Calculating likely Parallelism within Dependant Conjunctions for Logic Programs

Paul Bone
pbone@csse.unimelb.edu.au

October 31, 2008

Abstract

The rate at which computers are becoming faster at sequential execution has dropped significantly. Instead parallel processing power is increasing, and multicore computers are becoming more common. Automatically parallelising programs is becoming much more desirable. Parallelising programs written in imperative programming languages is difficult and often leads to unreliable software. Parallelising programs in declarative languages is easy, to the extent that compilers are able to do this automatically. However this often leads to cases where the overheads of parallel execution outweigh the speedup that might have been available by parallelising the program.

This thesis describes a new implicit parallelism implementation that calculates the speedup due to parallelism in dependent conjunctions for Mercury — a purely declarative logic programming language. This is done by analysing profiling data and a representation of the program in order to determine when during the execution of a parallel conjunct variables are likely to be produced and consumed. This enables us to calculate how long a conjunct may have to wait for a variable to be produced, and how much parallelism is actually available in a parallel conjunction. This information should enable the compiler to parallelise a program only in places where doing so is profitable.

We expect that two of the components we implemented for our implicit parallelism analysis, coverage profiling and a generic feedback framework, will also be quite useful in other applications. For example this can help the compiler select the best calls to inline.

Contents

1 Introduction **2**

2 Background **4**
- 2.1 Mercury . . . 4
- 2.2 Parallelism in Mercury . . . 6
- 2.3 Deep Profiling . . . 8

3 Implicit Parallelisation **10**

4 Coverage Profiling **13**

5 Finding good candidates for parallelism **17**
- 5.1 Variable Use Time Analysis . . . 17
- 5.2 Building Implicit Parallelisation Feedback . . . 23

6 Supporting Profiler Directed Optimisations **26**
- 6.1 Profiler Feedback Framework . . . 26
- 6.2 Refactoring the Deep Profiler Tools . . . 27

7 Conclusion **28**

List of Figures

- 1 Internal representation of Mercury goals . . . 5
- 2 Independent and dependant parallel conjunctions . . . 8
- 3 Sequential versus parallel executions of two conjuncts A and B . . . 11
- 4 Coverage inference algorithm — Part 1 . . . 14
- 5 Coverage inference algorithm — Part 2 . . . 15
- 6 Coverage inference algorithm — Part 3 . . . 16
- 7 Variable consumption time analysis — Part 1 . . . 18
- 8 Variable consumption time analysis — Part 2 . . . 19
- 9 Variable consumption time analysis — Part 3 . . . 20
- 10 Calls `a` and `b` may be parallelised . . . 24
- 11 Parallelisation of two calls separated by a unification . . . 24
- 12 Profiler Feedback Loop . . . 26
- 13 Deep profiler data flow . . . 27

List of Tables

- 1 Mercury's determinisms . . . 6
- 2 Average compilation times for coverage profiling . . . 17

1 Introduction

The computing industry has become used to the clock speeds of our processors growing exponentially, to the extent that Moore's law [11] is often misquoted as referring to clock speed rather than the density of transistors within an integrated circuit (IC)[1]. However, improvements in clock speeds have recently slowed down sharply, and most increases in performance have been due to increases in the number of CPUs (cores) per chip — a trend that has made parallel computing accessible on commodity hardware. These trends are expected to continue, in the future most of the increases in computational power will come from parallelism.

Software developers are now motivated to write programs that can be parallelised across multiple processors. Unfortunately, traditionally it has been difficult to write software that takes advantage of multiprocessors, because such software is often error prone and expensive to develop and to maintain.

Imperative programming languages (such as C, Java and Python) require the programmer to describe the program as a sequence of actions. These actions often involve side effects that alter the state of the program. When writing parallel programs in an imperative language, the programmer must manage the execution of code with side effects so that the side effects do not interfere with other parts of the program that may be running in parallel. This means that the threads of execution in a program must be synchronised so that, for example, data is not updated by one thread while another thread is reading it. One common synchronisation method is the use of mutual exclusion locks. Locks are used to protect shared resources; a thread may lock a resource so that it can use it exclusively.

It is very easy for programmers to make mistakes when writing synchronisation code for parallel programs. Common mistakes include: forgetting to write synchronisation code at all, which creates *race conditions* — conditions in which the program may fail depending on the timing of different events; creating code that locks multiple shared resources in the wrong order, which can cause potential deadlocks; and acquiring or releasing a lock too early or too late, which may cause race conditions or poor performance.

Synchronisation becomes increasingly complex as the size of the program grows, making it difficult to build large-scale parallel programs. Software components that include synchronisation code are not *composable*. Puttting several individually-correct components together may cause locks to be acquired more than once or in wrong order, which can lead to deadlocks.

Declarative languages do not have side effects, therefore programmers are not normally required to use locks in declarative languages. When locks are required, techniques for managing them, such as Software Transactional Memory [7, 10] are available.

Programmers also face the problem that finding opportunities for parallelism within a program is often difficult. Most sources of parallelism allow the creation of small short-lived parallel computations. While there are many of these, the overheads of executing these in parallel often outweigh the benefits.

When a lot of parallelism is available (in which case the program is said to be *embarrassingly parallel*), it is easy to parallelise the program beyond the parallel

[1]See the Wikipedia article regarding the misquoting of Moore's Law `http://en.wikipedia.org/w/index.php?title=Moore%27s_law&oldid=242887237`

execution capacity of the computer it is running on. While the amount of parallelism the machine can exploit cannot increase beyond the number of CPUs, the overheads of parallel execution continue to increase. This often cripples the performance of such programs. For example a ray-tracer that creates an image 1,000×1,000 pixels in size has 1,000,000 independent computations available for parallelisation. Parallelising all of these on a four processor machine creates far too many overheads than would be necessary to parallelise this program more optimally.

Parallelism work in declarative languages has created constructs for declaring when some computations should be done in parallel. Mercury's parallel conjunction [5] is an example of this, enabling the programmer to declare that the conjuncts of a parallel conjunction must be run in parallel. Originally, this was permitted only where the conjuncts were independent of one another, that is, no conjunct uses a variable that is the output of another conjunct.

Peter Wang modified Mercury's parallel conjunction [18, 19] so that it supported the parallelisation of dependant conjunctions. This allows parallel execution of code that has parallelism available but was not parallelisable in the original implementation. Dependant conjuncts cannot run completely in parallel, the second and later conjuncts may block during their execution and wait for one or more dependant variables to be produced by a previous conjunct.

Implicit parallelism enables the compiler, rather than the programmer, to parallelise a program. Implementations exist of parallel Prolog-like languages, which include Concurrent Prolog [12–14], Parlog [3, 4] and GHC [17]. Many of these implementations executed (almost) all code in parallel. This often caused them to perform badly due to embarrassing parallelism.

Granularity control [9, 15] has since been developed to attempt to reduce the impact of parallelism overheads by reducing the amount of unnecessary parallelisations. Different methods of granularity control exist. For instance, purely compile-time methods use static analysis to determine the complexity of computations; and then transform the program so that computations are only executed in parallel if the compiler expected them to be of a particular size or larger. Run-time granularity control methods use metrics such as instruction counters to measure the time spent since a parallel computation was spawned, so that it only executes a computation in parallel after a set amount of sequential execution has been performed.

Feedback directed implicit parallelism uses information from a profiled execution of the program and the number of processors available on the target machine to make better parallelisation decisions. This aims to optimise the program for best performance when performing similar work with that number of processors available. For example, a compiler with access to feedback information is aware of the likely sizes of different computations, it is able to decide if parallel execution is worthwhile for an individual case given an estimate of the parallelisation's overheads.

Harris and Singh [8] profile the execution of Haskell programs, measuring the cost of evaluating each thunk. They have reported up to an 80% speedup compared to the sequential execution of their test programs on a quad core machine. However they were not able to improve the performance of some programs, which they attributed to there being a lack of parallelism implicit in some programs. These results show that implicit parallelism is a promising idea for improving the performance of software.

Tannier's implicit parallelism implementation for Mercury [16] selects the most expensive predicates of a program and attempts to parallelise conjunctions within them, This implementation makes use of compile-time granularity control to reduce the over-parallelism that can occur in recursive code.

Tannier's work is the first attempt at automatic parallelisation of dependant conjunctions. He estimated the cost of parallelising dependant conjunctions based on the number of dependant variables they shared. This is a naïve calculation, since the time that these variables are produced by one conjunct and consumed by the other may not correlate with the number of dependant variables. For example, the consuming conjunct may need a dependant variable before it can begin any processing which may be produced rather late, resulting in almost no parallelism as is typical. We believe that Tannier's algorithm is, in general, too optimistic about the parallelism available in dependant conjunctions.

This thesis discusses a feedback analysis required to compute the likely speedup due to parallelisation, in particular, it calculates the speedup in dependant parallel conjunctions based on how long a consuming conjunct may have to wait for a variable to be produced. We hope that doing this provides information that can be used to parallelise conjunctions with the greatest potential speedup, making their parallelisation worth the cost of the parallelism overheads. Likely speedup due to parallelisation is the likely sequential execution time of the conjunction minus the likely parallel execution time. To calculate this, we use feedback information provided by a version of Mercury's deep profiler [6] that we modified to introduce coverage profiling.

We assume readers are familiar with logical and functional programming. Familiarity with Mercury is not required since the relevant Mercury concepts are described in Section 2.1. Section 2.2 introduces the previous work on parallelism in Mercury on which this work is based, whilst Section 2.3 introduces the Mercury deep profiler. Section 3 introduces how we determine if a particular conjunction is worth parallelising and provides our motivation for coverage profiling. Section 4 describes our modifications to the deep profiler to provide coverage information. Section 5 describes the process we use to calculate the parallelism available in dependant conjunctions. Section 6 discusses a generic framework for feeding information back into the compiler. Section 7 critically discusses our contribution and provides suggestions for future work.

2 Background

2.1 Mercury

Mercury is a pure logic programming language intended for the creation of large, fast, reliable programs. While the syntax of Mercury is based on the syntax of Prolog, semantically the two languages are very different, due to Mercury's purity, type, mode, determinism and module systems. A logic program is a collection of predicates, which are defined in terms of goals of which there are several types. Goals are logical expressions that either unify variables, call other predicates or describe a logical relationship on some other goals. The abstract syntax for the internal representation of goals used by the Mercury compiler is shown in Figure 1.

goal G :	$x = y \mid x = f(y_1, \ldots, y_n) \mid$	*Unification*
	$p(x_1, \ldots, x_n) \mid$	*Call*
	$(G_1, \ldots, G_n) \mid$	*Sequential Conjunction*
	$(G_1 \& \ldots \& G_n) \mid$	*Parallel Conjunction*
	$(G_1 ; \ldots ; G_n) \mid$	*Disjunction*
	$(V = v_1, G_1 ; \ldots ; V = v_n, G_n) \mid$	*Switch*
	$(if\ G_c\ then\ G_t\ else\ G_e) \mid$	*If then else*
	$not\ G \mid$	*Negation*
	$some\ [x_1, \ldots, x_n]\ G$	*Existential quantification*

Figure 1: Internal representation of Mercury goals

Mercury has a strong Hindley-Milner type system very similar to Haskell's. Mercury programs are statically typed; the compiler knows the type of every argument of every predicate (from declarations or inference) and every local variable (from inference). The mode system classifies each argument of each predicate as either input or output. The caller must ensure that input arguments are ground terms before the call is made. Output arguments must be distinct free variables before the call; the callee will instantiate them to ground terms. It is possible for a predicate to have more than one mode; the usual example is `append`, which has two principal modes: `append(in,in,out)` and `append(out,out,in)`. We call each mode of a predicate a *procedure*. The Mercury compiler generates different code for different procedures, even if they represent different modes of the same predicate.

The compiler's mode checker is responsible for reordering conjuncts as necessary to ensure that for each variable, the goal that generates the value of the variable comes before all goals that use this value. For each variable in each procedure, the compiler knows exactly which atomic goal in that procedure makes that variable ground.

Logic programs often make use of *backtracking*. This is used when an evaluation of a program hits a dead end and there is an unexplored alternative earlier in the program due to a disjunction. In such a case the execution *backtracks* to the most recent point in the program with an unexplored disjunct. Execution then resumes from the first such disjunct as if the failed computation had no effect.

Byrd's box model [2] for logic programs describes this in terms of four ports: the call, exit, redo and fail ports. When a procedure is invoked, execution enters the invocation via the call port. If it succeeds, execution leaves it via the exit port. If it has an unexplored disjunct, execution may reenter it via the redo port. If it cannot produce a solution, execution leaves via its failure port. Mercury adds a fifth port: when a procedure throws an exception, execution leaves the invocation through the excp port.

Execution must leave a goal exactly the same number of times as it enters. This relationship is shown in Equation 1. In this paper we refer to the number of times execution passes through a port as a *port count*.

$$Calls + Redos = Exits + Fails + Excps \tag{1}$$

Determinism	Maximum solutions	Minimum solutions
det	1	1
semidet	1	no
multi	∞	1
nondet	∞	no
erroneous	0	1
failure	0	no

Table 1: Mercury's determinisms

Each procedure has an associated determinism, which puts limits on the number of its possible solutions. Mercury's determinisms are given in Table 1. Procedures with a determinism of *det* succeed exactly once for each call; *semidet* procedures succeed at most once for each call; *multi* procedures succeed at least once for each call; while *nondet* procedures may succeed any number of times. Two other determinisms exist: *failure* and *erroneous*, which are used for goals that always fail and always throw exceptions respectively.

A procedure with determinism det is never backtracked into and it never fails, so its redo and fail port counts must both be zero. Similarly the redo port counts of semidet procedures must be zero. Multi and nondet procedures may have any non-negative numbers for any of their port counts, though for multi procedures, the sum of their exit and excp port counts cannot be less than their call port count.

2.2 Parallelism in Mercury

Modern operating systems provide two major multi-processing facilities, multiple *processes* and multiple *threads*. Typically each running program is a process that runs independently of other processes on the system. Multiple processes can be used to implement parallelism within programs. Processes do not implicitly share memory, therefore communication between processes is done via sockets, explicit shared memory, signals or a combination of these and other mechanisms.

It is the operating system's responsibility to schedule the execution of processes and threads, including assigning them to processors. The operating system may also move a thread or process to a different processor if the one it was using becomes unavailable. Modern operating systems use preemptive multitasking, meaning they may interrupt the execution of a program in order to run another program. This ensures that all programs on the system are able to respond quickly to the user and to each other.

A single process can have multiple threads. Threads can also be thought of as light-weight processes since like processes they have their own copy of CPU registers and stack. Other resources such as the heap, code and static data are shared between threads within that process. The POSIX standard set defines a threading library known as POSIX threads or pthreads [1] which is supported by many popular operating systems. Parallelism in Mercury is implemented using pthreads.

Mercury supports a number of execution algorithms each with their own backend code generator. The oldest and best supported of these is the low-level

C backend, which is implemented in terms of an abstract machine defined by C preprocessor macros. Note that Mercury programs are not interpreted, as the macros are expanded into a C program that is compiled by a C compiler. Like a regular processor, the abstract machine has a number of general purpose registers as well as stack registers for pointing to the tops of two stacks: the *det-stack*, used for procedures with at most one solution, and the *nondet-stack*, used for procedures that can have more than one solution. Some registers are implemented by using physical machine registers; their number depends on the architecture of the physical machine.

When a Mercury program is started, the Mercury runtime system will create a number of *engine* structures, each of which represents a virtual processor in the abstract machine with its own set of abstract machine registers. Each engine therefore runs in its own thread. The number of engines created depends on the value of an environment variable. This number should typically be set equal to the number of CPUs in the machine so that each engine can run on a separate processor, though the Mercury runtime can not control which engine runs on which processor.

Just as Mercury engines are abstractions of CPUs, Mercury contexts are abstractions of processes — computations running on CPUs. Each Mercury context represents a computation that can be executed in parallel with other computations. A context stores the state of a computation when its not running. This state consists of the values of the abstract machine registers and the contents of the det-stack and the nondet stack.

Creating new contexts is expensive, due to the expense of creating the stacks. However, when a spawned computation has not started executing yet, its stacks are empty. To allow finer granularity parallelism, Wang [18] introduced the *spark* structure, which represents a description of a call to be executed. A spark contains only three words of data: a pointer to the label at which it should begin execution, a copy of its parent's det-stack pointer register and a pointer to the next spark in a list. Sparks do not need a copy of the nondet stack pointer register since we do not support parallel execution of nondeterministic code, and they do not need copies of the values in general purpose registers because these are all pushed onto the det stack before the spark is created. Sparks are thus extremely light-weight compared to contexts. An engine's run queue may contain sparks as well as contexts. When a spark is scheduled to run, the computation it represents will be allocated a context, but in the meantime the program will need less memory. Whenever we can, we also reuse the memory used by contexts whose computations have completed.

Mercury implements for parallelism for through parallel conjunction goals. To avoid the need for coordinated backtracking between different engines, each parallel conjunct must be a deterministic computation. For a parallel conjunction of N conjuncts, the compiler will generate sparks for $N-1$ of them and leave one to execute in the current context. The last operation in each of the spawned-off conjuncts is an abstract machine instruction that registers the completion of this conjunct, and then ends the execution of the context by making the memory of the context available for reuse, and then invoking the equivalent of an OS scheduler to tell the engine on which the context used to run to switch to executing another context or spark.

A conjunct in a parallel conjunction may be dependant on one or more variables that a conjunct to its left produces; if so the conjunction is said to be

Figure 2: Independent and dependant parallel conjunctions

dependant. Figure 2b shows an example of this. In this example `a` and `b`'s first arguments are input and their second arguments are output. `Y` is the dependant variable, the predicate `a` is the producer while `b` is the consumer. Figure 2a shows an independent conjunction, there are no dependant variables between the two goals here.

If a consuming conjunct is running in parallel with a producing conjunct, it cannot proceed past the point at which it requires a variable binding until the producing conjunct has produced that binding. Mercury implements this by using *futures*, each of which represents a value that will be available in the future. A future is a wrapper around a normal variable, added by the Mercury compiler during code generation. A future contains: a word that will be set to the contents of the variable (this is a pointer for data larger than a memory cell), a boolean describing whether the variable has been produced yet, a pthreads mutex (mutual-exclusion lock) used to protect the future from concurrent access, and a list of contexts that are waiting on the production of the variable [19].

Two possible things can happen with regard to the future during execution. The consuming conjunct may attempt to acquire the value in the future before it is produced. If this happens, the consuming conjunct's context will add itself to the list of contexts waiting for the variable to be produced before suspending. When the producing conjunct generates the value of the variable, it will *signal* all the waiting contexts, allowing them to resume. Alternatively the consuming conjunct may attempt to acquire the value of the future after it has been produced. In this case the consuming conjunct dose not need to sleep and can immediately retrieve the value of the variable from the future.

We show in later sections how the time at which the dependant variables are likely to be produced and consumed can be used to calculate the amount of parallelism available in dependant conjunctions.

2.3 Deep Profiling

The Mercury deep profiler is a profiler engineered specifically for declarative programming languages [6]. Declarative languages have unique profiling requirements that conventional profilers do not handle.

Conventional profilers collect information on a per-procedure basis. They usually modify the program in two ways: they add instrumentation to the beginning of every procedure to count the calls made to that procedure, and they sample the program counter periodically to determine how much execution time is being spent in each procedure. However, the information one can collect this way is not sufficient for declarative programming languages, for several reasons.

One reason is that declarative programs contain more procedure and type polymorphism than imperative programs. For example, the Mercury standard

library defines a polymorphic type representing maps from keys to values. Different maps may have keys and values of different types, allowing the same code to work on many different kinds of maps. These different maps may have very different performance characteristics. Traditional profilers would collapse them all into a single set of measurements, as they associate profiling information only with individual procedures. In traditional languages, this is not a problem, as each kind of map would have its own separate code and thus its own separate profiling information.

A second reason is that declarative programs make heavy use of recursion, including mutual-recursion. Most profilers do not attribute costs correctly in the presence of recursion, because recursion necessarily breaks one of their core assumptions; if one call to `f` makes a recursive call to `f`, the assumption that all calls to `f` have the same cost cannot possibly be right.

A third reason is the much higher incidence of higher-order programming in declarative programs. For example most (if not all) declarative languages define a `filter` function to filter a list of values based on the result of a higher-order function. In some programs, the higher-order functions may be computationally expensive, causing the use of `filter` to be expensive. In other cases this function may be cheap. Declarative programmers want their profiling tools to separate out the profiling data for `filter` based on the callee of the higher-order call.

Because these aspects are common in declarative programs, it is important that the profiler collects accurate information for code that uses them. This provided the motivation for developing Mercury's deep profiler.

Deep profiling associates profiling statistics not only with a procedure, but also with its call site and its deep context. A call site's deep context is represented by an alternating chain of call sites and procedures that lead to its invocation. Call sites must be included in this chain because a procedure may make several calls to the same callee procedure, and those different call sites may have different performance characteristics.

The implementation of this alternating chain uses two types of structures: `CallSiteDynamic`, and `ProcDynamic`. Any number of these structures may exist for every call site or procedure in the program. Profiling data is stored in `CallSiteDynamic` structures. Each `CallSiteDynamic` structure contains a pointer to a `ProcDynamic` structure representing its callee. This is true even for higher order calls, since the callee is known at runtime, though since different calls through such call sites may call different procedures, such call sites may point to more than one `ProcDynamic`. Similarly each `ProcDynamic` structure contains an array of pointers to `CallSiteDynamic` structures that represent the call sites within that procedure.

The profiling data must refer to procedures by name, and call sites by the call type (normal first order call, higher order call, or method call). Procedures and call sites are both also identified to users by file name and line number within the source program. Rather than potentially storing this information multiple times in each `ProcDynamic` and `CallSiteDynamic` structure, we use two other structures, `ProcStatic` and `CallSiteStatic`, to store this information. Every `ProcDynamic` structure has a pointer to the `ProcStatic` structure that stores the shared (and mostly static) information about that procedure. Similarly each `CallSiteDynamic` structure has a pointer to the `CallSiteStatic` structure that stores shared static information. This saves a large amount of memory during

profiling, since there is exactly one `ProcStatic` structure for each procedure and one `CallSiteStatic` structure for each call site.

The `ProcStatic` and `CallSiteStatic` structures are created statically at compile time. The `ProcDynamic` and `CallSiteDynamic` structures are created at runtime as needed by the instrumentation code as execution passes from one procedure to the next.

Programmers are often interested in how much execution time is spent in each procedure and its descendants. Measuring this accurately on a modern computer requires interfacing with a high resolution clock, which cannot be done portably. However, because all loops in Mercury programs are created by recursive or mutually-recursive calls, the number of calls made by a procedure and its descendants is an approximation of time spent in that procedure and its descendants. Call counts have a much higher resolution than the portable clock ticks used by Mercury's deep profiler. Our implicit parallelism implementation uses call counts to approximate the execution time of a procedure or call.

Any program worth profiling (because it takes a non-trivial amount of time to execute) must contain recursion. If we always created a new `CallSiteDynamic` structure and/or a new `ProcDynamic` structure for each call, the profiler's data structures would consume several gigabytes of memory per second, and the machine would run out of memory very quickly. What we actually do is more much more space efficient: When a call is made to a procedure but an activation of that procedure is still active, we reuse the `ProcDynamic` structure of that activation, and therefore reuse its array of `CallSiteDynamic` structures as well. This means that profiling information for all the recursive invocations of a procedure get charged to the first invocation.

When the program finishes executing, the profiling data is written out to a file named `Deep.data`. This file is designed to be processed by the Mercury deep profiling tool. This tool takes the raw profiling data, which has raw profiling data in its `CallSiteDynamic` structures, and processes that data into more useful forms. The first step in this processing is discovering the cliques, the recursive and mutually recursive procedure invocations in the call graph. Replacing each clique in the call graph with a single node will by definition yield a graph with no cycles, which allows us to propagate profiling information from callees to callers. This allows the deep profiler to tell the user lots of information: the cost of a clique (by itself or including its descendants), the cost of a procedure (by itself or including its descendants), the list of the N most expensive procedures in their program, and so on. The information we are most interested in is the average costs of each call site, since we need this information to determine whether a conjunction containing two calls is worth parallelising or not.

3 Implicit Parallelisation

One lesson of research on parallel logic programming in the 1980s is that parallelising everything is a bad idea because of the overheads; there is no point in parallelising a conjunction unless parallelisation is likely to yield a speedup. The likely parallelisation speedup is the likely sequential execution time minus the likely parallel execution time (See Equation 2). We use the terms *likely* speedup and *likely* execution time because the execution time of a computation cannot be known before it has been executed.

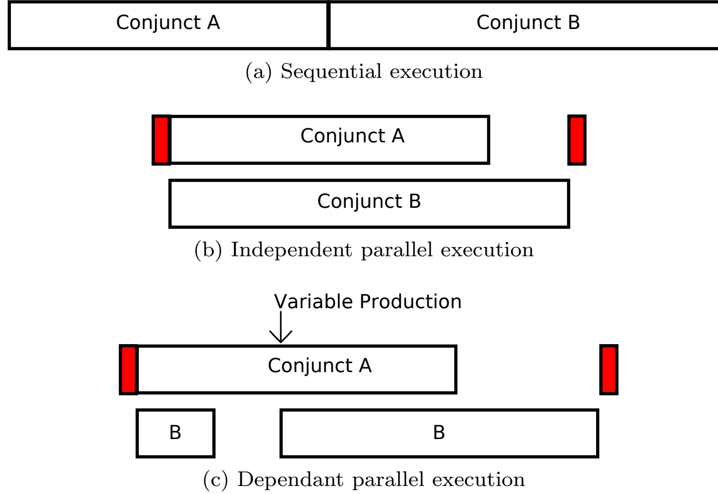

Figure 3: Sequential versus parallel executions of two conjuncts A and B

$$Speedup = T_{Sequential} - T_{Parallel} \tag{2}$$

Therefore to calculate the likely speedup, we must know the likely sequential and parallel execution times.

The likely execution time of a conjunction, either sequential or parallel, depends upon the execution times of the conjuncts within the conjunction. The likely execution time of other goals depends on their type: unifications are trivial to execute and are assumed to have an execution time of zero. The likely execution time of calls is determined by profiler feedback. The execution time of non-atomic goals depends upon the execution times of their components.

For our examples in this section, we consider a conjunction of two conjuncts, A and B, where B takes longer to execute than A. The alternative executions of these conjuncts are shown in Figure 3. Later we show how this can be extended to work for conjunctions of any size.

Given access to the kind of information collected by the deep profiler, the likely sequential execution time of these two conjuncts is simple to calculate: it is the sum of their individual execution times.

$$T_{Sequential} = T_A + T_B \tag{3}$$

Provided that there is a spare CPU available, the likely parallel execution time of the two conjuncts, if they are independent of each other, is the likely execution time of the longer running conjunct, plus the overheads of parallel execution, as shown below in Equation 4.

$$T_{Parallel} = max(T_A, T_B) + T_{Overheads} \tag{4}$$

(If there is no spare CPU available, then the time taken will be $T_{FailedParallel} = T_A + T_B + T_{Overheads}$, which is why having too much parallelism is bad.) Figure 3b shows the case where there is a spare CPU; the shaded bars in the figure represent the overheads of parallelism.

The largest sources of these overheads are the creation of a context (if we cannot reuse one), the creation and later destruction of a spark, and the invocations of the scheduler. We cannot know at compile time whether we will need a new context, nor can we determine the costs of these operations at all accurately. We must measure them, and these measurements will yield a range of costs. We prefer to make our estimate the maximum observed overhead rather than the average observed overhead, because it reduces the risk of parallelising conjunctions that lead to slowdowns.

The execution time of an dependant parallel conjunction of two conjuncts A and B that have one dependant variable is given as $T_{Parallel}$ below in Equation 5. This equation divides $T_{DependantB}$, the time that B takes to execute, into two components: the time before and the time after B first consumes the dependant variable. The second component cannot begin until A has produced the variable. Figure 3c shows the case where B must wait for A to produce a variable.

$$\begin{aligned} T_{DependantB} &= max(T_{BeforeProduceA}, T_{BeforeConsumeB}) + T_{AfterConsumeB} \\ T_{Parallel} &= max(T_A, T_{DependantB}) + T_{Overheads} \end{aligned} \quad (5)$$

Any of the execution time calculations can be performed for a conjunction with any number of conjuncts. The sequential execution time of a conjunction of any size is simply the sum of the execution times of all the conjuncts, see Equation 6 below.

$$T_C = \sum_{c \in C} T_c \quad (6)$$

A parallel conjunction may have dependencies between any of its conjuncts as long as the conjunction is well moded — the producer of a variable is on the left of all the consumers of that variable. When executing a parallel conjunction, sparks representing the next conjunct are created by each conjunct except the last. The parallel conjunction symbol can be thought of as a right-associative binary operator that sparks its second argument before executing its first. Therefore, the parallel execution time of a conjunction of any size can be expressed as the parallel execution time of a conjunction of two goals, where the second goal is the parallel conjunction of the all the goals except for the first. This rule is recursive, it should be applied to calculate the execution time of the conjunction of all the goals except the first. Our implementation does not attempt to parallelise more than two conjuncts against each other at this time.

Calculating the dependant parallel execution time of conjunctions requires an analysis that determines when dependant variables are likely to be produced by the first conjunct and consumed by the second. For some goal types this is easy, for example consider a sequential conjunction of three goals, where the first and last goals are calls and the middle goal is a unification that produces the variable that we are interested in. The execution time of the entire conjunction is the sum of the times of all the goals. The time before the variable is produced is the execution time of the first goal (which the deep profiler can give us, for the profiled run of the program) and of the second goal (which we approximate as zero, because unifications require only a few instructions).

For other goal types, this is harder. Consider a switch that produces the variable in each switch case. The execution time before the variable is produced is

the average of the time before the variable is produced in each case, as weighted by the probability of execution of each case. The execution time of the whole switch is also a weighted average of the execution times of the different cases. The only sound way to calculate the weights is to measure the frequency of execution of the different arms for disjunctions, switches, and if-then-else goals. The existing deep profiler did not provide this information, so we modified it to give it to us.

4 Coverage Profiling

The deep profiler already provides some coverage information in the form of port counts for each procedure and each call site within a procedure. However, for finding the most profitable places to introduce parallelism, more coverage information is required. As described in the previous section, we need information about the frequency of execution of branches in if-then-elses, switches and disjunctions, even if they do not include calls. We have therefore modified the deep profiler to support true coverage profiling.

True coverage profiling requires counting the number of times execution reaches particular points in the program, points that we call *coverage points*. One of our objectives was to minimise the number of coverage points, since this reduces the performance impact of coverage profiling and hence reduces distortions in the execution profile caused by executing the profiling instrumentation code.

The reason we do not need to collect coverage information at every point in the program is that execution counts at some points in a procedure can be used to infer the execution counts at some other points within that procedure. Later in this section we describe the coverage inference algorithm we use to infer the coverage information throughout a procedure.

The program is instrumented by the coverage profiling transformation within the compiler, which is executed immediately after the deep profiling transformation. For each point within the program where the execution count cannot be inferred from other counts, this transformation arranges for it to be measured at that point. A position within a execution count array for that procedure is allocated to store the execution count and the coverage point instrumentation is inserted at that point in the program. The instrumentation increments the counter in the array each time it is executed. The execution count array is stored in the procedure's `ProcStatic` structure. When the program exits, the values of all the counters are written out with the rest of the profiling data.

In order to interpret the coverage profiling data, a bytecode representation of the program is also written to a file. It is important that this bytecode representation matches the version of the program that the compiler will parallelise later. This can be ensured provided that the program has not been modified by the programmer in the meantime. In particular the relevant optimisations will have been performed but the program's structure will not be altered by the profiling transformations.

Each coverage point has associated static data that describes which goal in the bytecode representation this coverage point refers to, and whether the counter measures the count before or after that goal. Most coverage points measure the execution count after the goal they refer to. Others measure how

```
procedure infer_coverage(Goal, CountBefore):
  switch on the type of Goal:
    case unification:
      if Goal's determinism is det:
        CountAfter := CountBefore
      else if Goal's determinism is failure or erroneous:
        CountAfter := 0
      else
        CountAfter := look up the coverage point after Goal
      end if

    case call:
      assert call count from call site == CountBefore
      CountAfter := get exit count from call site

    case conjunction:
      CurrentCount := CountBefore
      for Conj ∈ Goal's conjuncts:
        CurrentCount := infer_coverage(Conj, CurrentCount)
      end loop
      CountAfter := CurrentCount
```

Figure 4: Coverage inference algorithm — Part 1

many times execution enters a branch, namely a switch case, disjunct or the then and else parts of an if-then-else goal.

The coverage inference algorithm (shown in Figures 4, 5 and 6) traverses the procedure depth-first, annotating goals with coverage information. This coverage information consists of the execution count both before the goal and after the goal. The large switch performs coverage inference for each individual goal type. The count before a goal is often used to infer the coverage counts of any inner goals, then the execution counts after those inner goals may be used to infer the execution count after the larger goal.

The coverage inference algorithm expects to find coverage points within the program at various places, including the beginning of the else branch in an if-then-else, after a unification that may fail, and after a negation. The decision to insert a coverage point at any point in a program depends only on information that is known at compile-time. The compiler's coverage profiling transformation therefore has the same structure as the coverage inference algorithm, except that wherever the coverage inference algorithm reads a coverage point, the compiler inserts a coverage point.

The Mercury compiler's mode system knows which goals might fail and sets the determinism of the goals appropriately. We use this to avoid putting coverage points after unifications that always succeed and after unifications that never succeed.

As we mentioned above, the execution counts before and after each call can be determined by the call site's port counts. Our coverage inference algorithm uses the exit port count as the execution count after the call. To help catch bugs in the coverage profiling and deep profiling systems, we use an assertion

```
case disjunction:
  SumCountAfter := 0
  for Disj ∈ Goal's disjuncts:
    if Disj is the first disjunct:
      CountBeforeDisj := CountBefore
    else:
      CountBeforeDisj := look up the coverage point before Disj
    end if
    CountAfterDisj := infer_coverage(Disj, CountBeforeDisj)
    SumCountAfter := SumCountAfter + CountAfterDisj
  end loop
  CountAfter := SumCountAfter

case switch:
  SumCountBefore := 0
  SumCountAfter := 0
  for Case ∈ Goal's cases:
    if Case is the last case and the switch is complete:
      CountBeforeCase := CountBefore - SumCountBefore
    else:
      CountBeforeCase := look up the coverage before Case
    end if
    CountAfterCase := infer_coverage(Case, CountBeforeCase)
    SumCountBefore := SumCountBefore + CountBeforeCase
    SumCountAfter := SumCountAfter + CountAfterCase
  end loop
  CountAfter := SumCountAfter
```

Figure 5: Coverage inference algorithm — Part 2

to check the invariant that the execution count inferred before the call is equal to the call port count of the call site.

Three rules describe coverage inference for a conjunction: the execution count before the first conjunct is equal to the count before the whole conjunction. The count before the later conjuncts is equal to the count after the previous conjunct. The count after the whole conjunction is equal to the count after the last conjunct.

Two rules describe coverage inference for disjunctions: the execution count before the first disjunct is equal to the count before the whole disjunction, and the count after the whole disjunction is the sum of the execution counts after each disjunct. We insert a coverage point at the beginning of every disjunct except the first, since in general we don't have any other way of knowing how many times those program points are reached; if the disjunction has no outputs (which makes the disjunction semidet), then execution will exit the disjunction as soon as one disjunct has succeeded.

In general, we have to place a coverage point at the start of every arm of the switch. There is only one exception. If the switch is complete — that is it has a case for every possible value of the switched-on variable — then the sum of the execution counts at the starts of the all arms must equal the execution

```
    case if_then_else:
      let Cond, Then and Else be the three components of the if-then-else
      CountAfterCond := infer_coverage(Cond, CountBefore)
      CountBeforeThen := CountAfterCond
      CountBeforeElse := look up the coverage before Else
      CountAfterThen := infer_coverage(Then, CountBeforeThen)
      CountAfterElse := infer_coverage(Else, CountBeforeElse)
      CountAfter := CountAfterThen + CountAfterElse

    case negation:
      let SubGoal be the goal within the negated context
      CountAfterIgnored = infer_coverage(SubGoal, CountBefore)
      CountAfter := look up the coverage after Goal

    case existential_scope:
      let SubGoal be the goal within the existential scope
      CountAfter := infer_coverage(SubGoal, CountBefore)
  end switch

  annotate Goal with <CountBefore, CountAfter>
  return CountAfter
end procedure
```

Figure 6: Coverage inference algorithm — Part 3

count before the switch itself. We can therefore omit the coverage point before one of the cases, and compute the execution count there from all other counts we have. Like disjunctions, the count after the switch is equal to the sum of the counts after the switch cases.

Three coverage inference rules apply to if-then-else (ITE) goals: the execution count before the condition goal is the same as the count before the ITE goal. The count before the then goal is equal to the count after the condition goal. The execution count before the else goal in an ITE is equal to the failure count of the condition, which can be calculated if the other port counts are known. Our implementation does not measure the number of exceptions thrown by a goal, therefore this is not implemented. Instead the execution count before an else goal determined by a coverage point inserted there. The count after the whole ITE is the sum of the count after the then and else goals.

As an alternative, the compiler provides an analysis that determines if a goal might throw an exception. It would be possible to use this analysis to remove the coverage point before the else goal if we know that the condition cannot throw an exception. This has not been implemented.

The execution count before a negated goal is equal to the count before the negation as a whole. The count after the negation is equal to the failure count of the inner goal. As described above, the failure count of a goal cannot be calculated without exception information, so the execution count after a negation cannot be inferred. We therefore put coverage points after all negations.

The execution count before a goal in an existential quantification scope is equal to the count before the whole goal. The subgoal of an existential quan-

Grade	Coverage profiling	Optimisations	Runtime (seconds)
asm_fast.gc	N/A	some	12.82
asm_fast.profdeep.gc	no	none	66.39
asm_fast.profdeep.gc	yes	none	70.17
asm_fast.profdeep.gc	yes	some	46.87

Table 2: Average compilation times for coverage profiling

tification may have more than one solution, if so the these solutions are never used. In other cases the subgoal will have the same number of solutions as the whole scope. In either case the execution count of the sub goal and the existential scope will be equal. Therefore the execution count after the existential quantification can be inferred as the count after its inner goal.

Table 2 compares the execution times of several versions of the Mercury compiler when given the task of compiling six of large modules. The first version is the usual version. The second has deep profiling enabled, while the third has both deep profiling and coverage profiling enabled. Both forms of profiling will by default disable all optimizations that could distort in the profile, but the fourth version reenables some of these; when coverage profiling is used for implicit parallelism analysis, optimisations should be enabled to minimize the differences between the program being profiled and the program we will want to parallelize. Each time shown is derived from the times of six runs, discarding the lowest and highest execution times, and averaging the rest. The test was performed on an Intel Core2Quad machine with four 2.4Ghz processing cores and 4GB of memory It runs Ubuntu, using version 2.6.24-19 of the Linux kernel and version 3.4.6 of GCC. These benchmarks show that while the deep profiler degrades the performance of a program, the overheads of coverage profiling do not make this significantly worse. Therefore coverage profiling is unlikely to distort the profile of the program much. While it is important that a profiler does not distort the profile of a program, all the measurements used for our implicit parallelisation analysis are collected synchronously and do not measure time, the execution of instrumentation will not affect these measurements.

5 Finding good candidates for parallelism

5.1 Variable Use Time Analysis

We can use coverage information to determine when variables are first consumed or produced in a goal.

The variable consumption time analysis algorithm is given in pseudo code as the procedure `goal_var_first_consume` in Figures 7, 8 and 9. The first argument of this procedure is the goal that should be searched for the first consumer of the variable in the second argument. The remaining two arguments are the cost from the beginning of the procedure containing the current goal up to the start of the current goal (the first argument) and a stack representing procedure calls that are already being analyzed. The return value is a tuple of three values:

```
procedure goal_var_first_consume(Goal, Var, CostBefore, Stack):
  switch on Goal's type:
    case unification:
      if Goal uses Var:
        CostBeforeConsume := CostBefore
        Found := yes
      else:
        Found := no
      CostAfter := CostBefore

    case call:
      if Call is higher-order and Var is the higher order value:
        CostBeforeConsume := CostBefore
        Found := yes
      else if Goal uses Var:
        if Goal is higher-order or a method call or Goal ∈ Stack:
          CostBeforeConsume := CostBefore
          Found := yes
        else:
          let Callee be the procedure called by Goal
          NewStack := push(Stack, Callee)
          (Found, CostBeforeProcConsume) :=
            proc_var_first_consume(Callee, Var, NewStack)
          CostBeforeConsume := CostBefore + CostBeforeProcConsume
      else:
        Found := no
      CostAfter := CostBefore + cost of this call site

    case conjunction:
      CostAfterConj := CostBefore
      Found := no
      for Conjunct ∈ Goal while Found = no:
        (Found, CostAfterConj, CostBeforeConsume) :=
          goal_var_first_consume(Conjunct, Var, CostAfterConj, Stack)
      CostAfter := Cost
```

Figure 7: Variable consumption time analysis — Part 1

- A found flag: a boolean that is true if a consumer of the variable has been found.
- The cost from the beginning of the procedure containing the goal until immediately after the goal in the first argument. This is valid if the found flag is false.
- The cost from the beginning of the procedure containing the goal until the variable is consumed. This is valid if the found flag is true.

The time in the procedure after the variable has been consumed can be calculated as the total execution time of the procedure as given by the profiler minus the time within the procedure before the variable is consumed.

The procedure traverses the goal left to right, counting the likely execution

```
case disjunction:
  CostAfterDisj := CostBefore
  Found := no
  for Disjunct ∈ Goal while Found = no:
    (Found, CostAfterDisj, CostBeforeConsume) :=
      goal_var_first_consume(Conjunct, Var, CostAfterDisj, Stack)
  CostAfter := CostAfterDisj

case switch:
  initialise CaseCostsAfter, CaseCostsBeforeConsume and Weights as
    empty lists
  Found := no
  for Case ∈ Goal:
    Weights := Weights ++ get_exec_count_before(Case)
    (CaseFound, CaseCostAfter, CaseCostBeforeConsume) :=
      goal_var_first_consume(Case, Var, CostBefore, Stack)
    if CaseFound:
      Found := yes
    else:
      CaseCostBeforeConsume := CaseCostAfter
    CaseCostsAfter := CaseCostsAfter ++ CaseCostAfter
    CaseCostsBeforeConsume := CaseCostsBeforeConsume ++
      CaseCostBeforeConsume
  CostAfter := weighted_average(CaseCostsAfter, Weights)
  if Var is the switched-on variable:
    CostBeforeConsume := CostBefore
    Found := yes
  else:
    CostBeforeConsume :=
      weighted_average(CaseCostsBeforeConsume, Weights)
```

Figure 8: Variable consumption time analysis — Part 2

time (cost) before the variable named by the second argument is first consumed.

Unification goals are trivial and assumed to have a cost of zero, therefore the cost after unifications is the same as the cost before them. If the unification consumed the variable, the cost before the variable is consumed is the cost before the unification.

Call goals can either be higher order calls, method calls or plain calls. Higher order calls call a higher order value. If that value is the value of the variable whose consumers we are searching for, then this goal consumes that variable. The cost before such a consumption is the cost before the higher order call. If the variable appears in the argument list of the call, then this goal consumes that variable, so we attempt to follow the call to determine how soon the variable is used in the called procedure. We know the identity of the callee only if this is a plain call; in other cases we make the pessimistic assumption that the variable is used at the beginning of the called procedure. The cost before the variable is used within the called procedure is added to the cost before the call and returned as the cost before the variable is consumed in this procedure. The cost immediately after a call is the cost before the call plus the cost of this

```
    case if-then-else:
      let Cond, Then and Else represent the components of the ITE.
      (FoundCond, CostAfterCond, CostBeforeConsumeCond) :=
        goal_var_first_consume(Cond, Var, CostBefore, Stack)
      (FoundThen, CostAfterThen, CostBeforeConsumeThen) :=
        goal_var_first_consume(Then, Var, CostAfterCond, Stack)
      (FoundElse, CostAfterElse, CostBeforeConsumeElse) :=
        goal_var_first_consume(Else, Var, CostAfterCond, Stack)
      ThenWeight := get_exec_count_before(Then)
      ElseWeight := get_exec_count_before(Else)
      CostAfter := weighted_average([CostAfterThen, CostAfterElse],
        [ThenWeight, ElseWeight])
      if FoundCond:
        Found := yes
        CostBeforeConsume := CostBeforeConsumeCond
      else if FoundThen or FoundElse:
        Found := yes
        if the consumer was not found one of the then or else
          branches, then set the cost before the variable was consumed
          in that branch to the cost after that branch.
        CostBeforeConsume := weighted_average(
          [CostBeforeConsumeThen, CostBeforeConsumeElse],
          [ThenWeight, ElseWeight])
      else:
        Found := no

    case negation or existential scope:
      let SubGoal be the goal within Goal
      (Found, CostAfter, CostBeforeConsume) =
        goal_var_first_consume(SubGoal, Var, CostBefore, Stack)

  end switch
  return (Found, CostAfter, CostBeforeConsume)
end procedure
```

Figure 9: Variable consumption time analysis — Part 3

call site as given by the deep profiler (but see below). Note that we must not follow all recursive or mutually recursive calls, as these would cause unbounded recursion within our analysis. To prevent this, we maintain a stack representing the procedures whose bodies we have followed so far, and we check to see that the callee we are about to follow is not already in the stack. Before following a call we push the callee's identity onto the stack.

Conjunctions are analysed from left to right so that the cost before each conjunct is set correctly. If a conjunct is the first use of a variable then there is no need to check the rest of the conjunction.

Disjunctions come in two types. Those that do not bind variables and therefore have at most one solution are the most common type. Disjunctions that bind variables and may thus have more than one solution are rarer. Our implicit parallelisation analysis does not handle the second type of disjunction.

This is because parallelism of non-deterministic code is not supported and non-deterministic code can not be reached if that code is called from Mercury's built-in `solutions/2` predicate. The `solutions/2` predicate makes a higher-order call to execute the non-deterministic code, and higher order calls are not followed by the variable consumption time analysis. Non-deterministic code can only be reached from a deterministic context by our variable consumption time analysis via an existential scope. Code within an existential scope may have more than one solution, however only the first solution will be used. Retrying code by entering unexplored disjuncts is much less common. Therefore the variable consumption time analysis does not need to handle disjunctions with more than one solution.

For simplicity, we assume that disjunctions run from beginning to end like conjunctions. This assumption is conservative for disjunctions that produce at most one solution because they cannot be re-entered if a goal to the right of the disjunction fails. To handle such disjunctions non-conservatively, we would need to calculate the probability of failure of each goal in each disjunct, so that we knew at what point a disjunct would fail and execution would proceed to the next disjunct. Disjunctions with more than one solution can be re-entered if a goal after the disjunction fails, so for them we would also need to know the probabilities of failures at points after the disjunction in order to calculate when disjuncts other than the first are entered.

The analysis iterates through the disjuncts until it finds the first time the variable is consumed. We compute the cost to the point just after the disjunction as the cost before the disjunction plus the sum of the costs of all disjuncts. This is inaccurate, but it is a conservative approximation, so it should not lead to over-parallelising the program.

If a switch switches on the variable whose first consumer the analysis is looking for, then the cost before the first consumption is the cost before the switch. Otherwise we check each case of the switch to see if it consumes this variable. If at least one case consumes the variable, then we take the average of when all the cases consume the variable, weighted by the number of times each case is executed as given by coverage profiling data. It is possible that some switch cases may consume the variable and some will not. Our analysis sets the variable's consumption time to the time at the end of the switch case for all the switch cases that do not consume the variable. It does this to model the action of the Mercury compiler, which will insert a call to wait for the variable at the end of such cases, in order to respect the invariant that all branches of a switch must bind the same set of nonlocal variable, and to allow goals after the switch to use the value without waiting for it. The cost after the switch is the weighted average of the costs after each of the cases.

If the variable whose first consumer the analysis is looking for is consumed in the condition of an if-then-else goal (ITE), the compiler will wait for the variable before the condition, so we return the time so far as the time before consumption. (If a part of the condition could fail before the first mention of the variable, then the compiler could do better, but it doesn't try, and our analysis must model the action of the compiler.) If the variable is consumed in the then and else branches, the analysis computes the average of the costs before the variable is consumed in each branch weighted by the execution count of each branch as given by coverage profiling. The variable may be consumed in one branch but not the other. As with switches, the compiler will ensure that the

program waits for the future to be produced in both branches by adding code to wait for the future at the end of the branch that does not contain a consumer. Therefore the analysis assumes that the variable is consumed at the end of the branch that does not contain a consumer. The cost after the entire ITE is the average of the cost after the then and else branches weighted by the execution count of the branches.

The case for negations and existential quantification scopes are very simple, they simply call `goal_var_first_consume` for the sub goal of the negation or existential scope and return its output.

Unfortunately retrieving the cost for a recursive or mutually recursive call site in a deep profile is not easy. The cost of a recursive call depends upon the current depth of the recursion, and this information is not directly collected by the deep profiler so that it does not run out of memory when profiling long running programs. Therefore the this analysis cannot retrieve the true cost of many call sites.

A feasible solution to this problem may be to calculate the cost of the base case of any recursive procedure (which is the cost when the recursion depth of a call to the procedure is zero), and to calculate the cost of calls with other recursion depths from this:

$$\begin{aligned} cost(0) &= cost_base \\ cost(d) &= cost_recursive(cost(d-1)) \mid d > 0 \end{aligned} \tag{7}$$

The obvious way to implement this would traverse the procedure body n times to calculate $cost(n)$, but one could also try to explicitly construct this recurrence and try to express it in a closed form. We have not yet implemented either solution.

We have deliberately chosen to search for a consumer starting at the beginning of a procedure and moving towards the end so that the analysis does not need to know the cost of calls that the variable is consumed in. When the analysis reaches a call that consumes the variable it uses and it cannot recurse into the call because it is a recursive, mutually recursive, higher order or a method call, it assumes the conservative default: that the call consumes the variable immediately. If this call is recursive or mutually recursive, then the analysis cannot retrieve the correct cost of the call from the deep profiler. However the cost of the call is not required because the cost before the variable is first consumed is made equal to the cost before the call as per this conservative assumption.

A similar analysis is used to determine when a variable is produced within a procedure. It is different to searching for the first consumer in one major way. When it reaches a recursive, mutually recursive, higher-order or method call it uses the conservative assumption that the procedure produces the variable immediately before returning. Therefore it uses the cost after the call as the cost after the variable is produced. This requires that the analysis traverses the procedure backwards in order to calculate the cost after the production of the variable by the time it reaches the producer. This allows the variable production time analysis to avoid part of the problem with retrieving costs for recursive call sites from the deep profiler.

5.2 Building Implicit Parallelisation Feedback

Calls are the only atomic goals that can take a non-trivial amount of time to execute. Non-atomic goals that contain a call goal may also have a non-trivial execution time, however a non-atomic goal may not always execute a call goal within it. Therefore we limit our initial implementation to consider only parallelising call goals against each other. Future work may wish to parallelise non-atomic goals against calls or other non-atomic goals. Also, this implementation only attempts to create parallel conjunctions of two conjuncts.

In Section 3 we introduced several formulas that can be used to calculate the speedup due to parallelism of conjunctions. Parallelising all the conjunctions within a program where $T_{Sequential} > T_{Parallel}$ will most likely degrade performance, because once all the CPUs are busy, any extra parallel computations will yield only extra overhead (since $T_{FailedParallel} > T_{Sequential}$). This will occur when the number of computations we try to run in parallel significantly exceeeds the computer's parallel processing capabilities. We therefore want to exploit only a smaller set of the parallelism opportunities. One simple heuristic is to parallelise only the conjunctions that have a speedup above a threshold whose size comfortably exceeds $T_{Overhead}$.

Equation 8 shows the calculation for the parallelisation speedup of an independent parallel conjunction of two goals A and B.

$$Speedup = T_A + T_B - max(T_A, T_B) - T_{Overheads} \tag{8}$$

Since $T_A + T_B - max(T_A, T_B) = min(T_A, T_B)$, if either T_A or T_B are smaller than $T_{Overheads}$, then it takes more time to spawn off the corresponding goal and execute it in parallel than to execute it sequentially, and *Speedup* will be negative. This is even more true for dependant parallel conjunctions, for which the available speedup is further limited by the need for synchronisation. Therefore we only consider parallelising calls that have an execution time greater than the overheads of parallelisation plus a speedup threshold. Procedures that have a higher cost are more likely to contain call sites that are worth parallelising. The parts of a program that contribute to most of its runtime are not always the longest-running parts of a program: code that runs quickly each time but is invoked many times can take more time overall than a longer-running piece of code that is invoked only a few times. Optimising the parts that contribute the most to the runtime is generally better than optimising the longest running parts. Therefore our implicit parallelisation analysis searches for parallelism opportunities only within procedures that have a *total* (not per-call) execution time above a configurable threshold. By *total* execution time we mean the total amount of time spent executing a procedure and its descendants.

Our analysis uses the program representation bytecode to scan the contents of such procedures to search for call sites whose per-call execution time is greater than the speedup threshold plus the parallel execution overheads. Calls may be parallelised against each other provided that they are in the same conjunction and separated by any number of trivial atomic goals such as unifications and calls to builtin procedures. It would be simple to extend this sot that it parallelises calls separated by trivial non-atomic goals — non-atomic goals that contain only trivial goals. An example is shown in Figure 10. Assuming that in each procedure, the first argument is input and the second is output, the unification

```
BeginningGoals,
p(A, B),
C = B + 3,
q(C, D),
EndGoals
```

Figure 10: Calls `a` and `b` may be parallelised

```
BeginningGoals,
(
  (
    p(A, B),
    C = B + 3
  ) &
  q(C, D)
)
EndGoals
```

Figure 11: Parallelisation of two calls separated by a unification

in the middle depends upon the call to `p/2` and the call to `q/2` depends on the unification, so these goals cannot be reordered.

Figure 11 shows one of two possible ways this code may be parallelised. The unification is placed in a plain conjunction that is the first conjunct of the parallel conjunction. Alternatively the unification could have been executed in sequence with the call to `q/2`. The optimal parallelisation depends on when `B` is produced by `p/2` and when `C` is consumed by `q/2`. The parallelisation shown in Figure 11 is most optimal when `B` is produced late by `p/2` and `C` is consumed late by `q/2`.

It is very common for unifications and calls to built in procedures to occur between calls, therefore our analysis is able to decide whether the goals between two calls should be run in sequence with either the first or second goals. Our analysis builds a variable dependency graph, and then queries this graph for each variable consumed by the second call. If it finds that it depends on a variable produced by the first call but only indirectly, then a goal between the calls must produce this variable. (In a mode correct program, this goal cannot be either before the first call or after the second.) Because the goals between the calls are trivial, there is no benefit in splitting them up, running some of them in sequence with the first call and some in sequence with the second call. If the time since the variable is produced in the first call is less than the time before the variable is consumed in the second call, then we will execute the goals between the calls in sequence with the first call, because this maintains the greatest amount of parallelism, otherwise we executed them in sequence with the second call. In either of these cases, we must adjust the formula for the expected parallel execution time, because either the first conjunct will produce the future at the very end of its computation or the second conjunct will consume the future at the beginning of its computation. We are currently working on extending this

to the case where there are several indirect dependencies between the calls.

Parallelising all call pairs whose parallel execution speedup is above a threshold may still result in embarrassing parallelism. Consider a recursive predicate such as `map/3` whose recursive case includes two calls, one of which is recursive. If these two calls are parallelised against each other, we will get embarrassing parallelism: if a call causes recursion 100 levels deep, we will spawn 100 sparks. Because these parallel executions are nested this parallelism will not be advantageous unless the computer they are executing on has at least 100 processors.

Granularity control is able to reduce the impact of poor parallelism decisions such as these. It works by transforming the program so that only every Nth recursive call creates a parallel execution where N is determined at compile time. Tannier [16] introduced such a transformation into the Mercury compiler. Such a granularity control transformation cannot assist in situations where parallel computations are nested but there is no recursion or mutual recursion.

Instead it is best to avoid creating nested parallel computations if that would lead to embarrassing parallelism. It is possible to use the deep profiling data, in particular the call graph of the program to do this. A possible implementation is to traverse the call graph incrementing the number of active parallel computations each time a parallelism opportunity that has a speedup above the speedup threshold is reached. This should be done until the number of active parallel computations exceeds the number of available processors on the target machine, once we reach that point, we can decide not to parallelise any computations below this point in the call graph. A call graph will usually have cycles, which the Mercury deep profiler represents as cliques; its call graph is a tree of cliques. This can be used together with other profiling information and the bytecode representation of the program to mimic a complete call graph with better information about recursive calls so that the analysis can avoid unbounded recursion. This feature has not been implemented at this time.

The programmer may modify the sources of the program or change the configuration of optimisations between building the profiling version of the program and the parallelised version of the program. This may cause differences between the bytecode representation of the program and the program that is to be parallelised by the compiler. We attempt to handle some subtle differences between these representations of the program so that parallelisation can still be performed in many cases. One such difference is that goals may have been added or removed from the conjunction containing the calls to be parallelised.

We have carefully selected what feedback information should be provided to the compiler. The information provided for each pair of calls to parallelise is:

- The module name, predicate name, arity and mode of the procedure containing this conjunction.
- A path through the tree of goals to the conjunction containing the two calls to parallelise within the procedure.
- The names of the modules and predicates of the callees that should be parallelised, these may be unknown if the call is a higher order or method call.
- The variable names used in the source program for the arguments of each of the calls (variables introduced by the compiler do not have useful names and are not included here). This is included because several calls to the same predicate may exist in a single procedure.

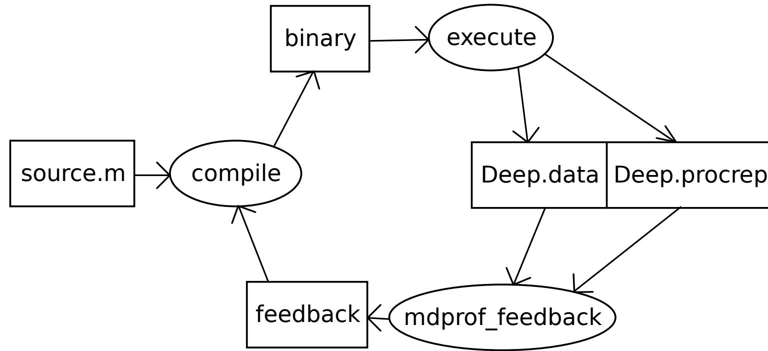

Figure 12: Profiler Feedback Loop

Provided that the differences between the analysed representation of the program and the representation the compiler parallelises are minimal, this information will be sufficient for the compiler to locate most of the parallelism opportunities that our optimisation has recommended. This is likely, because the programmer will usually use the implicit parallelism implementation when compiling a release version of their program. It is unlikely that they will make significant changes to the program after profiling it and before compiling the parallelised version.

6 Supporting Profiler Directed Optimisations

6.1 Profiler Feedback Framework

Our implicit parallelism analysis operates on profiling data and a bytecode representation of the program. A program compiled for both deep profiling and coverage profiling produces both of these when executed.

An automatic parallelisation tool may wish to traverse the entire call tree of the program. For example, to limit the amount of parallelism at each stage of execution to the number of CPUs available. Such a tool needs to be able to work on the whole program. The Mercury compiler never operates on the whole program: it compiles separate modules separately. We therefore implemented our implicit parallelisation analysis in a new tool called `mdprof_feedback`. For now, its task is to use profiling data and our variable use analysis to choose the conjunctions that are most profitable to parallelise, and to record this information in a *feedback file* that can then be given to the compiler. The data flow of the feedback process is shown in Figure 12.

Telling the compiler what to parallelise is just one use of profiler feedback. Other optimisations (such as inlining) could also benefit from profiling feedback.

Using a feedback-directed optimisation requires that the user compile and run their program before feedback information is produced, which takes a significant amount of time. Because of this, feedback-directed optimisations do not make sense unless the total execution time of the optimised program — which depends mostly on how many times it will be executed — is significantly greater than the time it takes to compile and execute the profiling version of

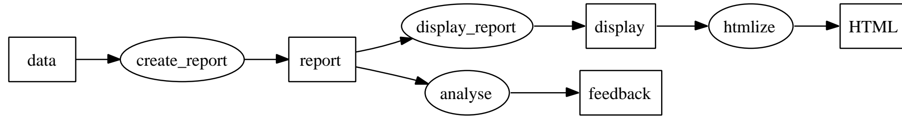

Figure 13: Deep profiler data flow

the program. We expect that feedback-directed optimisations would normally be used when the programmer is building a *release candidate* version of their program, after they have performed testing and fixed any bugs.

In all cases the program should be profiled on an input that is representative of the inputs the program is expected to handle when it is used normally. This is important to ensure the compiler makes decisions using feedback information that improves rather than degrades the performance of the program in general. How a representative input is chosen and how important the exact choice of the profiling input is will change from program to program.

We have designed and implemented a generic feedback framework that allows tools (both inside or outside the compiler) to create feedback information for the compiler. The feedback information can be expressed as any Mercury type, making the feedback framework very flexible. New feedback-directed optimisations may require feedback information from new analyses. We expect that many new feedback information types will be added to support these optimisations. We also expect that an optimisation may use more than one type of feedback information and that more than one optimisation may use the same feedback information.

The on-disc format of the feedback information file is very simple, it contains a header that identifies the file format, including a version number. What follows is a list of feedback information items. When the file is read in the file format identifier and revision number are checked, the list of information items is checked to ensure that no two items describe the same feedback information, for example there can only be at most one item in this list that describes how to automatically parallelise a program.

6.2 Refactoring the Deep Profiler Tools

The deep profiling tool reads in the deep profiling data and pre-processes it as described in Section 2.3. A user may then use the web based user interface to explore their program's profile by producing different reports about different parts of their program, including modules, procedures and cliques. These reports have traditionally been generated in one step, from the post-processed profiling data directly to HTML. This made it difficult to implement tools such as `mdprof_feedback`, since what they want is the *semantic* content of one or page pages that the user may want to display, without having to recover it by scraping HTML.

We have therefore have changed the deep profiler to generate HTML not directly but indirectly, going through two new data structures we designed. The first defines the semantic contents of a deep profiling *report*, the kind of information that tools like `mdprof_feedback` need. The second defines a medium-independent *display* structure that specifies how to display a report to the user

in terms of headers, paragraphs, text, tables and so on. We have written the code to convert a report structure into a display structure and to convert a display structure into HTML for display to the user. We then changed the implementation of the query types whose answers are needed by our implicit parallelism analysis to generate report structures instead of HTML; later, other members of the Mercury team similarly modified the implementation of the remaining query types. Figure 13 shows the data flows of the profiling tool and the feedback tool using the new structures.

The `mdprof_feedback` tool shares some of its code with the deep profiler tool. When it starts, it reads and pre-processes the profiling data and the program representation. Each analysis has access to this data and may use it to create a deep profiling report so that it can use more high-level information to perform its analysis. For example the implicit parallelisation analysis uses the top procedures report to find parallelism in the longest running procedures, and the procedure report to retrieve information about the call sites in each procedure. It also uses two new reports that we have implemented: one that returns a coverage annotated representation of a procedure definition, and another that returns the times when arguments are produced and consumed by a procedure along with the procedure's total cost.

Besides allowing us to generate feedback to the compiler about what conjunctions to parallelize, all these changes will also make it easier to develop both new profile feedback directed optimisations and new user interfaces to the profiling tool.

7 Conclusion

This thesis has described a implicit parallelism implementation for the Mercury programming language. Achieving the goal of correctly computing the speedup due to parallelisation of dependant conjunctions has been made possible by our coverage profiling implementation and feedback framework, which both provide other benefits.

The new feedback framework we have created will also make other feedback directed optimisations easier to write. Coverage profiling may be used for other profile directed analyses. For example inlining is a trade off between avoiding the overheads of procedure calls and the size of the program's compiled code. Coverage profiling provides execution counts on goals used within branching structures such as switches and if then else goals, this information can be used to inline procedure calls in branches that are executed most frequently. This minimises the number of procedure calls made by the program without unduly increasing the size of the compiled code.

Although our implicit parallelism implementation is incomplete — the compiler does not yet use the feedback information to parallelise the program — the remaining work is straight-forward.

Profiler-directed implicit parallelism is not perfect, a program parallelised using this approach may perform worse than its sequential equivalent when given some inputs. Although this is true for any parallelisation method, we believe that this is less of a problem for a profiler-directed approach, as such an approach will have access to performance metrics about the program when it is run with a representative input. This problem can be further addressed by

allowing the collection and exploitation of profiling data from more than one invocation of the program. Other worst case behaviours including embarrassing parallelism can be prevented by implementing algorithms such as parallelising calls nearer to the top of the program's call graph as we discussed in Section 5.2.

Features such as specialising procedures for parallelism could be added. Specialising procedures for parallelism generates both a sequential and parallel version of a procedure that should use sequential execution in some cases and parallel execution in others. The sequential version of the procedure can be used in cases where there are already a lot of parallel computations running. Other improvements include improving the accuracy of calculations that estimate when variables are consumed and produced, which can be done by calculating the expected execution time of a recursive call site or procedure as we discussed in Section 5.2. We expect that many more improvements can be made.

Our approach is different from the existing implicit parallelism strategies, of which the most closely related are work on Ciao Prolog [9] and the work of Shen, Costa and King [15]. Where they parallelised many computations by default and then attempted to prevent embarrassing parallelism by choosing to convert some of those parallel computations back into sequential computations, our implementation only parallelises computations that it believes will improve the performance of the program. Tannier [16] also chooses only to parallelise code that would lead to a speedup, however his implementation is too optimistic when calculating the speedup in dependant parallel conjunctions. It considers only the overheads of managing the each future and does not consider that execution of the second conjunct will most-likely be delayed until the first conjunct produces the future. Also, our analysis has access to a representation of the program including mode information. This information is required to determine which calls are dependant on other goals they might be parallelised with and determine how much speedup there may be in the case of dependant parallelism.

Overall this work brings the goal of good implicit parallelism optimisations closer by proving that it is possible to calculate the likely speedup due to parallelism in dependant conjunctions. The compiler should be able to use this information to exploit opportunities for parallelism in a program only when doing so is profitable and not when the overheads would outweigh the available speedup.

I would like to thank my supervisor Zoltan Somogyi. I have enjoyed working with him very much and look forward to working with him again. I would also like to thank the past and present contributors to the Mercury project for their contributions.. Thanks also to the free software community, without whose efforts I would not have had the resources to learn so much about computer programming. Finally, thanks to my wife Liz, who in the last few weeks has tolerated the importance of this work over other tasks, namely the mountain of dishes on our sink.

References

[1] David Butenhof. *Programming with Posix Threads*. Addison-Wesley, Boston, USA, 1997.

[2] Lawrence Byrd. Understanding the control flow of Prolog programs. In *Proceedings of the 1980 Logic Programming Workshop*, pages 127–138, Debrecen, Hungary, July 1980.

[3] Keith Clark and Steve Gregory. Notes on systems programming in PARLOG. In ICOT, editor, *The international conference on fifth generation computer systems*, pages 299–306. ICOT, ICOT, 1984.

[4] Keith Clark and Steve Gregory. PARLOG: parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, January 1986.

[5] Thomas Conway. *Towards parallel Mercury*. PhD thesis, Department of Computer Science and Software Engineering, The University of Melbourne, Melbourne, Australia, July 2002.

[6] Thomas Conway and Zoltan Somogyi. Deep profiling: engineering a profiler for a declarative programming language. Technical report, Department of Computer Science and Software Engineering, University of Melbourne, Melbourne, Australia, July 2001.

[7] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *The tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60, New York, NY, 2005, 2005.

[8] Tim Harris and Satnam Singh. Feedback directed implicit parallelism. *SIGPLAN Not.*, 42(9):251–264, 2007.

[9] P. Lopez, M. Hermenegildo, and S. Debray. A methodology for granularity-based control of parallelism in logic programs. *Journal of Symbolic Computation*, 22(4):715–734, 1996.

[10] Leon Mika. Software transactional memory in Mercury. Technical report, Department of Computer Science and Software Engineering, Melbourne University, Melbourne, Australia, October 2007.

[11] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[12] Vijay A. Saraswat. *Problems with Concurrent Prolog*. PhD thesis, Carnegie-Mellon University, 1985.

[13] Vijay A. Saraswat. The concurrent logic programming language CP: Definition and operational semantics. Carnegie-Mellon University, September 1986.

[14] Ehud Shapiro. Flat Concurrent Prolog as a general-purpose parallel machine language. The Weizmann Institute of Science.

[15] Kish Shen, Vïtor Santos Costa, and Andy King. Distance: A new metric for controlling granularity for parallel execution. *Journal of Functional and Logic Programming*, 1999 (Special Issue 1), 1999.

[16] Jéröme Tannier. Parallel Mercury. Master's thesis, Institut d'informatique, Facultés Universitaires Notre-Dame de la Paix, 21, rue Grandgagnage B-5000 Namur Belgium, 2007.

[17] Kazunori Ueda. *Guarded Horn Clauses*. PhD thesis, University of Tokyo, March 1986.

[18] Peter Wang. Parallel Mercury. Honours thesis, Department of Computer Science and Software Engineering, The University of Melbourne, Melbourne, Australia, October 2006.

[19] Peter Wang and Zoltan Somogyi. Minimising the overheads of dependant AND-parallelism. Currently a draft, October 2008.