runtime system also uses to ensure that each node is written out only once. Actually, CallSiteStatic nodes do not need an address map table; since each CallSiteStatic structure is reachable only from exactly one ProcStatic structure, we can assign them ids that are unique by construction. Each CallSiteDynamic structure is also reachable only from exactly one ProcDynamic structure, but we have an address map table for CallSiteDynamic structures anyway in order to let us write out the nodes of the graph in a convenient order (depth-first, modified to check whether the current node has already been traversed in order to handle cliques).

Many of the fields of the nodes in the call graph contain non-negative integers. We write these integers out using a multi-byte representation in which the least significant 7 bits of each byte are data and the most significant bit is a continuation bit. With this representation, numbers up to 2^7 are represented in one byte, numbers up to 2^{14} in two bytes, and so on. As we will see in section 6.1, this simple form of compression is reasonably effective: node ids are dense allocated small numbers, the port counts and memory allocation counts in many CallSiteDynamic structures are small, and the vast majority of fields counting clock ticks contain zero.

We consider the root of the tree of ProcDynamic and CallSiteDynamic structures that we write out to the profiling data file to be an artificial procedure representing the Mercury runtime system, which has a single "callback" call site. Programs in which `main` is written in Mercury will execute three callbacks to Mercury code through this call site: one to initialize the Mercury standard library, one to call `main`, and one to finalize the Mercury standard library. In programs in which `main` is written in another language, the call to a Mercury `main` will be replaced by other calls to Mercury code made by the foreign language program. The same code in the Mercury runtime system finalization routine handles both kind of programs.

If we want a program to generate a consistent profiling data file, we must ensure that all exceptions are caught, even if the program itself does not catch the exception. To this end, the runtime system should catch any exceptions that propagate out of the toplevel calls it makes to Mercury code. If it did not, the profiling data structures may not be consistent when the Mercury standard library finalization code executes.

5 The profiling tool

5.1 Processing profile information

The information saved in the profiling data file is in a raw form, and much of the information it contains is implicit. For example, the CallSiteDynamic structures it contains give you how much time was spent inside the code of a given procedure when called from a given ancestor context, but they do not tell you give you how much time was spent inside the code of a given procedure *and its descendants* when called from a given ancestor context. The contents of the data file must therefore be processed before being shown to the programmer. This is done by a profiling tool.

The profiling tool starts by reading a data file created by a profiling run, and recreating a slightly modified variant of the call graph that existed in the profiled executable. Some of the modifications are omissions; the tool's version of ProcDynamic nodes have no `ps_outermost_proc_dynamic` or `ps_activation_counter` fields, since these are needed only to detect recursion in a running program. Some of the modifications are variations: nodes refer to each other using integers, not pointers. And some of the modifications are additions. In a profiled executable, only CallSiteDynamic nodes contain profiling measurements, and those measurements cover only the called procedure, not its descendants. Inside the profiling tool, both CallSiteDynamic and ProcDynamic nodes have a pair of fields whose values are derived from profiling measurements. These pairs consist of the `csd_self_profile` and `csd_descendants_profile` fields in CallSiteDynamic nodes and the `pd_self_profile` and `pd_descendants_profile` fields in ProcDynamic nodes.

The `csd_self_profile` field of a CallSiteDynamic structure `csd` contains the measurements derived from invocations of a given procedure when called from a given call site in a given ancestor context. The "given procedure" is `csd->csd_callee_ptr`. The "given call site" is the one identified by the element of the caller procedure's `ps_call_sites` array corresponding to the position of `csd` in the caller invocation's `pd_call_site_ptrs` array. The "given ancestor context" is identified by the chain of ProcDynamic ancestors of `csd` modulo SCCs. What we mean by that is the following. Start with the chain of procedure invocations that are the ancestors of the procedure invocation created at the call site represented by a given CallSiteDynamic structure. Convert this to a chain of ProcDynamic structures by replacing each procedure invocation by the ProcDynamic structure that represents that procedure invocation in the call graph. Find which groups of these ProcDynamic structures form SCCs; each SCC will correspond to a contiguous subsequence of the chain. Then replace each of these subsequences with its first element (the one closest to the root of the call graph), which represents the unique entry point to the SCC. The resulting filtered sequence is what we mean when we say "the ancestor context of the CallSiteDynamic structure".

The `pd_self_profile` field of a ProcDynamic structure contains the measurements derived from invocations of that procedure in a given ancestor context when called from *any* call site in that context. For recursive calls, the call site's CallSiteDynamic structure will be in the same ancestor context as the ProcDynamic structure; for non-recursive calls, the call site's CallSiteDynamic structure will be in a different ancestor context. In each SCC, there will be exactly one ProcDynamic structure that has a non-recursive call made to it; the CallSiteDynamic structure of this call is the *entry point* of the SCC.

The `csd_self_profile` fields of CallSiteDynamic structures contain two kinds of subfields. One kind consists of the counts giving e.g. the number of times the called procedure was called from the call site and ancestor context represented by the CallSiteDynamic structure. The other kind consists of the amount time spent and the numbers of cells and words allocated in the body of the called procedure when called from that call site and ancestor context. The `csd_descendants_profile` fields of CallSiteDynamic structures, which are intended to hold the total costs incurred by all the descendants of the called procedure, contain only the second set of subfields: sums of call counts, exit counts etc derived from many different procedures are not very meaningful, and they don't help programmers either understand or optimize their programs. Alternatively, one can view `csd_descendants_profile` fields as having their call, exit, fail, redo and excp counts hardwired to zero and unaffected by updates.

The `pd_self_profile` and `pd_descendants_profile` fields of ProcDynamic structures are analogous to the `csd_self_profile` and `csd_descendants_profile` fields of CallSiteDynamic structures. The `pd_self_profile`, and `pd_descendants_profile` and `csd_descendants_profile` fields all have their subfields initialized to zero before being filled in by the profiling tool's propagation algorithm, which we show in figure 15.

The first step of the algorithm fills in the `pd_self_profile` field of each ProcDynamic structure with the sum of the `csd_self_profile` fields of all the CallSiteDynamic structure that point to that ProcDynamic structure; these are all the call sites that call the procedure in the ancestor context represented by the ProcDynamic structure. Once all the `pd_self_profile` fields have been computed, they effectively become readonly.

The rest of the algorithm is concerned with filling in the `csd_descendants_profile` and `pd_descendants_profile` fields in the call graph. Since we are propagating information from descendants to ancestors, we must process ProcDynamic structures in a bottom-up order. To help us find the bottom-up order, we build a graph G whose vertices are ProcDynamic structures, and which contains an arc from one ProcDynamic structure to another (e.g. p1 to p2) if a procedure in-

```
local G : set(arc(ProcDynamic, ProcDynamic))
local T : sequence(set(ProcDynamic))
local sum : profile_info

foreach csd: csd is a CallSiteDynamic
    csd->csd_callee_ptr->pd_self_profile += csd->csd_self_profile
done

G := {}
foreach pd : pd is a ProcDynamic
    foreach csd : csd is a CallSiteDynamic in pd->pd_call_site_ptrs
        insert arc from pd to csd->csd_callee_ptr into G
    done
done
T := topological_sort(G)

foreach clique in T, in bottom-up order
    let e be the CallSiteDynamic that is the entry point to clique
    sum := 0
    foreach pd in clique
        foreach csd : csd is a CallSiteDynamic in pd->pd_call_site_ptrs
            if csd->csd_callee_ptr is in a lower clique than pd then
                pd->pd_descendants_profile +=
                    csd->csd_self_profile + csd->csd_descendants_profile
            fi
        done
        sum += pd->pd_self_profile + pd->pd_descendants_profile
    done
    e->csd_descendants_profile := sum - e->csd_self_profile
done
```

Fig. 15. Profile measurement propagation algorithm

vocation whose ProcDynamic structure was p1 made a call to another procedure invocation whose ProcDynamic structure was p2. (The inner foreach ignores slots in the `pd->pd_call_site_ptrs` array that contain null pointers, but it loops over all the ProcDynamic structures that may be pointed to by slots that correspond to higher order call sites, typeclass method call sites, generic unifications or comparisons or foreign language code.) We represent G as a set of these directed arcs, and use a topological sort to find the cliques in the graph. Each clique is a set containing the ProcDynamic structures of mutually recursive procedure invocations. The topological sort returns these cliques in bottom-up order, which means that the ProcDynamic structures in each clique may contain arcs to ProcDynamic structures in earlier cliques but not in later cliques. This means that the cliques of leaf procedures will be near the front of the sequence, while cliques close to the program entry point will be near the tail of the sequence.

The main loop of the algorithm processes each clique in bottom-up order. Its job is to transmit profiling information from lower cliques to higher ones through the `csd_descendants_profile` fields of the CallSiteDynamic structures representing calls from higher cliques to lower ones. The `csd_descendants_profile` fields of CallSiteDynamic structures representing recursive calls (where the caller and the callee use ProcDynamic structures that are in the same clique) are not touched by the algorithm and they remain zero. This is because we don't want to propagate e.g. times from one ProcDynamic in a clique to another, because we don't maintain records about ancestor/descendant relationships within cliques. (The whole point of detecting recursion is to discard this information and thereby reduce the size of the call graph to a manageable level.) Instead, for each procedure in the clique, we record the time spent in its body (in the `pd_self_profile` field) and inside the call trees of the calls it makes to procedures in lower cliques (in the `pd_descendants_profile` field).

We ignore the time spent inside the call trees of calls to procedures in the same clique, since that time will be accounted for when we add up the `pd_self_profile` and `pd_descendants_profile` fields of all the ProcDynamic structures in the clique. (We do the same thing for memory allocation information, but we will focus on times in our exposition.)

The innermost loop computes, for each ProcDynamic structure in the clique, the time spent inside the call trees of calls to procedures in lower cliques from that procedure. The middle loop and the two assignments flanking it are there to compute the time spent in the clique as a whole. This consists of the sum of all the times spent in the call trees of calls to lower cliques by procedures in this clique, and the sum of all the times spent in the bodies of procedures in this clique. This is the cost we want to transmit through the CallSiteDynamic structure of the entry point of this clique (represented by the variable `e`) to the higher level clique containing the procedure that made that call. However, a part of this cost (usually a small part) is already accounted for by the `csd_self_profile` field of `e`, which contains the time spent inside the body of the entry procedure when invoked from the entry point call site. This time, `e->csd_self_profile`, has already been included in `e->csd_callee_ptr->pd_self_profile`, which is `pd->pd_self_profile` for the entry procedure in the clique and which thus has already been included in `sum`. To maintain the invariant that all measurements below a cross-clique call site are included exactly once in the profiling information associated with that call site, we must therefore subtract `e->csd_self_profile` from `sum` before assigning it to `e->csd_descendants_profile`.

Consider the example call graph fragment in figure 16, whose nodes are ProcDynamic structures and whose edges are CallSiteDynamic structures. Table 1 shows the result of propagating the profiling data recorded in the `csd_self_profile` fields of these CallSiteDynamic structures first into the `pd_self_profile` fields of ProcDynamic structures and then into the `pd_descendants_profile` and `csd_descendants_profile` fields of ProcDynamic and CallSiteDynamic structures respectively.

The propagation algorithm never updates the `csd_descendants_profile` fields of csdBB, csdFG and csdGF, because they represent calls that do not cross clique boundaries. While csdD and csdE represent calls that do cross clique boundaries, those calls do not lead to any further calls, which is why the algorithm sets their `csd_descendants_profile` fields to zero. Only CallSiteDynamic structures that represent cross-clique non-leaf calls can have non-zero `csd_descendants_profile` fields.

The only clique in this call graph fragment that contains more than one ProcDynamic structure is the one consisting of pdF and pdG. The total cost of the clique consisting of pdF and pdG is the sum of the total costs of pdF and pdG. Since neither pdF nor pdG has any descendants, the total cost of pdF is csdF + csdGF, while the total cost of pdG is csdFG. The total cost of the clique is this csdF + csdGF + csdFG. The CallSiteDynamic structure that represents the entry point of the clique, csdF, has the same total cost but has it sliced up somwhat differently, because from its point of view, csdGF is a descendant cost, not a self cost. The same is true for pdB and csdB.

Normally, computing the total cost of a procedure requires nothing more than summing up the costs of all the ProcDynamic structures that represent calls to that procedure. For example, if pdB and pdF represent calls to the same procedure (say procedure `p`, whose execution obviously followed a different path in the two calls), then the total cost of the body of `p` is csdB + csdBB + csdF + csdGF, while the total cost of the descendants of `p` is csdD.[7]

The simple summary algorithm works as long as we maintain the condition that if the call graph contains two ProcDynamic structures for the same procedure, then neither structure is an ancestor of the other. Our recursion detection algorithm is designed to ensure this condition.

[7] Whether the costs of the procedures mutually recursive with `p` and their descendants, which in this case consist only of csdFG, ought to be included in the costs of the descendants of `p`, and whether such inclusion should be conditional on `p` being the entry procedure of the clique, are questions one could meaningfully argue either way; the "right" answer depends on what programmers expect, and different programmers expect different things. We have chosen not to include the costs of mutually recursive procedures in procedure summaries.

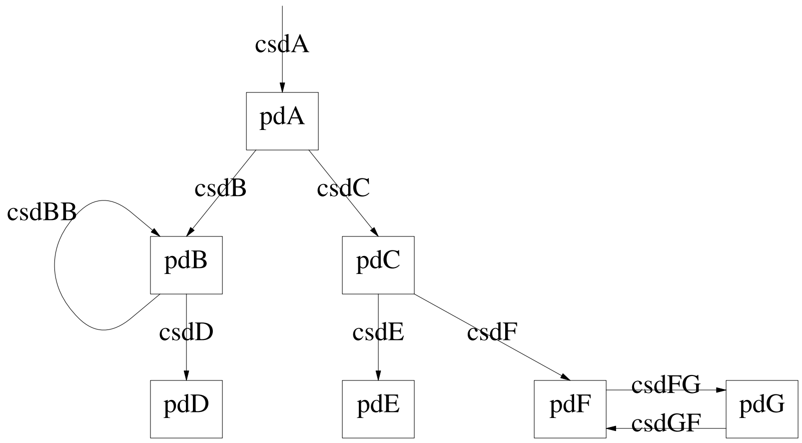

Fig. 16. An example call graph

ProcDynamic	`pd_self_profile`	`pd_descendants_profile`
pdA	csdA	csdB + csdBB + csdD + csdC + csdE + csdF + csdGF + csdFG
pdB	csdB + csdBB	csdD
pdC	csdC	csdE + csdF + csdFG + csdGF
pdD	csdD	0
pdE	csdE	0
pdF	csdF + csdGF	0
pdG	csdFG	0
CallSiteDynamic	`csd_self_profile`	`csd_descendants_profile`
csdA	csdA	csdB + csdBB + csdD + csdC + csdE + csdF + csdGF + csdFG
csdB	csdB	csdBB + csdD
csdBB	csdBB	-
csdC	csdC	csdE + csdF + csdGF + csdFG
csdD	csdD	0
csdE	csdE	0
csdF	csdF	csdGF + csdFG
csdFG	csdFG	-
csdGF	csdGF	-

Table 1. Propagation in the example call graph

However, our use of zeroing is explicitly designed to defeat recursion detection for procedures that make higher order calls, because the intuitive call graph we want to create *does* have two (or more) ProcDynamic structures for such procedures in an ancestor-descendant relationship.

Suppose that instead of pdB and pdF, it is pdA, pdC, pdD and pdF that represent calls to `p`; this is possible if csdB, csdC and csdF all represent higher order calls. The naive summary algorithm will say that the total cost of the body of `p` is csdA + csdC + csdD + csdF + csdGF, while the total cost of `p`'s descendants is csdB + csdBB + csdD + csdC + csdE + csdF + csdGF + csdFG (from pdA) plus csdE + csdF + csdGF + csdFG (from pdC) plus zero (from pdD) plus zero (from pdF). The cost for the body of `p` is right, but the cost for `p`'s descendants is wrong, because there are several costs it counts twice. From pdA, the summary should count only csdB and csdBB, since the other components of its descendant costs are accounted for by other invocations of `p`. For the same reason, the summary should count only csdE and csdFG from pdC.

To avoid this problem, we need a data structure that tells us which costs the naive algorithm counts twice, and thus allows us to compensate for the double counting. To this end, we add a new field, the `pd_compensation_table` field, to each ProcDynamic structure, and a corresponding `csd_compensation_table` field to each CallSiteDynamic structure (although we only use the

ProcDynamic	pd_compensation_table
pdA	{p → csdD + csdC + csdE + csdF + csdGF + csdFG}
pdB	{p → csdD}
pdC	{p → csdF + csdGF}
pdD	{}
pdE	{}
pdF	{}
pdG	{}
CallSiteDynamic	**csd_compensation_table**
csdA	{p → csdA + csdB + csdBB + csdD + csdC + csdE + csdF + csdGF + csdFG}
csdB	{p → csdD}
csdC	{p → csdC + csdE + csdF + csdGF + csdFG}
csdD	{p → csdD}
csdE	{}
csdF	{p → csdF + csdGF}

Table 2. Compensation tables in the example call graph

csd_compensation_table fields of CallSiteDynamic structures that represents cross-clique calls). Each of these fields contains a table that maps procedures (represented by their ProcStatic structures) to costs. An entry that maps (say) procedure p to csdD, which would be the only entry in the tables attached to pdB and csdB in our scenario, says that of the costs of the descendants of pdB, the part incurred by invocations of p is given by csdD. (The actual tables of course contain numerical costs, not algebraic variables such as csdD.) Our intention is that when the summary algorithm looks at a ProcDynamic structure, it should be able to find out how much of the costs in the pd_descendants_profile field of that structure are incurred by other invocations of the same procedure, and thus exactly what costs much the naive algorithm is counting twice, simply by looking up the procedure in the ProcDynamic structure's pd_compensation_table field.

The algorithm we use to fill in the new fields is an extended version of the last part of the algorithm in figure 15; it is shown in figure 17. Table 2 gives its output for our example.

The new code relies on two operations on compensation tables. The first operation is adding two compensation tables to produce a third. If a procedure appears in only one input table, the output table will map it to the costs that that table maps it to; if a procedure appears in both input tables, the output table will map it to the sum of the two costs it is mapped to. The second operation is overriding one table by another. If a procedure appears in only one input table, this operation's output table will also map it to the costs that that table maps it to; but if a procedure appears in both input tables, the output table will map it to same cost as the first input operand. In our algorithm, this is the one named overridetable in the only use of this operation.

The entries in compensation tables that aren't for pd_proc_static (for tables attached to ProcDynamic structures) or csd_callee_ptr-¿pd_proc_static (for tables attached to CallSiteDynamic structures) are not needed by the summary algorithm, one could thus avoid storing them. However, they are needed during propagation. Without them, there would be no way to transmit information about the costs incurred by pdD to pdA through pdB, since pdB represents a call to a different procedure.

Compensation tables are pretty small in any case. A procedure appearing in a compensation table has to satisfy two criteria: it must make higher order calls (since only such procedures are ever put into newtable), and it must appear among the descendants of the structure to which the compensation table is attached. Relatively few procedures satisfy the first criterion, and the second filters out most of the ones that do, although the filtering becomes less effective as one approaches the top of the call graph.

The extended algorithm allows for the possibility that a clique may contain two or more ProcDynamic structures for the procedure. This is rarely, but it does happen. The reason why it can happen is that the cliques that this algorithm deals with are cliques in a graph of ProcDynamic

```
foreach clique in T, in bottom-up order
    let e be the CallSiteDynamic that is the entry point to clique
    sum := 0
    sumtable := empty table
    overridetable := empty table
    foreach pd in clique
        foreach csd : csd is a CallSiteDynamic in pd->pd_call_site_ptrs
            if csd->csd_callee_ptr is in a lower clique than pd then
                pd->pd_descendants_profile +=
                    csd->csd_self_profile + csd->csd_descendants_profile
                pd->pd_compensation_table += csd->csd_compensation_table
            fi
        done
        sum += pd->pd_self_profile + pd->pd_descendants_profile
        sumtable += pd->pd_compensation_table
        if pd->pd_proc_static can ever have its activation counter zeroed then
            let newtable contain a single entry, mapping pd->pd_proc_static to
                pd->pd_self_profile + pd->pd_descendants_profile
            overridetable += newtable
        fi
    done
    e->csd_descendants_profile := sum - e->csd_self_profile
    e->csd_compensation_table := overridetable  override sumtable
done
```

Fig. 17. Extended propagation algorithm

structures, not cliques in a graph of procedures. In the absence of recursion through higher-order calls and zeroing, this procedure cannot be the entry procedure; that procedure is marked as being active for the entire duration that control resides in the clique, and thus any call to that procedure inside its own call tree will reuse the ProcDynamic structure of the entry call. However, in a clique containing e.g. procedures `p` and `q` in which the entry procedure is `p`, it is possible that `p` calls `q` from more than one call site, and that all calls to `q` from one call site have returned by the time `p` calls `q` from another call site. Since `q` would not be marked as active at that time, the algorithms of section 4 would create a new ProcDynamic structure for it.

Programmers may be interested in differentiating the performance measurements of `q`'s two call sites only in some cases, but not all. Therefore it would be useful for the profiling tool to optionally run a pass that merges the information in the two ProcDynamic structures. This would require not only replacing the two ProcDynamic structures with one containing the sum of the profiling measurements in the two originals, but also similarly merging all their corresponding descendants, down to the bottom of the graph. This may require not just merging the cliques of the ProcDynamic structures being merged, but also turning across-clique calls into within-clique calls. This is because our cliques represent actual, not potential recursion. If we want to merge two different ProcDynamic structures representing calls to procedure `p`, it may be that both invocations of `p` called `q`, but only one of those calls to `q` resulted in a call back to `p`. The merged ProcDynamic structure of the two calls to `p` will be in the same clique as the merged ProcDynamic structure of the two calls to `q`, which leaves one of the two calls from `p` to `q` within-clique but changes the other from an across-clique call into a within-clique call.

5.2 Viewing profile information

The annotated call graph produced by this algorithm is usually very big: hundreds, thousands or even tens of thousands of pages. Writing it to a file and letting the programmer look around in that file would not help programmers very much, since the call graph is useful to programmers only if they can navigate around in it easily. This is exactly what web browsers are for. We have therefore implemented our profiling tool as a server accessed from the Web through a CGI script.

Programmers can explore the information contained in a profiling data file by giving their Web browser a URL of the form http://*servername*/cgi-bin/mdprof?*datafilepathname*. This URL will cause the web server to execute the `mdprof` CGI script. The script will check whether an `mdprof` server process already exists for the named data file, and if not, it will start one. It will then send a request to the server process, relay the response to the web server, and exit.

The server process starts by reading in the named profiling data file, propagating times from descendants to ancestors using the algorithm we have just discussed and creating several kinds of data summaries (see below). It then goes into a loop waiting for and servicing requests of various kinds and returning responses as HTML pages with embedded links that, when followed, generate other requests. It exits when it hasn't received a request in a while. (The timeout period is configurable; the default is half an hour.) Any request received after that time will cause the server to be restarted, which incurs a small but noticeable delay (see section 6.2 below).

The CGI script and the server program communicate via a pair of named pipes. The names of these named pipes contain the (encoded) name of the profiling data file that the server process is serving requests for, and so do the links embedded in the HTML pages returned by the server. This allows a single web server to support several programmers viewing profiles at the same time, whether they are viewing different profile data files or the same ones.

The first page of a profile viewing session, the one identified by the starting URL above, gives the programmer a menu of choices. Some of the choices available include lists of the most expensive procedures when measured by several criteria, including time taken by the body of the procedure, time taken by the call tree of the procedure (self and descendants), memory allocated taken by the body of the procedure, and memory allocated taken by the call tree of the procedure (self and descendants). These tell the programmer where the hot spots of the program are. However, the choice that is most characteristic of the deep profiler is exploration of the call graph one clique at a time. Figures 18 and 19 show what the web page of a clique looks like. In this case, the profiled program was the Mercury compiler itself, and both pages contain the clique of a call to `optimize:repeat/11-0` [8], the procedure in the optimizer of the Mercury compiler whose job it is to iterate the application of a set of low-level code improvement algorithms to the generated code until the iteration reaches either a fixpoint or the limit on the number of iterations permitted. The iteration is useful because some optimizations introduce code sequences that can be further improved by other optimizations.

Besides the headers and column headings, a clique's page contains two parts. The top part gives all or part of the ancestor context: the filtered chain of ancestor call sites, i.e. the entry call sites of all the ancestor cliques of this clique, in order. Each line gives the location of a call and the procedure being called. In this case, the top part is most of the page. Most of the ancestors of `optimize:repeat` are in the source file mercury_compile.m. The top-level ancestor of `optimize:repeat` is course `main`, which calls `process_all_args` at line 134, which calls `process_args` at line 258, and so on. Eventually, `optimize:proc` calls `optimize:repeat` at line 59 of optimize.m; this is the entry point of the clique. Each line in the ancestor list represents a call site that is the entry point of a clique. The first two columns give the location of the call site and the name of the procedure in which it occurs; in live web pages, the procedure name itself is a link leading to the web page of the caller's clique. The remaining columns give the cost of the callee clique.

The bottom part of a clique's page contains the actual profiling information about the clique. This part contains as many blocks as there are procedures in the clique; usually one, but occasionally more. The block of the entry procedure is always first. Each block starts with a summary line, giving the cost of the procedure in this ancestor context. In this case, that procedure is `optimize:proc`. The rest of the block lists all the call sites in the procedure that were actu-

[8] In this name, `repeat` is the name of the predicate or function, `optimize` is the name of the module defining it, 11 is the arity and 0 is the mode number. For functions, the arity is given as N+1 if the function has N arguments; the +1 is for the return value.

```
Clique 185356:
                                                                            Clock ticks
Source                  Procedure                                    Self     % Total       %

Ancestors:

mercury_compile.m:134  mercury_compile:main/2-0                         0 0.00   579 100.00
mercury_compile.m:258  mercury_compile:process_all_args/4-0             0 0.00   579 100.00
mercury_compile.m:385  mercury_compile:process_args/4-0                 0 0.00   579 100.00
mercury_compile.m:394  mercury_compile:process_arg_list_2/4-0           0 0.00   579 100.00
mercury_compile.m:421  mercury_compile:process_arg/4-0                  0 0.00   579 100.00
mercury_compile.m:650  mercury_compile:process_module/4-0               0 0.00   579 100.00
mercury_compile.m:705  mercury_compile:process_module_2/6-0             0 0.00   569  98.27
mercury_compile.m:730  mercury_compile:compile_all_submodules/8-0       0 0.00   569  98.27
list.m:1159            list:foldl/4-0                                   0 0.00   569  98.27
mercury_compile.m:848  mercury_compile:compile/7-0                      0 0.00   542  93.61
mercury_compile.m:1006 mercury_compile:mercury_compile/4-0              0 0.00   167  28.84
mercury_compile.m:1815 mercury_compile:backend_pass/7-0                 0 0.00   164  28.32
mercury_compile.m:1880 mercury_compile:backend_pass_by_preds_2/8-0      0 0.00   164  28.32
mercury_compile.m:1904 mercury_compile:backend_pass_by_preds_3/10-0     0 0.00   164  28.32
mercury_compile.m:1971 mercury_compile:backend_pass_by_preds_4/11-0     0 0.00    28   4.84
optimize.m:59          optimize:proc/5-0                                0 0.00    24   4.15

Procedures of the clique:

optimize:repeat/11-0                                                    0 0.00    24   4.15

optimize.m:169         optimize:repeated/12-0                           0 0.00    24   4.15
optimize.m:172         optimize:repeat/11-0                             0 0.00     0   0.00
```

Fig. 18. A typical clique's web page: clock ticks, all ancestors

ally called through in this ancestor context. For first order call sites, there will be a single line giving the costs of the call site. For call sites corresponding to higher order calls, method calls and callbacks, there will be one line giving the cost of the call site as a whole, and one or more lines giving a breakdown of this cost based on the identity of the called procedure. In this case, there are only two call sites, both first order. The first is to `optimize:repeated`, a procedure in

```
Clique 185356:
                                                                          Port counts
Source                  Procedure                                    Calls Exits Fails Redos

The 2 closest ancestors:

mercury_compile.m:1697 mercury_compile:backend_pass_by_preds_4/5-0    220   220     0     0
optimize.m:59          optimize:proc/5-0                              220   220     0     0

The procedures of the clique:

optimize:repeat/11-0                                                  529   529     0     0

optimize.m:169         optimize:repeated/12-0                         526   526     0     0
optimize.m:172         optimize:repeat/11-0                           309   309     0     0
```

Fig. 19. A typical clique's web page: port counts, two ancestors

a lower clique, while the second call is self-recursive. In the web version of this page, the name `optimize:repeated/12-0` is a link to the page of the lower clique, while the second one is not a link at all. We could have made it a link that leads back to the same page, but such links could confuse the person viewing the profile, and in any case we have found the distinction between the appearance of links and non-links to be a useful indication of the distinction between non-recursive and recursive calls. The name `optimize:repeat/11-0` in the summary line, where it spans the first two columns, is a link, but not to a clique page; it leads to a procedure summary page, which gives the costs of *all* calls to `optimize:repeat/11-0`, not just the one from this call site and ancestor context.

As happens quite often, the port counts of figure 19 tell a useful tale, even though all the procedures involved are deterministic and therefore have exactly one exit for each call. Since `optimize:proc` is called exactly once on each procedure in the module being compiled and there are 220 calls to `optimize:proc`, the module being compiled must contain 220 procedures. There are also 220 calls to `optimize:repeat` from `optimize:proc` at line 59 in the source file optimize.m. The last line of the page shows that `optimize:repeat` also calls itself recursively 309 times from optimize.m:172, which is why the summary line reports a total of 529 calls to `optimize:repeat`. The number of calls to `optimize:repeated` is 526. Since the body of `optimize:repeat` invokes `optimize:repeated` unless the iteration has reached the limit on the number of iterations permitted, the difference between 529 and 526 tells us that we reached this limit for only three of the 220 procedures in the module being compiled. Increasing the iteration limit would therefore increase the runtime of the compiler only minimally. Reducing the limit, on the other hand, may reduce the compiler's runtime by some fraction of the 4.15% that `optimize:repeat` now accounts for. The call counts imply that each invocation of `optimize:repeat` performs an average of 529/220 = 2.4 iterations. Reducing the maximum number of iterations to 2 would reduce the time spent in `optimize:repeat` by at least the fraction 0.4/2.4 = 0.17, which would reduce the runtime of the compiler by 0.17 * 4.15% = 0.7%. (The average number of iterations cannot be higher than the maximum number, but it can be lower, since some procedures' code cannot be improved at all; in such cases `optimize:repeat` reaches fixpoint after one iteration.) Whether a 0.7% saving in compilation time is worthwhile depends on how much one values compilation time compared to the speed of the generated object code, but the deep profiler gives programmers the data they need to make an informed choice without (in this case) the need to run experiments.

Figure 20 shows the web page of a clique that is more complex in several respects. First of all, the clique contains two procedures, not one. `find_bind_var/11` has 40 calls from `find_bind_var/9` in the parent clique and 32 mutually recursive calls from `conj_find_bind_var/11`, for a total of 72; `conj_find_bind_var/11` has 29 mutually recursive calls from `find_bind_var/11` and 3 self-recursive calls from itself, for a total of 32. `conj_find_bind_var/11` contains a higher order call site, although in this case all the closures that it invokes were constructed at the same source location and call the same procedure, so the call site's summary line doesn't help much. The call counts of the various call sites in `find_bind_var/11` differ; if one wanted to, one could use the actual call counts of the procedures in the conditions and in the then-parts and else-parts of if-then-elses to find out how often the conditions of those if-then-elses succeed. This case also shows that `find_bind_var/11` calls `apply_rec_substitution/3` from two sites, one on line 497 and one on line 500, and that the deep profiling keeps separate statistics for the two call sites (although in this case the port counts of the two call sites happen to be identical.)

Figure 21 shows a procedure summary page from a program that uses Monte Carlo methods to calculate the volume of a complex shape, which is described to the program as a set of primitive shapes (cubes, spheres, cones, cylinders etc) manipulated and composed in various ways. Each basic shape is implemented as a predicate that returns true iff the point specified by its arguments is within its borders. The manipulation and composition operations are higher order predicates that take one or more other shapes as arguments. The translate procedure, for example, takes one shape argument and other arguments that specify a displacement vector, and applies that displacement to the point it is asked to test; it then makes a higher order call to its shape argument, giving it the representation of the resulting point as its arguments. The union procedure takes two shape

```
                                                               Port counts
Source                  Procedure                         Calls Exits Fails Redos

...
cse_detection.m:491     switch_detection:find_bind_var/9-0         40    40     0     0
switch_detection.m:455 switch_detection:find_bind_var/11-0        40    40     0     0

switch_detection:find_bind_var/11-0                                72    72     0     0

switch_detection.m:488 switch_detection:conj_find_bind_var/11-0   29    29     0     0
switch_detection.m:497 term:apply_rec_substitution/3-0            34    34     0     0
switch_detection.m:500 term:apply_rec_substitution/3-0            34    34     0     0
switch_detection.m:503 Unify for term:var/1                        34    28     6     0
switch_detection.m:505 higher_order_call                           28    28     0     0
                        cse_detection:find_bind_var_for_cse/7-0    28    28     0     0
switch_detection.m:507 hlds_goal:conj_list_to_goal/3-0            28    28     0     0
switch_detection.m:516 det_util:interpret_unify/4-0               10    10     0     0

switch_detection:conj_find_bind_var/11-0                           32    32     0     0

switch_detection.m:542 switch_detection:find_bind_var/11-0        32    32     0     0
switch_detection.m:546 switch_detection:conj_find_bind_var/11-0    3     3     0     0
```

Fig. 20. A more complex clique's web page: port counts, two ancestors

arguments and succeeds if the point it is asked to test is within either one of those two shapes, i.e. if either of its procedure-valued arguments succeeds on that point.

The web page in figure 21 is the procedure summary page for the union procedure. The first higher order call site in union calls four procedures; the second calls three, one of which is union itself. The line for that recursive higher order call is made somewhat peculiar by the fact that the total cost of the call is listed as zero clock ticks, even though the number of clock ticks in the body of the called procedure is not zero. The reason for this is that the `csd_compensation_table` fields of the CallSiteDynamic structures for the recursive higher order calls at this call site all state that all of their costs are accounted for by the inner invocations of union. Their costs are all accounted for elsewhere, spread among the other lines; counting them here as well would be double counting. This effect also propagates to the summary line for the higher order call site itself, where it causes the total clock ticks (106) to be less than the self clock ticks (113). Despite this counterintuitive effect, this summary shows that our algorithms do indeed preserve clock ticks (and other measurements). The time spent in the descendants of union for this run of the program is 299 + 106 = 405 clock ticks (299 in descendants from the first call site, 106 from the second), while the time spent in the body of union itself is 89 clock ticks (of which 49 were in calls to union from its own second call site, with the balance being in calls to union from other predicates). The total is 405 + 89 = 494, just as the summary page says.

Our system can display four different kinds of measurements for each procedure and call site. These are

- port counts (counts of calls, exits, fails and redos, and eventually exceptions)
- clock ticks
- number of memory allocations
- amount of memory allocated

Our system displays each measurement in its own set of columns. The port count measurement always takes four columns, because there are (at the moment) four ports, and each has a column for its count. For clock ticks, the basic four columns are the number of clock ticks in the procedure body (Self), the number of clock ticks in the procedure body and its descendants (Total), and those

Summary of procedure geom:union/5-0:

Source	Procedure	Clock ticks Self	%	Total	%
geom:union/5-0		89	8.88	494	49.30
geom.m:92	higher_order_call	124	12.38	299	29.84
	geom:translate/7-0	83	8.28	178	17.76
	geom:torus/4-0	20	2.00	30	2.99
	geom:difference/5-0	9	0.90	28	2.79
	geom:rotateY/5-0	12	1.20	63	6.29
geom.m:94	higher_order_call	113	11.28	106	10.58
	geom:union/5-0	49	4.89	0	0.00
	geom:translate/7-0	45	4.49	86	8.58
	hg:body/3-0	19	1.90	20	2.00

Fig. 21. The summary page of procedure geom:union: clock ticks

two numbers expressed as percentages of the total number of clock ticks taken by the program. These can be extended by two further self/total pair of columns, giving the time corresponding to the clock tick count (which is simply the clock tick count multiplied by the interval taken by one tick), and the time per call. The memory allocation measurement takes up to six columns, giving the number of calls to the memory allocator by the procedure body, the number of calls to the allocator by the procedure body and its descendants, and percentages and per-call counts for both of these. The display of the amount of memory measurements is analogous to the memory allocations, except that it can be displayed using either words or bytes as the basic unit. [9]

The maximum number of columns one can display at one time is twenty six: one for a source context, one for a procedure name, four for port counts, eight for clock ticks, and six each for memory allocations and amount of memory. However, such a display would not be very useful, because it would make the web page too wide to fit on the screen, especially if some of the procedure names are long. Therefore we have made the set of measurements displayed configurable, with the default set being port counts, clock ticks and words allocated. (Displaying clock ticks instead of times helps remind programmers about the granular nature of timing measurements: it is much harder for a programmer to mistakenly attribute undue precision to “one clock tick” than to “ten milliseconds”.) Programmers can toggle the display of each measurement via links in a footer at the bottom of each page. These work by asking the server to send a variant of the current page with a different set of measurements: the set of measurements to display is encoded in the URL. The URLs embedded in the body of each page generated by the server pointing to other pages (e.g. to other cliques) preserve the URL component controlling measurement selection, so that programmers' expressions of preference are persistent: having once chosen what set of measurements they want to see, they see that set of measurements in all the pages they call for until they change their preferences again.

We also this mechanism to let users express other preferences. The set of preferences includes

- The set of columns of profiling data to display.
- The limit on the number of ancestors displayed on each clique page.
- Whether at higher order call sites, the system should display a summary line and one line for each distinct callee, or just a summary line. (The latter can make it easier to get an overview of a procedure body that contains many call sites.)

[9] We never display measurements derived from descendants alone; we always display total (self + descendants) measurements instead. This is what programmers are interested in, and some cases involving compensation tables, e.g. the one in figure 21, a separate measurement for the descendants only would not make sense.

– The sorting criteria that determines the order in which the lines of the various call sites in procedure bodies are presented. The default order is lexicographic order on the source context, but one can also sort by the name of the called procedure, and by any aspect of the cost of the call site. The cost aspect is specified by three parameters: the measurement (calls, time, allocation count, amount of memory allocated), whether the costs of descendants are to be included, and whether one wants overall costs or per-call costs. One can thus sort procedure bodies e.g. on the average time spent in the body of the called procedure per call, or on the total amount of memory allocated by the called procedure and its descendants. (Many of the column headers are in fact links that lead to a version of the page sorted by the cost criterion represented by that column.)
– Whether the system should apply *contour exclusion* when generating the list of a procedure's callers.

Clique pages and procedure summary pages are only two of the page types the deep profiler can display. Top procedures pages and procedure callers pages are another two. A top procedures page lists the most expensive procedures in the program, sorted by any aspect of cost specified using the three parameters above. Once the programmer has chosen an expensive procedure he or she wants to investigate, they can click on its name, which takes them to its procedure summary page. That page itself contains links that lead to the pages listing the callers of that procedure: a page listing its calling call sites, a page listing its calling procedures, a page listing its calling modules, and a page listing its calling cliques. They can then see whether any of these calls can be eliminated.

Often, looking at the immediate parents is unenlightening. For example, the procedure `tree234:search`, which searches for a key in a 2-3-4 tree, is often one of the most expensive procedures. However, as we mentioned in section 3, this procedure is called from only one place in the source, from `map:search`. `map:search` itself is called from many places, some of which are in the compiler proper, and some of which are in the Mercury standard library (which contains, amongst others, the `map` and `tree234` modules). If one wants to reduce the total cost of `tree234:search` by eliminating calls to it, it has to be done indirectly, by eliminating calls to `map:search` and to *its* callers.

This is where contour exclusion comes it. Contour exclusion requires the programmer to provide an exclusion file, whose name is simply the name of the profiling data file and a fixed suffix. The exclusion file specifies a set of procedures. It does so by listing a set of modules, and saying whether the set of procedures should include all procedures in the module, or all the *internal* (i.e. non-exported) procedures in the module. The set of excluded procedures draws a line (a *contour*) through the program, with the excluded procedures below the line.

When a programmer clicks on a link asking for a procedure's callers, the server looks up the list of CallSiteDynamic structures that call that procedure. However, if the preferences say contour exclusion should be applied, it will then check whether the caller associated with that CallSiteDynamic structure is an excluded procedure. If it isn't, it will use the CallSiteDynamic structure as before, i.e. it will consolidate its data with others from the same call site, procedure, module or clique (depending on the request) before displaying it. If it is, it will look up CallSiteDynamic structure that is the entry point of the clique containing the caller's ProcDynamic structure, and starts again with that, until it finds an ancestor CallSiteDynamic structure whose caller is not an excluded procedure.

Another pair of interesting page types is concerned with modules. The first lists all the procedures in a given module, and gives the total costs incurred by each procedure and its descendants. Such pages are usually reached via links from the other part of the pair, which lists all the modules of the program, and for each module, gives the total costs incurred in the bodies of its procedures. It does not give the total costs incurred by the *descendants* of those procedures, because for an accurate accounting of those costs, we would need to know, for each procedure, how much of the costs of its descendants was incurred by lower level calls to procedures in the same module, so we could avoid avoid double counting those costs. We could get that information simply by omitting the test for zeroing in the extended propagation algorithm of figure 17, but doing so would greatly increase the cost of the algorithm in both space and time.

The final interesting page type is the menu page, the page you get when you start up the deep profiler. It gives you some statistics about the call graph (the number of cliques and the number nodes of each type, the number of clock ticks inside user code and inside profiling instrumentation) and offers you a choice of links to explore it. These include links leading to

- the page listing all the modules of the program,
- pages giving the top hundred procedures by some of the most commonly used cost criteria,
- the page of the root clique of the call graph, and
- the page of the (unique) clique whose execution time (self and descendants) accounts for at least 90% of the clock ticks counted in user code but which has no child cliques that also account for 90+%.

The last link is useful because it usually allows programmers to jump past any top-level procedures that deal with various aspects of option handling, and arrive with a single click at the clique of the procedure that divides up the task of the program into its components.

5.3 Security considerations

While we expect that most users of the Mercury deep profiler will use the web server on their own machine, the nature of the web means that the `mdprof` server running on one machine can be accessed from anywhere on the internet. This fact requires us as the designers of `mdprof` to ensure that it cannot be used to subvert the security of the system it is running on. In our design of `mdprof`, we addressed the three most important aspects of security: unauthorized alteration of data, unauthorized disclosure of data, and denials of service.

We ensure that `mdprof` cannot be used to modify the data stored on the machine it is running on in any way that would be detrimental to its operation by not opening any file for writing, in either the CGI script or the server process, except the named pipes that the two programs use to communicate with each other. These reside in a directory designated for public use (/var/tmp), whose contents therefore should not be depended on by any security-critical system service. The names of the named pipes themselves start with the string "mdprof", which makes it highly unlikely that the act of creating the named pipes will accidentally overwrite another program's data.

The CGI script and the server also do not open any file for reading, except the named pipes, the data file named in the URL, and possibly its contour exclusion file. If the data file does not start with the magic cookie that profiled executables put at the start of profile data files, the server exits after printing an error message without revealing any of the file's contents. We generate the same error message if the named file does not exist, so one cannot use the text of the error message to check for the existence of files. (The check for the existence of the cookie takes only a little longer then the test for the existence of the file, so the *timing* of the error message doesn't reveal the existence of files either.) The server process will not be able to open the file if its permissions, or the permissions of its parent directories, deny access to the user id under which the web server executes CGI scripts, which in practice means that the server can open only world-readable files. The same considerations apply to the contour exclusion file. An outsider can therefore use `mdprof` to gain access to the content of a file only if that file is a world-readable profiling data file and he or she already knows or can guess its pathname.

The upper bound on the amount of computation that an invocation of the CGI script can do is very small; the script contains no loops (recursive predicates) at all, except those that iterate over (parts of) the URL. The only way to mount a denial of service attack by causing the CGI script to consume machine resources is therefore saturating the network over which requests for web services arrive. Since whether an attacker can perform such a saturation is independent of whether the machine in question has `mdprof` installed, the CGI script does not increase the risk of a denial of service attack.

The server process is a different question. Once it has been created, it stays around until its timeout expires. A dormant server process does not consume CPU time, but it does consume significant amounts of virtual memory. An attacker who can create server processes at will would therefore be able to run the target machine out of virtual memory. There are two defenses against such attacks. First, the server process will exit immediately if it cannot open the specified file during its initialization, or if the file's format does not match the format expected of profiling data files, thus tying down only minimal VM resources for a minimal time. Since the CGI script ensures that at most one server process at a time exists for each profiling data file, the number of server processes an attacker can create is limited by the number of profiling data files already on the filesystem of the target machine whose filenames the attacker can discover or guess. In the absence of a previous penetration of security, this will be a very small number. If this is not sufficient reassurance, one can also employ a second line of defence, which is to give the CGI script a limit of `mdprof` N servers, and make it refuse to create new ones if N servers are already active.

6 Performance evaluation

There are two aspects of performance one must consider in the design of a profiling system. First, how does enabling profiling affect the performance of user programs? Second, what is the performance of the profiling tool that analyzes profiling information and presents it to the programmer? We answer these two questions in the two subsections of this section. All the benchmark results we report were obtained on a Gateway 5300 XL laptop with a 700 MHz Pentium III CPU and 256 Mb of memory, running Linux 2.2.18pre21.

6.1 Performance of profiled executables

To gauge the performance impact of enabling deep profiling, we have compiled the same Mercury program with eleven different sets of different compilation options. The results are reported in table 3. Each row of table 3 reports on one set of compilation options and gives the speed and size of the resulting executables. The Mercury program happens to be the Mercury compiler itself; during the timing runs, its task was to compile to C code the largest Mercury module we have access to, make_hlds.m (a part of the Mercury compiler), which contains 7958 lines.

The meanings of the columns specifying the compilation options are as follows.

grade The Mercury runtime system and execution algorithm have several optional features, such as debugging, gprof style profiling and deep profiling. The grade selects a subset of these features, giving the executable access to the features it needs without incurring overhead to support the features it does not need. The basic grade is asm_fast.gc. It enables the use of the Boehm garbage collector but no other optional features, and is optimized for speed. The grade asm_fast.gc.debug allows the program to be debugged (similar to the -g switch on many C Unix compilers). The grade asm_fast.gc.prof prepares the program for time profiling using the old Mercury profiler `mprof`, which is a clone of gprof. The grade asm_fast.gc.memprof prepares the program for memory profiling, making the executable record, at every memory allocation, the identity of the procedure that performs the allocation. The grade asm_fast.gc.profdeep prepares the program for deep profiling using the algorithms discussed in this paper.

recursion detection Specifies which of the two methods described in section 4.5, the save/restore mechanism or the activation counting mechanism, the executable uses to detect recursion. This option is meaningful only if deep profiling is enabled.

preserve tail recursion Specifies whether the compiler uses the modified transformation presented in section 4.6 to preserve self-tail-recursion. This option is meaningful only if deep profiling is enabled.

h.o. zero Specifies whether the compiler generates code to zero the activation counter of the caller's ProcDynamic structure (or, equivalently, to set its outermost ProcDynamic pointer field to null if using the save/restore mechanism for recursion detection) when making higher order calls, as described in section 4.8. This option is meaningful only if deep profiling is enabled.

#	grade	recursion detection	preserve tail rec.	h.o. zero	h.o. spec.	run time	run time no write	executable size
1	asm_fast.gc	N/A	N/A	N/A	default	16.6s	16.7s	6,009 kb
2	asm_fast.gc	N/A	N/A	N/A	maximum	16.5s	16.5s	6,095 kb
3	asm_fast.gc.debug	N/A	N/A	N/A	default	38.8s	39.0s	34,266 kb
4	asm_fast.gc.prof	N/A	N/A	N/A	default	28.4s	28.5s	9,501 kb
5	asm_fast.gc.memprof	N/A	N/A	N/A	default	88.7s	88.8s	9,099 kb
6	asm_fast.gc.profdeep	save/restore	yes	yes	default	44.7s	40.4s	13,719 kb
7	asm_fast.gc.profdeep	save/restore	no	yes	default	45.0s	40.4s	12,615 kb
8	asm_fast.gc.profdeep	activation count	yes	yes	default	45.3s	40.7s	13,381 kb
9	asm_fast.gc.profdeep	activation count	no	yes	default	45.3s	40.7s	12,269 kb
10	asm_fast.gc.profdeep	save/restore	yes	yes	maximum	44.5s	40.0s	13,935 kb
11	asm_fast.gc.profdeep	save/restore	yes	no	maximum	44.3s	39.8s	13,901 kb

Table 3. Benchmark results

h.o. specialization Specifies how much specialization of higher order predicates the compiler performs. By default, there are limits on the size of the higher order predicates the compiler will specialize, in order to limit the cost of the specialization in terms of both memory consumption and instruction-cache misses. However, as described in section 4.8, forcing the specialization of even large predicates may yield better profiling information.

Apart from the options listed above, there was no difference in how the various versions of the program were generated. In particular, they were all compiled with the default optimization level. The options listed therefore account for all the differences one can see in the table.

The times shown were obtained by obtained by executing each test eight times, discarding the lowest and highest times for each test, and taking the average of the remaining six times. For the times shown in the column marked "no write", we set an environment variable that disabled the writing of the profiling data files at the end of execution. This should of course affect only rows 4 to 11; the differences in the first three rows are a result of the limited reproducibility of timing measurements.

The timing results show that deep profiling increases the runtime of the program from less than 17 seconds to about 45 seconds, a factor of about 2.7. However, those roughly 45 seconds drop to about 40 seconds if we disable the saving of profiling data to disk. The code of the program being profiled itself is therefore slowed down only by about a factor of 2.4 (40.4/16.7). Given that gprof-style profiling slows down the executable by about a factor of 1.7 (28.5/16.7) while more accurate profiling systems, including cost centre profiling and Quantify, slow down executables by roughly a factor of two [16], this is not bad, since the extra overhead enables deep profiling to gather significantly more detailed profiling information. In fact, the deep profiler has substantially less overhead than the existing Mercury memory profiler, which performs two hash table lookups on every memory allocation.

The performance of deep profiling is not very sensitive to the options used to control it. The data shows that the save/restore approach to recursion leads to slightly faster code than the activation counter approach, but the difference is not great. Similarly, switching off the use of the technique for preserving tail recursion that we have presented in section 4.6 leads only to a slight slowdown, because the extra overhead of saving and restoring the old recursion depth counters almost outweighs the speedups from preserving tail recursion and from halving the number of service functions called. Lifting the limits on the size of procedures that are subject to higher order specialization produces only a small effect (in both the basic and deep profiling grades) because only a small number of procedures that can be usefully specialized are above the default size limit. The similarly limited effect of switching off zeroing is due to the small number of calls to procedure-valued variables (only 1.6% of all calls). The default set of options for deep profiling therefore uses the save/restore approach to recursion detection (for speed), enables tail recursion preservation (mostly to limit stack usage, but also for speed), uses the normal level of higher order specialization (to minimize differences between the profiled and unprofiled programs) and uses zeroing (to yield intuitive call graphs in the presence of accidental higher order recursion).

The options used also have only a minor impact on the sizes of executables compiled with deep profiling. These executables are roughly 2.3 times the size of executables compiled with the default grade, and 1.4 times the size of executables compiled with call graph profiling, due to the code of the extra calls to the service functions and the bulk of the static ProcStatic and CallSite structures. However, they are much smaller than debuggable executables. Debuggable executables contain much more static data than deep profiled executables, they contain slightly more calls to service functions, and (for reasons explained in [17]) each such call is followed by extra instructions.

As we mentioned in section 4.2, we set a flag during the execution of profiling primitives, and we charge profiling interrupts to `current_call_site_dynamic` only if this flag is not set. During most profiling runs, this mechanism causes somewhere between 25% and 30% of profiling interrupts to be discarded. A mechanism that could discard all profiling interrupts that arrive during the execution of profiling overhead, leaving only the profiling interrupts that the original, unprofiled program would have received, would have to discard about 58%, leaving only 42% (mirroring the execution time ratio of roughly 140% overhead to 100% original code).

There are two reasons why our mechanism compensates only for about half of the profiling overhead. The first is that much of the overhead is in the form of storing values returned by profiling primitives on the stack and later putting them back into registers for calls to other profiling primitives. Such data shuffling is carried out by the same mechanisms that perform data movement on behalf of the user program, and the deep profiling transformation cannot tag the extra data shuffling it causes because data shuffling of any kind is invisible at the level of source-to-source transformations. (In fact, even notions such as stack slots do not meaningfully exist at that level.) An implementation of deep profiling that operates at the level of the code generator could avoid this problem, by setting/clearing the flag not at entry to or exit from profiling primitives but before the first load and after the last store around the call to the profiling primitive.

The second reason is much more intractable. The presence of profiling increases the sizes of stack frames and procedure bodies and forces values that could otherwise remain in registers to be saved/restored in stack slots across calls to service functions. It thus increases the stresses on the cache system and (in extreme cases) on the virtual memory system. It also changes (usually for the worse) the effectiveness of pipelines, branch prediction mechanisms, and other architectural features designed to speed execution. With modern CPUs, such Heisenberg effects are unavoidable in any profiler that relies on code instrumentation, a requirement that only profilers using specialized hardware can escape.

One consequence of this fact is that when programmers modify the program in an effort to speed it up, they must check that the change reduces runtime not only with profiling enabled, but also with profiling disabled. One of our performance tests a few years ago showed that an optimization that only ever removed instructions from the program, and did not add any instructions or modify any of the remaining instructions, actually reliably *reduced* the speed of the program on a specific given input. The culprit much have been cache effects, specifically increased conflict misses in the machine's direct mapped instruction cache; the new layout of the program's instructions must have caused two frequently executed pieces of code that previously mapped to different cache blocks to now map to the same cache blocks. As long as such effects exist, the cost/benefit ratios of efforts to improve the "masking out" of *explicit* profiling overhead are not very favourable.

Overall, deep profiling has about the same level of time overhead as debugging. This is not very surprising, since the implementations of both consist mainly of adding calls to service functions at various points in procedure bodies. Both transformations add calls at procedure call, exit, redo and fail. Debugging also inserts code at entries to the branches of branched control structures (if-then-elses, disjunctions, switches), while deep profiling inserts code at calls of all kinds.

In the case of debugging, all the inserted calls call the same service function, which returns immediately if (as is the case in the test runs from which row 3 in the table is derived) the program is not being run under the Mercury debugger. In the case of deep profiling, the inserted calls call the service functions listed in table 4. The various subsections of section 4 have discussed all these service functions, with the exception of `builtin_call_port_code`, which relates to the special

`prepare_for_normal_call`	allocate new CallSiteDynamic	220,553
	reuse existing CallSiteDynamic	78,224,779
	total	78,445,332
`prepare_for_higher_order_call`	allocate new CallSiteDynamic	2,339
	reuse existing CallSiteDynamic	1,903,738
	total	1,906,077
`prepare_for_method_call`	allocate new CallSiteDynamic	481
	reuse existing CallSiteDynamic	923,534
	total	924,015
`prepare_for_generic_call`	allocate new CallSiteDynamic	33,091
	reuse existing CallSiteDynamic	12,939,267
	total	12,972,358
`prepare_for_callback`	allocate new CallSiteDynamic	3
	reuse existing CallSiteDynamic	0
	total	3
`prepare_for_tail_call`	allocate new CallSiteDynamic	26,106
	reuse existing CallSiteDynamic	22,434,412
	total	22,460,518
`call_port_code`	allocate new ProcDynamic	214,437
	reuse outermost ProcDynamic	37,827
	reuse ProcDynamic in call site	92,455,370
	total	92,707,634
`builtin_call_port_code`	allocate new ProcDynamic	4,203
	reuse ProcDynamic in call site	1,535,948
	total	1,549,151

Table 4. Node reuse statistics

status of unification and comparison. In Mercury, unification and comparison are defined on values of every type, even in the absence of an explicit declaration such as Haskell's "deriving (Eq,Ord)". Unification and comparison on user-defined types are implemented as calls to compiler-generated Mercury procedures, which deep profiling transforms just as it transforms all other procedures. Unification and comparison on builtin types are implemented in the Mercury runtime system. When polymorphic code tries to unify or compare two values whose type (actually, whose top type constructor) it does not know, it calls the generic unification or comparison code in the runtime system, which forwards the call to the unification/comparison routine appropriate to the actual type of the two values. Therefore, while the rest of the compiler treats calls to the generic unification and comparison procedures as ordinary calls, deep profiling treats them as a special kind of method call, and prefaces them with a call to the `prepare_for_generic_call` service function. When the generic unification or comparison procedure steers the call to the compiler-generated type-specific unify or compare procedure, the call to `prepare_for_generic_call` will be followed by the `call_port_code` of that procedure. When the call is steered to the unify or compare code in the runtime system for a builtin type, it will be followed by an "call" to the service routine `builtin_call_port_code`. The "call" is in quotes because the call is in fact inlined in the hand-written code of the runtime system; this additional code is enabled only in deep profiling grades. Similar code also records the success or failure of the call.

Table 4 presents statistics about the behavior of each service function. These statistics, and those in tables 5 and 6, were derived from a test run that is identical to the test run reported in row 6 of table 3 except that the executable being tested had compiled into it extra code to collect statistics.

The test run included 116,708,303 calls of various types. Of these, 22,460,518 of these were self-tail-recursive calls, which involved only a call to `prepare_for_tail_call`. The other calls involved both a call to one of the other `prepare` functions in the caller and a call to either `call_port_code` or `builtin_call_port_code` in the callee.

node type	number of nodes	size per node	total size
CSD nodes	282,573	32 bytes	9,042,336 bytes
PD nodes	218,641	8 bytes	1,749,128 bytes
PD array slots	696,626	4 bytes	2,786,504 bytes
list nodes	35,914	12 bytes	430,968 bytes
total	1,233,754		14,008,936 bytes

Table 5. Call graph size breakdown

call graph in memory (dynamic parts only)	14,008,936 bytes
call graph on disk	8,196,174 bytes
call graph on disk, gzipped	4,067,527 bytes

Table 6. Call graph size statistics

The table tells us that in almost all cases, the various `prepare` functions can reuse an existing CallSiteDynamic structure left in their slot by a previous call, and that `call_port_code` and `builtin_call_port_code` can similarly reuse an existing ProcDynamic structure left in their parent CallSiteDynamic structure by a previous call. The fraction of service functions that require memory allocation is well below 1%: only 0.24% of calls to service functions that can allocate a CallSiteDynamic structure do so, and only 0.18% of calls to service functions that can allocate a ProcDynamic structure do so. These proportions are very similar to the analogous measurement reported for cost centre stack profiling [11]. This is not surprising. Numbers substantially higher than these (say in the whole percent range) would require a program whose activity is much more evenly spread than it is in most typical programs, not just through the code of the program but also through the possible set of ancestor contexts.

The program being profiled normally has a working set around 20 Mb, of which about 6 Mb is code and read-only data, and the rest is writeable data. Deep profiling roughly doubles both components. Table 5 gives a breakdown of the data memory overhead. The bulk of the extra 14 Mb of data consists of CallSiteDynamic nodes, and much of the rest is `pd_call_site_ptrs` arrays. Only about 40% of the slots in those arrays contain valid pointers to CallSiteDynamic nodes. A small minority (at non-first-order call sites) contain pointers to list nodes; the rest contain null. Had we made the arrays contain CallSiteDynamic structures themselves instead of pointers to them, the call graph would be significantly bigger.

Table 6 shows that the simple compression system we have described in section 4.12 is fairly effective; it reduces the size of the call graph by about a factor of two in the process of writing it out to the profiling data file. (There is no simple way to calculate the total sizes of the ProcStatic and CallSiteStatic structures in the executable or in the running program, because some components of that static data (e.g. strings) are also needed by other, non-profiling-related data structures. However, the sizes of these structures are included in the measurement of the size of the profiling data file.) In the future, we intend to see whether we can use a compression library to reduce the size of the data file still further. The data shows that this could compress the file by more than a further factor of two. The question is whether the use of more sophisticated compression techniques would produce excessive slow down when writing out the data file in the profiled executable and when reading it back in the `mdprof` server process.

The prepare functions for non-first-order calls include code to search a list of procedures called from the same call site in the caller's ProcDynamic structure. Our instrumentation shows that these list searches are *very* short. During the test run above, none of the list searches in the `prepare_for_higher_order_call`, `prepare_for_method_call`, and `prepare_for_callback` functions ever searched a list that contained more than one element, which means e.g. all calls through a given higher order call site in a given ancestor context always called the same procedure. (Two calls from the same call site have called different procedures, but they have done so from different ancestor contexts.) Even within the `prepare_for_generic_call` procedure, the maximum search length is only five, with 95% of searches stopping after the first test, and 4% stopping after the second. It is therefore not surprising that a simple linear search is best; our experiments with using move-to-front lists instead have produced no detectable timing changes. The same is true for our experiments with the use of different keys.

We have tried quite hard to reduce the overhead of deep profiling to as low a level as possible. The area that superficially looks most promising for further reductions is our use of two calls to service functions, one in the caller and one in the callee, for each (non-self-tail-recursive) call in the original program. Changing this to use only a single call to a service function in the caller would require several changes. Most obviously, the representation of closures would need to be modified to include a pointer to the ProcStatic structure of the procedure they encapsulate. The typeinfo structures from which the generic unify and compare routines extract the addresses of the type-specific unification and comparison procedures, and the typeclassinfo structures from which method calls extract the addresses of the methods would need similar changes. However, a more important problem is that eliminating `call_port_code`also eliminates the ability to pick up TopCSD, which deprives `exit_port_code`, `fail_port_code` and `redo_port_code` of their main input. We could fix this by adding upward links to the call graph, or by doing the work of `exit_port_code`, `fail_port_code`and `redo_port_code` in the caller rather than in the callee. Unfortunately, both of these fixes have their own costs, and are likely to be at least as expensive as the scheme we have presented.

Going the other way, moving the functionality of `prepare_for_normal_call` into `call_port_code` looks more likely to yield speedups. It would require passing an extra argument with every call to identify the location containing the pointer to the call's CallSiteDynamic structure, if it has one. For first order calls, this would mean passing the address of `current_call_site_dynamic->csd_callee_ptr->pd_call_site_ptrs[call_site_num]`, which could be done by the code generator. For higher order calls, we would still need something like `prepare_for_higher_order_call`, `prepare_for_method_call` etc to search the list of callees, to create a new list node if necessary, and to return the address of the CallSiteDynamic pointer inside the right list node to be passed as the extra argument, since the callee can't know whether it was called from a first order call site. However, eliminating the overhead of the second service function call even from just first order calls may yield useful speedups. The downside is that this scheme would require integrating deep profiling into the code generator.

6.2 Performance of the profiling tool

The execution of the profiling tool can be divided into two phases: the execution of startup code and the execution of the request processing loop. In our implementation, the request processing loop is not a problem for performance: each request gets only about a pageful of data, and that pageful does not need a significant amount of computation; it needs only to be looked up in data structures that have already been constructed by the startup code, and formatted. The reason why we do things this way instead of summarizing data on demand has to do with the pages at which the programmer may start his or her exploration of the call graph: the page showing the clique of the top-level call to `main/2`, the page showing the most expensive procedures and the page showing the most expensive call sites. Displaying any of these pages requires work (propagating profiling information all the way to the top) that subsumes the work required by other page requests. If the work has to be done before displaying the first page anyway, the system may as well save the results of the work instead of repeating the work later.

The time taken by the startup phase has five main components:

- the time taken to read in the profiling data file,
- the time taken to post-process the call graph, explicitly recording relationships that the profiling data file left implicit in the interest of compactness,
- the time taken to find the cliques of the call graph and to link cliques to procedures and vice versa,

- the time taken to propagate profiling data up the call graph using the algorithm of figure 15,
- and the time taken to summarize the data in the call graph in various ways, e.g. to group together the ProcDynamic structures that represent calls to the same procedure and thus find the total cost of that procedure.

With our current implementation, all this takes about 22 seconds for the roughly eight megabyte profiling data files produced by the test runs of section 6.1. We consider this performance to be eminently acceptable; even profiling tools that generate much less useful profiling information, such as gprof, take significantly longer to digest their raw profiling data.

Reading in the data file accounts for about a third of the startup time, while the other four components account for roughly a sixth each. The reason why ancillary functions such as post-processing consume so much time relative to e.g. the propagation algorithm is that the startup code was not our main focus in optimizing the server. After its startup phase, the server process spends most of its lifetime waiting for requests. Given all the precomputation in the startup phase, the amount of work the server has to do for each request is so small that request processing is not speed critical; even a slow implementation would be fast enough for the human viewer. On the other hand, the server occupies space in main memory or on disk for its entire lifetime. We have therefore designed the main persistent data structures of the server process (those which are needed after the startup phase) for compactness more than for speed. For example, the profiling information associated with each CallSiteDynamic structure contains seven integers: four port counts (call, exit, redo, and fail), a count of clock ticks, a count of memory blocks allocated, and a count of memory words allocated. Many of these integers are often zero, for several reasons.

- Only a tiny minority of CallSiteDynamic structures contain a nonzero count of clock ticks.
- A large fraction of procedures do not contain code to allocate memory, and thus their memory blocks and memory words fields will be zero.
- In most programs, only a small minority of procedures can succeed more than once. Since only procedures that can succeed more than once can be backtracked into, almost all redo fields will contain zero.
- In most programs, only a (reasonably large) minority of procedures can fail, and thus most fail fields will contain zero.

Our representation of profiling measurements therefore is not a simple tuple of seven integers. Instead, it is a data type with several alternative function symbols. Each function symbol implies that a particular set of measurements is zero, and its arguments include only the remaining measurements. Using the normal Mercury representation, this means that each set of measurements is implemented as a tagged pointer to a block of integers, the tag specifying the alternative and thus both the size of the block of integers and the meaning of each integer in the block. The average number of integers per block is much smaller than seven, especially given that all alternatives represent the number of calls implicitly (i.e. computed from the other port counts).

Even using this space optimization, the server process needs around than 110 Mb of memory when viewing e.g. the 8 Mb profile generated by the test runs of section 6.1. This is not very surprising, since those 8 Mb were compressed from a 14 Mb original, and the server's data structures record explicitly, for quick access, many pieces of information that were only implicit in the 14 Mb original, including propagated times, the list of CallSiteDynamic nodes that all originate at the same call site in different ancestor contexts, the list of CallSiteDynamic structures that call the same procedure, etc. However, 110 Mb is still more than we would like. We will therefore look into compressing the server's data structures even further.

7 Comparison to previous work

Flat profilers such as the original Unix profiler "prof" measure the amount of time spent in each function by sampling the program counter, and use instrumentation to count the number of times each function is called. (A flat profiler for a logic programming language such as Prolog would count exits, redos and fails as well [4].) Such profilers have low overheads, and they are also very effective at what they try to do: telling programmers where the hot spots are, so they can be speeded up. Their main drawback is that they do not tell programmers where the hot spots are usually called from, so they do not have any guidance as to which calls to hot spot functions are worth eliminating or specializing, and which are not.

Graph profilers such gprof [7] try to supply that information as well. Rather than using one call counter per function, it has a call counter per caller-callee pair. It therefore knows not only how many times a function was called, but also how many times it was called from each of its parents. However, it still has only a single counter for the time spent inside each function. This gives accurate results for a flat time profile, which answers the question "how much time was spent in the code of this function?", but leads gprof astray when it tries to squeeze out of these same counters the answer the question "how much time was spent in the code of this function and its descendants?". The reason is that `gprof` attempts to answer the second question by propagating times from the lowest levels of the call graph to the highest, distributing the time spent in each function and its descendants (if any) amongst all the callers of that function by assuming that on average, all calls to a given function take the same time. [10] Unfortunately, different calls to a function take different times, and for many programs this causes gprof to produce incorrect and misleading answers to the second question. Our experience with the first Mercury profiler, `mprof`, which is a clone of gprof, bears this out, and suggests that the more frequently polymorphism, recursion and higher-order code appears in a program, the less useful the output of a gprof-style profiler will be in answering the second question. Since in declarative languages, all interesting programs use recursion and large programs are very likely to use polymorphism and higher-order code, this problem makes gprof-style profilers almost unusable for such languages.

Ponder and Fateman [13] describe a modification to graph profilers that avoids this problem and allows accurate answers to the second question. Their approach associates with each function in the program a variable that represents the time spent inside that function and its descendants. They then bracket each procedure body with code that subtracts the current time from the variable at call and adds it back at return. The total effect is of course to increment the variable by the time spent inside the function and its descendants. In order to avoid double-counting the time spent inside recursive calls to the same function, the decrement/increment code in the function prolog/epilog is executed only for top-level (i.e. non-recursive) calls to the function.

The first two questions are not the only ones that programmers may wish to have answered by a profiling tool. Two others are "how much time was spent in the code of this function when called from this call site?", and "how much time was spent in the code of this function and its descendants when called from this call site?". The latter (fourth) question can be answered by a simple adaptation of the Ponder and Fateman approach, which has a counter for each call site and brackets each call site with code to decrement/increment the call site's counter. In a language without higher order calls, such a system can dispense with the per-function counters, since the time spent inside a function and its descendants can be computed by simply summing the counters associated with all the call sites that call that function. In a language that has higher order calls, a single call site can call many different functions. One can then either use separate a counter for each callee (similar to the technique we presented in section 4.7) or simply retain the per-function counters. Since retaining per-function counters slows down all functions while using per-callee counters at higher order call sites only slows down functions that make higher order calls, the former approach is preferable. Although we have not been able find publications that give details about the algorithms used by commercial profilers, we believe that some (e.g. Rational Quantify) use this kind of approach.

Deep profiling of course has all the desirable properties of Ponder and Fateman's technique and its variants, and more, at the cost of higher overhead.

[10] Actually, gprof's propagation algorithm resembles ours in that its basic units are not functions but groups of mutually recursive functions, and there is no propagation of times between mutually recursive functions. It differs from ours in that the cliques handled by gprof's algorithm are not ancestor-context-specific and do not have a unique entry point, as well as in the use of that unjustified assumption.

Cost centre profiling [16], is a profiling technique developed by Sansom and Peyton Jones, for use with lazy functional languages. Their idea is that expressions, the basic units of functional programs, may be annotated with a construct that identifies the expression as a cost centre. The cost of any evaluation is credited to the closest enclosing cost centre. By doing this, the system successfully distinguishes between calls to a given function from different cost centres. This is particularly important for lazy functional languages such as Haskell because the prevalence of higher order code means that some functions get used very frequently in a wide variety of contexts, and in ways that will vary greatly in cost.

A significant complication with cost centre profiling is that the cost of a function call is paid not in a single block of time bounded by the function call and return, but in dribs and drabs as the thunks created by the function call are evaluated over time on demand from the rest of the program. Cost centre profiling therefore has a field in each thunk that says what *cost centre* should be charged with the cost of evaluating the thunk. All inner thunks created by the evaluation of a thunk will inherit its cost centre, with the exception of expressions that create their own cost centre.

Another complication of cost centre profiling is that while it looks like it answers the fourth question, it doesn't quite do so. If you associate a cost centre with a call site, some of the cost of the call at that call site (including the called function and its descendants) will be associated with that cost centre, but some of it may be associated with other cost centres in the call tree. This means that the answer to the fourth question can be recovered only if each cost centre has a unique parent cost centre. If programmers choose which expressions are cost centres, they can try to ensure this condition. However, the condition cannot be ensured if the compiler makes every call or every function body a cost centre, which is a compilation option in the available implementations of cost centre profiling. This is a significant drawback, because in many cases, asking the compiler to make everything a cost centre is the only practical alternative to an extensive series of recompilations as one tries to find the set of function bodies or call sites that must be made cost centres in order to get useful profiling information [11].

Morgan and Jarvis developed **cost centre stack profiling** [11] as a solution to these problems. As its name implies, cost centre stack profiling associates with each measurement not just a single cost centre but a stack of the enclosing cost centres as well. The stack of cost centres is very similar to our implicit stack of ProcDynamic structures (if the cost centre expressions are entire function bodies) or to our implicit stack of CallSiteDynamic structures (if the cost centre expressions are individual function calls). However, cost centre stack profiling and deep profiling also have major differences in both the data gathered and the implementation.

The deep profiler handles call sites and procedures as different kinds of entities during a profiling run. Cost centre stack profiling makes no such distinction. Its cost centres can represent either function bodies or function calls (or any expression between those two size extremes), but it has no knowledge of the syntactic relationships between different cost centres. It thus cannot use those relationships to make the output reflect the programmer's expectations more closely, e.g. by presenting a function body and apportioning the cost of the function body among the code of the function body itself and the various calls occurring in it. Also, in its usual mode of use, which designates function bodies as cost centres, cost centre stack profiling does not distinguish different call sites to the same function in a given function body and thus does not help much with the profiling of programs that exploit polymorphism within callers (one of our requirements from section 3). The deep profiler does.

The deep profiler and the cost centre stack profiler both detect recursion, but they do so differently and do different things when they find it. When entering a new cost centre, the cost centre stack profiler searches the list of cost centres that have been previously entered from the current cost centre stack configuration, looking for the cost centre being entered right now. Usually, it will find it, and the system can then simply reuse the extended cost centre stack configuration allocated previously. The deep profiler, by contrast, does a similar search only at higher order call sites. At first order call sites, which are the vast majority, it does not do any search, which reduces overhead.

If the search shows that the program has not previously entered the new cost centre from the current cost centre stack configuration, then the cost centre stack profiler will search the cost centre stack to see whether it already contains the cost centre being entered. If yes, it will create a new configuration by effectively deleting the previous occurrence of the cost centre from the stack. This deletion destroys the call centre stack profiler's record of the entry point of the current cost centre clique, which contains all the cost centres on the stack more recent than the one being deleted. By contrast, the deep profiler preserves this information for presentation to the programmer, and it implements the recursion check in constant time. Since the cost centre stack can be quite deep even after the removal of duplicate entries, the avoidance of all those unsuccessful searches of it is a significant advantage.

One could argue that our constant time check depends on setting the values of the `ps_outermost_proc_dynamic` fields in ProcStatic structures on the way down from the root, and that this is just as linear as searching the stack up to the root. However, that argument would be wrong: given the branching factors of the call trees of most programs, the time taken by an invocation of `call_port_code` to set up a `ps_outermost_proc_dynamic` field is amortized over a large number of checks by other invocations of `call_port_code`.

On the other hand, there is one significant cost that deep profiling incurs that cost centre stack profiling does not: the creation and initialization of the `csd_callee_ptr` arrays. However, this happens only when a procedure is entered from a given ancestor context for the first time, and measurements of both deep profiling and cost centre stack profiling show that this occurs on less than 1% of calls. Overall, we expect deep profiling to have significantly lower time overheads than cost centre stack profiling. Unfortunately, [11] does not have figures directly comparing the speeds of executables compiled with cost centre stack profiling and speeds of executables compiled without any form of profiling; it gives figures only for the overhead of cost centre stack profiling over plain cost centre profiling, these overheads ranging from 2% to 384%. For the overhead of cost centre profiling over unprofiled programs [11] directs the reader to Sansom's thesis [15]. However, this contains only a single statement that the time overhead of cost centre profiling is a factor of 1.64 for time profiling only and a factor of 2.33 for both time and heap profiling, and does not identify the program on which those measurements were taken. A fair comparison of the Mercury deep profiler and the Haskell cost centre stack profiler would in any event be significantly complicated by the major differences in the efficiencies of the underlying execution algorithms: due in large part to the absence of lazyness in Mercury, Mercury programs are often much faster than the corresponding Haskell programs. (For a rough, non-scientific test, see [14].) This implies that a level of overhead that slows down a Mercury program by e.g. a factor of 2.4 would slow down an equivalent Haskell program by a significantly smaller factor.

A final difference between the two profiling methods is that the cost centre stack profiler does not even try to break accidental cycles through higher-order predicates. The resulting merging of what to the programmer seem to be unconnected cliques can make the presented profiling information quite confusing.

Of course, our solutions to the problem of programs containing recursion through higher order predicates are not perfect either. The only approach that we know of that can avoid distorting the profile of such programs is *call path profiling* [8, 9], which records not just the current function but also all its ancestors at each measurement. The down side of this approach is the cost of traversing the stack all the way to the root. This is not a problem for gathering timing information via profiling interrupts, since on most machines profiling interrupts occur only sixty or a hundred times a second. The cost *is* a problem for gathering memory allocation statistics, since memory allocations may occur millions of times a second. This makes traversing the stack at each memory allocation impractical, particularly since programs written in declarative languages tend to allocate memory more often and to have deeper nesting of function calls (partly due to the much greater use of recursion) than imperative programs. Hall's proposed solution is to make the profiling of memory allocation statistical as well: instead of recording every memory allocation, record only every Nth allocation, choosing a value for N that makes the overhead acceptable (e.g. N=1000). Unfortunately, statistical memory allocation information is significantly less useful than accurate memory allocation information, not so much when looking for hot spots as when exploring program behavior: for example, one cannot infer from the absence of a recorded memory allocation for a function that e.g. the arm of an if-then-else in that function that causes memory allocation was never taken. The same arguments about cost and usefulness apply with much greater force to recording counts of calls, exits, and failures.

Several systems, including MLWorks [1] and IBM's Java Virtual Machine profiler interface [20] (which is not a profiler itself but rather a piece of infrastructure on which profilers can be built), adopted another remedy. This is to record with each measurement only a fixed number of functions; in IBM's JVM, the default is the current function and its three nearest ancestors. This makes recording a context cheap enough to be done at every measurement, including every memory allocation and every method entry. However, the cost in usefulness is great. The contexts are not connected to the root of the call tree, which means that programmers cannot explore the call tree as they can with deep profiling. An even more serious problem is that the data gathered in this way is not sufficient to compute the total costs (self plus descendants) of either functions or call sites; to get that information, one must use another technique as well (e.g. Ponder and Fateman's), incurring two sets of overheads.

One cannot really compare the Mercury deep profiler with basic block profilers such as pixie and QP, with hardware profilers such as Intel's VTune, with profilers that measure issues arising from threading (e.g. lock contention), or with profilers that record non-monotonic properties such as the sets of memory blocks that are alive at particular points in program execution, since their objectives are disjoint. However, exploring how some of those objectives could be reached in a system based on deep profiling technology would be interesting future work.

8 Conclusions

Our main contributions are the following.

- The design, implementation and validation of a reasonably simple, cheap and non-distortive method for associating detailed contexts with all profiling measurements. These detailed contexts are essential for meaningful profiling information in the presence of any of several forms of polymorphism.
- The design and implementation of a profiling method that meets almost all the requirements of declarative programming languages, with the sole exception being the intuitive handling of accidental recursion through higher order calls in all cases.
- The identification of the intuitive handling of accidental recursion through higher order calls as a tough but important problem for profilers for declarative languages.
- The design of three techniques (specialization, signatures and zeroing, two of which we have implemented) that each yield meaningful profiles for large classes of programs that contain recursion through higher order calls.

We can report that the detailed contexts associated with each measurement by our new profiler has already led us to a performance problem we probably would not have found any other way; we certainly did not find it using the old, gprof-style Mercury profiler. The performance problem consisted of a code fragment that checked two copies of a large data structure for equality, even though the two copies were guaranteed to be identical. (The Mercury determinism system did not catch the error because the extra unification was in the condition of an if-then-else, and therefore it was allowed both to succeed and to fail.) The time reported by the old Mercury profiler for the procedure containing the extra unification was not large enough to draw attention to itself, being based on average and not actual times spent inside the children of that procedure; the average times of such unifications were much lower than the actual times because most other comparisons of such data structures failed almost immediately. Even if we did decide to speed up that procedure, the profile would not have lead us to the redundant unification, since several other children of the procedure had higher average times recorded for them. However, on a deep profile the problem jumped right out: the time recorded for the procedure was large enough to draw our attention, and the time recorded for the redundant unification accounted for the vast majority of that time.

Eliminating the redundant unification sped up the program involved (the Mercury compiler) by about 10%; not bad for a one-line change.

A practical profiler needs to have a way to merge profiling data files from different runs, since finding a representative *set* of sample inputs is usually much easier than finding a single representative sample input. Our deep profiler does not yet have a way to do this, but we are in the process of providing such a mechanism; it should be a relatively simple extension of the algorithm we plan to use to recursively merge two or more ProcDynamic structures in a given clique that represent the same procedure, and their descendants.

Many compilers have a mechanism that allows them to take one or more profiling data file and use the information they contain about the typical behavior of the program to help them optimize the program more effectively. (For example, the SPEC CPU benchmark suite available from www.spec.org specifies training data as well as test data for each benchmark program, and most SPEC submissions exploit it for profiling feedback.) This could mean adjusting the optimization level for each procedure depending on the time cost of that procedure: e.g. optimizing time-consuming procedures for speed and other procedures for space. With a profiler that collects reliable information about the costs of call sites as well as procedures, it could also mean inlining a procedure in some of its call sites but not in others. However, these and other opportunities for tailoring optimization more closely to the program are worth exploiting only if the cost in compilation time is not too high. Since the cost of reading and processing a multi-megabyte profiling data file on every compiler invocation *would* be too high, in the future we intend to extend the profile viewing tool with a command to write out a summary file containing just the information the compiler needs.

One use of profiling feedback that we expect will be particularly useful in a purely declarative language such as Mercury is automatic parallelization. In imperative languages, automatic parallelization has traditionally been about the struggle to prove that e.g. two function calls are independent and can therefore be executed in parallel. In declarative languages such as Mercury which make data flow explicit, proving calls independent is usually trivial (all it requires is checking that neither consumes the other's output), and most programs contain many parallelization opportunities. The problem is that most of those opportunities are too small; the overhead of creating a work package from a call so that it can be given to another CPU often takes more time than evaluating the call would have taken. Therefore in the absence of profiling information, parallelization can (and often does) slow down the program. With profiling information about the average costs of call sites, the compiler can avoid spawning off work packages that are *consistently* too small. However, as we discussed in section 3, in many cases the cost of a call through a given call site will vary by several orders of magnitude depending on the ancestor context in which it is invoked. Deep profiling will tell the compiler in which contexts a call site is worth paralellizing. If a call site turns out to be worth parallelizing in some contexts but not others, the compiler may, if it wishes, create two versions of the procedure containing the call, only one of which spawns off the call to be executed in parallel. (It may also have to clone some of the ancestors of that procedure to make sure that the right version is called in each ancestor context.) We expect that this capability will significantly improve the speedups one can get from automatic parallelization. Even more important, we believe that a sufficiently sophisticated form of it should be able to reduce almost to zero the probability that automatic parallelization leads to a slowdown, thus making routine use of automatic parallelization practical.

The idea of deep profiling and our algorithms can be adapted to other declarative programming languages, since they share the same set of requirements. We expect the adaptation process to be reasonably simple, partly because source-to-source transforms are relatively easy to implement in compilers for declarative languages. The adaptations should need only minimal changes, and most of these changes should be simplifications. For example, Prolog has no determinism information, which means that one would have to wrap the full set of `call_port_code`, `exit_port_code`, `redo_port_code` and `fail_port_code`around every procedure body; there is no need for the optimized transformations that Mercury uses for procedures that can succeed at most once. On the other hand, in most functional languages all functions succeed exactly once, so `redo_port_code` and `fail_port_code`are unnecessary for them. For lazy languages, one could associate with each expression the address of a ProcDynamic structure (or the CallSiteDynamic structure pointing to it) the same way that cost centre stack profiling associates with each expression a pointer to a cost centre stack. The compiled code of the expression would have to know the call site numbers of the calls inside the expression and would have to preface those calls with a call to the appropriate service function, but this should be easy to arrange.

Deep profiling is also applicable to imperative languages, whether or not they are object-oriented, since their requirements tend to be a subset of the requirements of declarative languages. (For example, the expansion of C++ templates eliminates one source of polymorphism from the running program.) While profilers that focus on that subset should be able to achieve lower overheads than deep profiling, the overheads of deep profiling itself are in the same ball park as the overheads of commercial profilers for imperative languages. (Deep profiling increases the runtimes of programs by roughly a factor of 2.4, while the corresponding figure for profilers such as Quantify is roughly a factor of 2.) We expect that the additional information generated by deep profiling will be useful enough to be worth the extra overhead required to generate it in a significant fraction of cases.

Versions of the Mercury system supporting deep profiling are now available from the Mercury web site `http://www.cs.mu.oz.au/mercury` as releases of the day. The web site will soon also have a page containing a demo of `mdprof` that readers should be able to use to get feel for what it is like to explore a deep profile.

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

References

1. Nick Barnes. GC and profiling. In *Garbage collection mailing list (gclist)*. April 2001. Available from `<htpp://www.iecc.com>`.
2. Ralph Becket. Mercury tutorial, 1999. Available from `<http://www.cs.mu.oz.au/research/mercury/tutorial>`.
3. Lawrence Byrd. Understanding the control flow of Prolog programs. In *Proceedings of the 1980 Logic Programming Workshop*, pages 127–138, Debrecen, Hungary, July 1980.
4. Saumya K. Debray. Profiling Prolog programs. *Software — Practice and Experience*, 18(9):821–858, September 1988.
5. A.P. Ershov. On programming of arithmetic operations. *Communications of the ACM*, 1(8):3–6, August 1958.
6. Jeroen Fokker. Functional parsers. In *Proceedings of the First International Spring School on Advanced Functional Programming Techniques (LNCS 925)*, pages 1–23, Bastad, Sweden, May 1995.
7. S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 120–126, June 1982.
8. Robert J. Hall. Call path refinement profiles. *IEEE Transactions on Software Engineering*, 21(6):481–496, June 1995.
9. Robert J. Hall and Aaron J. Goldberg. Call path profiling of monotonic program resources in UNIX. In *Proceedings of the Summer 1993 USENIX Conference*, pages 1–13, Cincinnati, Ohio, June 1993.
10. Fergus Henderson, Thomas Conway, Zoltan Somogyi, David Jeffery, Peter Schachte, Simon Taylor, and Chris Speirs. The Mercury language reference manual. Available from `<http://www.cs.mu.oz.au/mercury/>`, 2000.
11. Stephen A. Jarvis and Richard G. Morgan. The results of: profiling large-scale lazy functional programs. In *Proceedings of the Eighth International Workshop on Implementations of Functional Languages*, pages 200–221, Bad Godesberg, Germany, September 1996.
12. David Jeffery, Fergus Henderson, and Zoltan Somogyi. Type classes in Mercury. In *Proceedings of the Twentythird Australasian Computer Science Conference*, pages 128–137, Canberra, Australia, January/February 2000.
13. C. Ponder and R. J. Fateman. Inaccuracies in program profilers. *Software — Practice and Experience*, 18(5):459–467, May 1988.
14. Richard M. Reid. Transforming Mercury to Haskell. Technical report, University of Glasgow, Glasgow, Scotland, April 1996. Available from `<http://www.dcs.gla.ac.uk/~reidrm/project/root.html>`.
15. Patrick M. Sansom. *Execution profiling for non-strict functional languages.* PhD thesis, University of Glasgow, Scotland, 1994.

16. P.M. Sansom and Simon L. Peyton Jones. Time and space profiling for non-strict, higher order functional languages. In *Proceedings of the Twentysecond Symposium on Principles of Programming Languages*, pages 355–366, San Francisco, California, January 1995.
17. Zoltan Somogyi and Fergus Henderson. The implementation technology of the Mercury debugger. In *Proceedings of the Tenth Workshop on Logic Programming Environments*, pages 35–49, Las Cruces, New Mexico, November 1999.
18. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 26(1-3):17–64, October-December 1996.
19. Jayen Vaghani, Kotagiri Ramamohanarao, David B. Kemp, Zoltan Somogyi, Peter J. Stuckey, Tim Leask, and James Harland. The Aditi deductive database system. *VLDB Journal*, 3(2):245–288, April 1994.
20. D. Viswanathan and S. Liang. Java virtal machine profiler interface. *IBM Systems Journal*, 39(1):82–95, 2000.