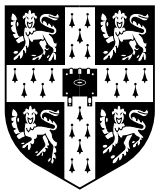
Extending old languages for new architectures

Leo P. White



University of Cambridge
Computer Laboratory
Queens' College

October 2012

This dissertation is submitted for
the degree of Doctor of Philosophy

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated in the text.

This dissertation does not exceed the regulation length of 60 000 words, including tables and footnotes.

Extending old languages for new architectures

Leo P. White

Summary

Architectures evolve quickly. The number of transistors available to chip designers doubles every 18 months, allowing increasingly complex architectures to be developed on a single chip. Power dissipation issues have forced chip designers to look for new ways to use the transistors at their disposal. This situation inevitably leads to new architectural features on a fairly regular basis. Enabling programmers to benefit from these new architectural features can be problematic.

Since architectures change frequently, and compilers last for a long time, it is clear that compilers should be designed to be extensible. This thesis argues that to support evolving architectures a compiler should support the creation of high-level language extensions. In particular, it must support extending the compiler's middle-end. We describe the design of EMCC, a C compiler that allows extension of its front-, middle- and back-ends.

OpenMP is an extension to the C programming language to support parallelism. It has recently added support for task-based parallelism, a dynamic form of parallelism made popular by Cilk. However, implementing task-based parallelism efficiently requires much more involved program transformation than the simple static parallelism originally supported by OpenMP. We use EMCC to create an implementation of OpenMP, with particular focus on efficient implementation of task-based parallelism.

We also demonstrate the benefits of supporting high-level analysis through an extended middle-end, by developing and implementing an interprocedural analysis that improves the performance of task-based parallelism by allowing tasks to share stacks. We develop a novel generalisation of logic programming that we use to concisely express this analysis, and use this formalism to demonstrate that the analysis can be executed in polynomial time.

Finally, we design extensions to OpenMP to support heterogeneous architectures.

Acknowledgments

I would like to thank my supervisor Alan Mycroft for his invaluable help and advice. I would also like to thank Derek McAuley for helpful discussions. I also thank Netronome for funding the work. Special thanks go to Zoë for all her support.

Contents

Chapter 1

Introduction

1.1 Evolving architectures require extensible compilers

The number of transistors available to chip designers doubles every 18 months, allowing increasingly complex architectures to be developed on a single chip. Power dissipation issues force chip designers to look for innovative ways to use the transistors at their disposal.

This situation inevitably leads to new architectural features on a fairly regular basis. Recent changes have lead to multi-core chips with a greater variety of cores and increasingly complex memory systems. Allowing programmers to benefit from such features can be problematic.

Compilers have very long lifetimes. The most popular C/C++ compilers in the world today are arguably Microsoft Visual C++ [44] and the GNU Compiler Collection (GCC) [68], which are 29 and 25 years old respectively. Compilers are very complex systems with large codebases. There are also very few economic incentives for creating a new compiler. This means that there is little competition from new compilers, and so old compilers are rarely replaced. The only notable exception in the last 20 years has been the arrival of the LLVM compiler framework.

Since architectures change frequently, and compilers last for a long time, it is clear that compilers should be designed to be extensible. Computer architectures can be complicated, esoteric and notoriously poorly specified: producing efficient machine code for them requires specialist knowledge. This makes it very important that a compiler's extensibility is both exposed and simple to use, in order that the extensions may be written by experts in the architecture, rather than experts in the compiler.

1.2 Extensibility and compilers

A system is said to be extensible if changes can be made to the existing system functionalities, or new functionalities added, with minimum impact to the rest of the system. In software engineering extensibility is often considered a *non-functional requirement*: a requirement that is a quality of the system as a whole rather than part of the system's actual behaviour. From this perspective extensibility refers to the ease with which developers can extend their software with additional functionalities. For example, software quality frameworks, such as ISO/IEC 9126 [35], include extensibility as a sub-characteristic of maintainability.

Extensibility can also be part of the functional requirements of a system. Such systems allow their users to add additional functionality to them, and ensure that such extensions will continue to work across multiple releases of the system. In this thesis we will mostly consider *non-functional extensibility*, although most of the ideas and tools described could also be applied to functional extensibility.

Extensibility is closely related to *modularity*, which is the degree of separation between the components of a system. The more autonomous the modules the more likely that an extension will only affect a small number of modules, rather than requiring changes across the entire system. Extensibility can also be considered a form of *code reuse*: rather than write a new system for our new problem we can reuse the code from a previous system.

An important aspect of extensibility and modularity is the minimisation and explicit tracking of dependencies. Dependencies between the different components of a system decrease the modularity of that system: changes to one component may require changes in another. Where dependencies cannot be avoided they should be explicit in the system. If dependencies are not explicit then changing one component requires checking all other components for possible dependencies on the part of the component that has been changed, a difficult and error prone task. By making dependencies explicit the amount of the system that must be checked is minimised, decreasing the difficulty and risk in extending the system.

The purpose of a compiler is to translate from an *input language* to an *output language*. Extending a compiler means extending this translation, by either changing how the input language is translated or extending the input or output languages. For example, a compiler could be extended by adding a new syntactic form to the input language, or by adding a new optimisation.

In practise, compilers translate through a series of *intermediate languages*. In order to extend the input and output languages, it is important to also be able to extend these intermediate languages. Otherwise, analyses and optimisations that operate on these intermediate languages cannot be applied to the language extensions.

1.2.1 The front-, middle- and back-ends

A typical compiler is divided into three stages (Fig. 1.1):

Front-end The front-end is divided into three phases:

i) Lexical analysis
ii) Syntactic analysis
iii) Semantic analysis

Lexical analysis converts the source code into a stream of tokens. Syntactic analysis builds an Abstract Syntax Tree (AST) from the stream of tokens. Semantic analysis collects semantic information from the AST and then translates the AST into a more "semantic" form, called an Intermediate Language (IL). During these three phases, the front-end also checks the program for errors.

Middle-end The middle-end optimises the program. In order to decide what optimisations should be applied, the middle-end performs analyses to deduce additional semantic information about the program. This new semantic information may also be used to translate the program into other ILs that are more suited to performing optimisations.

Back-end The back-end takes the program from the middle-end and generates the final output, typically machine code. It may also perform some machine-dependent low-level optimisations.

Separating the compiler into these three stages allows the same compiler to accept multiple languages (through multiple front-ends) and target multiple architectures (through multiple back-ends). The increased modularity provided by this three stage model also improves the extensibility of the compiler: simple extensions may well be confined to a single front- or back-end.

1.2.2 Declarative specification of languages

The division of the compiler into three stages means that we can extend an input or output language by writing a new front- or back-end for the compiler. However, this is a large and complex task, we would rather extend an existing front- or back-end. How extensible are these front- and back-ends?

Both front- and back-ends are very complicated and interconnected systems, which can make them difficult to extend. The traditional approach to improving the extensibility of these stages is to implement them using *declarative language specifications*. This enables developers to specify the properties of an input or output language which are then mechanically converted into (parts of) the front- and back-ends.

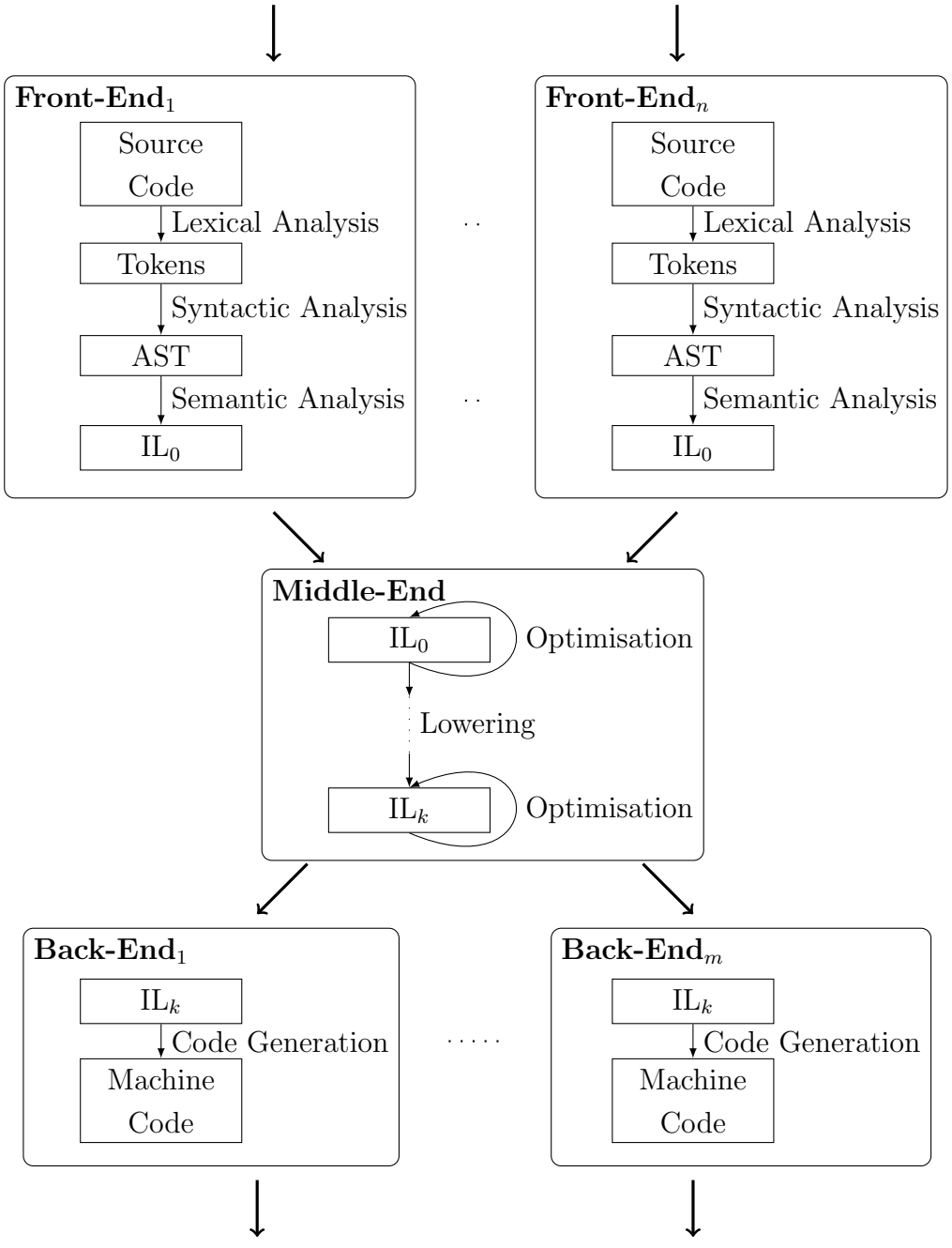


Figure 1.1: A Typical Compiler

The classic example of this is the use of regular expressions to specify the lexical analysis and context-free grammars to specify the syntactic analysis of a front-end. Traditional tools for this, such as Lex[49] and Yacc[38], make it easier to extend a front-end by adjusting the declarative specification of its input language. More recently, some compilers (e.g. JustAdd [32]) have also supported declarative specification of the semantic analysis of the input language using attribute grammars.

Similar techniques have also been very successful in specifying the output languages of compilers. Different back-ends are built from *machine descriptions*, which are used to derive rules for instruction selection, instruction scheduling and register allocation. This makes it easy to define a new back-end for a new or updated architecture. The fast rate of evolution of architectures and desire for portability has made this a necessity.

1.2.3 Preprocessors and Macros

An alternative to implementing a new front-end to extend the compiler's input language is to use a *preprocessor*. A preprocessor is itself a form of compiler, whose output language is the input language of the main compiler. The classic example is the C preprocessor which is used to expand simple text-based macros as part of the C language.

Most preprocessors are for low-level extensions. The translation is usually done without performing any semantic analysis of the program, and often without any syntactic analysis.

Preprocessors can be used to implement quite large high-level language extensions. For example, the Cilk [70] language is implemented as a preprocessor for C. These preprocessors implement a full syntactic and semantic analysis to produce an AST annotated with semantic information. The extensions are then translated out of the AST and the result is pretty-printed. Such preprocessors are very difficult to implement as they contain most of a compiler's front-end.

Another method of extending a compilers input language is to use macros. Macros are procedures that are run before or during the compiler's front-end. They might be built into the compiler's input language, as with Lisp, or part of a preprocessor, as with C. Their capabilities vary widely, but since they are executed at compile-time, they can be considered a mechanism for extending a compiler.

Macros are a form of functional extensibility: designed for use by a compiler's users. They are particularly popular for implementing *Domain Specific Languages(DSLs)*. DSLs are small languages for use in specific programming domains. An *embedded DSL* is a DSL that exists within another programming language. Embedded DSLs are a form of language extension, and they have become increasingly popular in recent years, coinciding with the creation of increasingly powerful macro systems (e.g. Template Haskell [67], Scala Macros [11]).

1.3 Extending old languages for new architectures

Architectures change frequently, and compilers must be extended to support the new features of a new architecture.

For many low-level architectural features, extending the back-end is all that is required to enable programmers to exploit them. For example, adding a new instruction to an architecture may only require that a template for that instruction be added to the instruction selection process. However, sometimes the best way to use a new architectural feature is through a language extension.

None of the forms of extensibility discussed in the previous section support extending the middle-end of the compiler. They all handle input language extensions by translating them into the AST of the original language. This means that all analyses and optimisations used to implement the extension must be carried out on the AST. However, ASTs are not a suitable representation for many optimisations. They contain syntactic artefacts which can add a lot of unnecessary complexity to an analysis' implementation. It also means that the analyses cannot be shared with a similar extension to another language.

In general, the problem with supporting language extensions exclusively through extensions to the front-end is that the analyses (and optimisations) applied to the extension are completely separate from those applied to the base language. This means that if an optimisation of the extension requires an existing analysis of the base language, that analysis must be reimplemented so that it can be applied in the front-end. More importantly, it means that there is no "semantic integration" of the extension with the rest of the language. The analyses and optimisations of the base language remain completely oblivious to the extension and any semantic information it may contain. If our extensions are to have parity with the rest of the language, we must be able to extend the middle-end of the compiler.

Extending the middle-end of a compiler, requires us to be able to extend the compiler's intermediate languages. This is difficult because it may require changes to any of the front-ends, back-ends or optimisations, all of which depend on intermediate languages. The problem is that the front-ends, back-ends and optimisations all access the intermediate language directly: there is no abstraction between them which might allow a front-end to target multiple intermediate languages, or allow an optimisation to operate on any intermediate language that provides the right operations.

This problem can also be thought of a one of tracking dependencies at a too coarse-grained level. Front-ends, back-ends and optimisations all depend on intermediate languages as a whole, rather than specifying which parts or properties of the intermediate language they actually depend on. This makes it very difficult to know which parts of the compiler must be changed to accommodate extensions to the intermediate languages.

Chapter 3 describes the design and implementation of a compiler that supports ex-

tensions to its middle-end. Section 3.1 contains a discussion of existing approaches to extending middle-ends.

1.4 Compilers for multi-core architectures

The most dramatic recent change in architecture design has been the arrival of multi-core on PCs. While parallelism has been a feature of architectures for many years, its arrival on desktop computers has dramatically increased the number of programmers writing programs for parallel architectures.

However, despite multi-core having been available in desktop computers since 2005 (and simultaneous multithreading as early as 2002), the compilers used by most programmers targeting these architectures are still oblivious to the existence of parallelism.

This means that the only support for shared-memory parallelism is through threading libraries. Such support is inherently low-level and, as discussed by Boehm [8], is actually unable to guarantee the correctness of the translation of multi-threaded programs. In 2011, both C and C++ finally added some native support for shared-memory parallel programming. However, it is mostly provided through standard library support for explicit multi-threading. Only the new atomic operations require special handling within the compiler, which allows them to avoid the correctness issues which affect threading libraries.

This reluctance among compiler implementers to make the compiler aware of parallelism has meant that the middle-ends of these compilers have not changed despite a dramatic change in the computation model. If the middle-end is not aware of parallelism, then it cannot perform analyses and optimisations of parallel code. There are many examples in the academic literature of analyses and optimisations for parallel programs (e.g. deadlock detection). However, these are only described in terms of small calculi, rather than implemented in working compilers. If the middle-end of a compiler were extended to accommodate parallelism, then these analyses and optimisations could be applied to real-world programs.

Further, this lack of support for parallelism in the middle-end of compilers prevents the adoption of higher-level approaches to parallelism, which require analyses and optimisation to be implemented efficiently. This means that explicit multi-threaded programming remains the most common approach to parallelism, despite general acceptance that it is difficult to reason with and produces hard-to-find bugs.

1.5 OpenMP: Extending C for multi-core

This thesis will look at OpenMP as a language extension to handle the addition of multi-core to PC architectures. OpenMP is an interesting example as it was originally designed to be simple enough to implement with a preprocessor, but more recent additions to the language require more complex analyses in order to be implemented efficiently. Chapter 5 describes an efficient implementation of OpenMP using our EMCC compiler.

OpenMP [61] is a shared-memory parallel programming language that extends C with compiler directives for indicating parallelism. It was originally designed for scientific applications on multi-processor systems. It provided a higher level of parallelism than that provided by threading libraries, but did not require complicated program transformation, and could be implemented in the front-end of the compiler or with a simple preprocessor.

The rise of multi-core has caused OpenMP to evolve towards supporting mainstream applications. These applications are more irregular and dynamic than their scientific counterparts. With this in mind, OpenMP 3.0 included support for *task-based* parallelism.

Task-based parallelism is a high-level parallel programming model made popular by languages such as Cilk [70]. Implementing it efficiently requires much more involved program transformation than the simple static parallelism originally supported by OpenMP. These transformations are best implemented in the middle-end of the compiler.

However, current implementations of OpenMP are still being implemented in the front-end of the compiler. This has prevented their performance from competing with that of Cilk and other task-based programming languages [60].

1.6 Contributions

The main contributions of this thesis are:

- The design and implementation of EMCC, a C compiler that allows extensions of its front-, middle- and back-ends. The middle-end is made extensible by abstracting its IL using functors.

- The design of a customisable library for atomic operations and concurrent data structures. The design of this library makes it easy to use with a new architecture or a new programming model.

- A new implementation of the OpenMP programming language which takes advantage of our extensible compiler to implement OpenMP tasks using more lightweight methods than previous implementations, allowing it to match Cilk in terms of performance.

- A theoretical demonstration that OpenMP tasks can be implemented in a space-efficient way without affecting time efficiency.
- The design of an analysis of OpenMP programs, which detects when it would be safe for multiple tasks to share a single stack. This optimisation is implemented using EMCC.
- A novel generalisation of logic programming that we use to concisely express the above analysis. This enables us to demonstrate that the analysis can be executed in polynomial time.
- The design of extensions to OpenMP to support heterogeneous architectures. These extensions allow the programmer to chose how work is allocated to different processing elements on an architecture. They are implemented using EMCC and our customisable run-time library.

1.7 Dissertation outline

The rest of this dissertation proceeds as follows: Chapter 2 details the technical background required by the other chapters. Chapter 3 describes the design of EMCC. Chapter 4 describes the design of our customisable run-time library. Chapter 5 details how we implemented OpenMP using EMCC. It also contains a discussion of the space-efficiency of OpenMP, and a lightweight method for implementing OpenMP tasks. Chapter 6 describes the design and implementation of a novel optimisation for OpenMP programs, which allows multiple tasks to share a single stack. Chapter 7 describes extensions to OpenMP to support heterogeneous architectures. Chapter 8 concludes.

Chapter 2

Technical background

This chapter describes some technical material used in the rest of this thesis. Chapter 3 describes a compiler implemented in the OCaml programming language, so Section 2.1 describes some of the features of that language. Chapter 5 describes an implementation of OpenMP's task-based parallelism, so the Sections 2.2 and 2.3 of this chapter give outlines of task-based parallelism and OpenMP respectively. Chapter 5 also contains some discussion of the theoretical efficiency of OpenMP programs, for this discussion we develop a model of parallel computation in Section 2.4. Chapter 6 describes an analysis of OpenMP program's memory usage, we present this analysis using a novel generalisation of logic programming, which is described in Section 2.5.

2.1 The OCaml programming language

The EMCC compiler is implemented in the OCaml [48] programming language, and parts of its design rely on features of that language. This section contains an outline of the features of OCaml relevant to the design of EMCC.

OCaml, originally known as Objective Caml, is a programming language from the ML family. It was created in 1996 based on a previous language called Caml. In addition to the traditional ML features it includes a number of additions, including support for object-oriented programming.

2.1.1 The module system

The OCaml module system is based on the ML module system. It provides a means to group together and encapsulate collections of types and values. There are three key parts in the module system: *signatures*, *structures*, and *functors*. Signatures correspond to interfaces, structures correspond to implementations, and functors are functions over structures.

Fig. 2.1 shows some simple code using the OCaml module system. `EQUALSIG` is a signature with one type (`t`) and one function (`equal`). `MakeSet` is a functor that takes a parameter (`Equal`) that obeys the `EQUALSIG` signature, and produces a structure that implements a simple set. The `StringEqual` and `StringNoCase` structures both implement the `EQUALSIG` signature for strings. The `StringSet` and `StringSetNoCase` structures are created by applying the `MakeSet` functor to `StringEqual` and `StringNoCase` respectively. Both `StringSet` and `StringSetNoCase` implement sets of strings, but `StringSetNoCase` compares strings case-insensitively.

The important point about this code, and the OCaml module system in general, is that it abstracts the details of the implementations of `StringEqual` and `StringNoCase` from the code in `MakeSet`. This allows `MakeSet` to be used to create different set implementations that have the same interface.

2.1.2 The class system

The OCaml class system provides structurally typed objects and classes. Like most object-oriented systems, the OCaml class system provides support for *open recursion*. Open recursion allows mutually recursive operations to be defined independently. Fig. 2.2 shows a simple example using open recursion.

The `int_string_list` and `string_int_list` types represent lists of alternating integers and strings. The `printer` class has two mutually recursive methods `int_string_list`, which creates a string from an `int_string_list`, and `string_int_list`, which creates a string from a `string_int_list`. The `quoted_printer` class inherits from the `printer` class and overrides the `string_int_list` with a version which surrounds strings in the list with single quotation marks.

The open recursion in this example comes from the mutual recursion between the `printer` class's `int_string_list` method and the `quoted_printer` class's `string_int_list` method. These methods call one another (through the special `self` variables) even though they are defined independently.

2.1.3 Generalised Algebraic Datatypes

Generalised Algebraic Datatypes (GADTs) are an advanced feature of OCaml that allows us to encode constraints about how data can be constructed, and have the OCaml type checker ensure that those constraints are obeyed. Fig. 2.3 shows the classic example of GADTs: an evaluator for simple typed expressions.

The `expr` GADT represents simple typed expressions. The types of these expressions are represented by the OCaml types `int` and `bool`. Integer expressions are represented by the type `int expr`, whilst boolean expressions are represented by the type `bool expr`. By

```
module type EQUALSIG = sig
  type t
  val equal : t -> t -> bool
end

module MakeSet (Equal : EQUALSIG) =
struct
  type elt = Equal.t
  type set = elt list
  let empty = []
  let mem x s = List.exists (Equal.equal x) s
  let add x s = if mem x s then s else x :: s
  let find x s = List.find (Equal.equal x) s
end

module StringEqual  = struct
  type t = string
  let equal s1 s2 = (s1 = s2)
end

module StringNoCase  = struct
  type t = string
  let equal s1 s2 =
    String.lowercase s1 = String.lowercase s2
end

module StringSet = MakeSet(StringEqual)
module StringSetNoCase = MakeSet(StringNoCase)
```

Figure 2.1: An example of the OCaml module system

```
type int_string_list =
    INil
  | ICons of int * string_int_list
and string_int_list =
    SNil
  | SCons of string * int_string_list

class printer = object (self)
  method int_string_list = function
    INil -> ""
  | ICons(i, sil) ->
      (string_of_int i) ^ "; " ^ (self#string_int_list sil)
  method string_int_list = function
    SNil -> ""
  | SCons(s, isl) ->
      s ^ "; " ^ (self#int_string_list isl)
end

class quoted_printer = object (self)
  inherit printer
  method string_int_list = function
    SNil -> ""
  | SCons(s, isl) ->
      "'" ^ s ^ "'; " ^ (self#int_string_list isl)
end

let p = new printer

let q = new quoted_printer

let s1 = p#int_string_list (ICons(3, SCons("hello", ICons(5, SNil))))

let s2 = q#int_string_list (ICons(3, SCons("hello", ICons(5, SNil))))
```

Figure 2.2: An example of the OCaml class system

```
type 'a expr =
    Const : 'a -> 'a expr
  | Plus : int expr * int expr -> int expr
  | LessThan: int expr * int expr -> bool expr
  | If: bool expr * 'a expr * 'a expr -> 'a expr

let rec eval : type a. a expr -> a = function
    Const x -> x
  | Plus(a, b) -> (eval a) + (eval b)
  | LessThan(a, b) -> (eval a) < (eval b)
  | If(c, t, f) -> if (eval c) then (eval t) else (eval f)

let x = eval (If(LessThan(Const 1, Const 2),
                 Plus(Const 3, Const 2),
                 Const 0))
```

Figure 2.3: An example of GADTs

using a GADT the OCaml type-checker ensures that only correctly typed expressions can be created—for example, we cannot create an expression that tries to sum two boolean expressions. Since all expressions are correctly typed, we can write an evaluation function for the expressions that is guaranteed to succeed (`eval`).

2.2 Task-based parallelism

With the advent of multi-core processors, many programming languages have introduced parallel programming models. These programming models can be divided into *data-parallel* models, which focus on allowing an operation to be performed simultaneously on different pieces of data, and *task-parallel* models, which focus on allowing multiple threads of control to perform different operations on different data.

Task-parallel programming models can be further classified as either *static* or *dynamic* task-parallel programming models. Static task-parallel models use a fixed number of threads and are optimised for computations with long-lasting threads and a similar number of threads to the amount of physical parallelism in the architecture. Dynamic task-parallel models support the frequent creation and destruction of threads and are optimised for computations using many short-lived threads.

Dynamic task parallelism divides computations into an increased number of smaller tasks compared to static task parallelism, which increases the amount of logical parallelism exposed to the system. By encouraging programmers to increase the amount of logical parallelism exposed to the system, dynamic task-parallel programming models can in-

crease system utilisation in the presence of delays (synchronisation costs, communication latency, etc.) and load-imbalance. This works because excess parallelism can be scheduled during delays and short-lived threads are easier to divide up evenly. However, each thread requires a number of resources (e.g. a full execution context), and these resources have associated time and space costs.

A common approach to dynamic task parallelism is *task-based parallelism*. This executes program threads using a fixed number of *worker threads*—often implemented as heavyweight kernel threads. To differentiate program threads from worker threads they are often referred to as *tasks*—hence the name task-based parallelism. Tasks are scheduled onto threads cooperatively (i.e. no preemption) at run-time. This division between worker threads and tasks allows tasks to be more lightweight—requiring fewer resources per-task and enabling the system to support many more tasks simultaneously.

As the granularity of parallelism is refined, the time cost of supporting many tasks could ultimately outweigh the benefit of the potential increase in utilisation. Furthermore, programmers have certain expectations about how a program's space costs may increase when executed in parallel (we say that a program that meets these expectations is *space efficient*). As the granularity of parallelism is refined the space cost of supporting many tasks may mean that a program fails to meet these expectations.

The simplest method of reducing the cost of supporting many tasks is *load-based inlining*. This means that once some measure of load (e.g. the number of tasks) reaches a certain level (called the *cut-off*) any new tasks are *inlined* within an existing task. When a new task is inlined within an existing task, the existing task is put on hold while the new task executes using its resources, then when the new task has finished the original task is allowed to continue. This inlining can be done cheaply using an architecture's procedure call mechanism. However, inlining prevents tasks executing in parallel with their parent, effectively reducing dynamic task parallelism to static task parallelism.

The alternative to inlining is to make tasks as lightweight as possible, so that the system can support as many tasks as required without using up the available resources. In particular, it is important that resource usage scales linearly with the number of tasks. This is more difficult to implement than load-based inlining, but it does not constrain the parallelism of the program so can scale much more efficiently.

2.2.1 Support for task-based parallelism

A number of languages and libraries provide support for task-based parallelism. These include Cilk, OpenMP and Intel's Threading Building Blocks.

Cilk [70] is a parallel programming language developed since 1994. Parallelism is expressed by annotating function definitions with the `cilk` keyword. Functions marked by this keyword are treated as tasks and all calls to them must be annotated with the `spawn`

```
cilk int fib(int n)
{
  if (n<2) return n;
  else
  {
    int x, y;
    x = spawn fib (n-1);
    y = spawn fib (n-2);
    sync;
    return (x+y);
  }
}
```

Figure 2.4: An example of some Cilk code

keyword. Further keywords are provided to allow tasks to synchronise and communicate with each other. Cilk places various restrictions on where tasks can be created and how tasks synchronise with each other (e.g. a task cannot outlive its parent). These restrictions provide certain guarantees about the behaviour of tasks in Cilk. An example of Cilk is shown in Fig. 2.4.

OpenMP is also based on the C programming language, and since version 3.0 [61] has provided support for task-based parallelism. It is the focus of much of the work in this thesis, and is described in detail in the next section.

Intel's Threading Building Blocks [64] is a C++ library with support for task-based parallelism. Since it is a library, it cannot use any program transformations on the bodies of its tasks. This forces it to restrict the parallelism of its tasks. Sukha [69] showed that, in the worst case, such restrictions can remove all the parallelism from a task-based program.

Other programming languages and libraries with support for task-based parallelism include Microsoft's Task Parallel Library [47], X10 [13] and Wool [25].

2.3 OpenMP

OpenMP is a shared-memory parallel programming language that was first standardised in 1996. It was originally designed for scientific applications on large clusters. Parallelism is expressed by annotating a program with compiler directives. The language originally only supported data parallelism and static task parallelism. However the emergence of multi-core architectures has brought mainstream applications into the parallel world. These applications are more irregular and dynamic than their scientific counterparts, and require more expressive forms of parallelism. With this in mind, in 2008 the OpenMP

Architecture Review Board released OpenMP 3.0 [61], which includes support for task-based parallelism.

Before the addition of task-based parallelism, OpenMP did not require complicated program transformation, and could be implemented in the front-end of the compiler or with a simple preprocessor. However, implementing task-based parallelism efficiently requires much more involved program transformation and these transformations are best implemented in the middle-end of the compiler. Chapter 5 discusses the requirements of an efficient implementation of task-based parallelism in OpenMP.

The execution model of OpenMP within the parallel sections of a program consists of *teams* of *threads* executing *tasks* and *workshares*. These teams are created using the `parallel` directive.

Workshares support data parallelism: they divide work amongst the threads in a team. For instance, the `for` workshare allows the iterations of a `for` loop to be divided amongst the threads.

More dynamic forms of parallelism can be exploited using tasks. Tasks are sequences of instructions to be executed by a thread. Tasks do not need to be executed immediately, but can be deferred until later or executed by a different thread in the team. When a team is created each thread begins executing an initial task. These tasks can in turn *spawn* more tasks using the `task` directive. A task can also perform a *sync* operation using the `taskwait` directive, which prevents that task from being executed until all of the tasks that it has spawned have finished.

There are two types of OpenMP task that can be spawned: *tied* and *untied*. There are two restrictions related to tied tasks:

1. Once a thread has started executing it, a tied task cannot migrate to another thread. We say that the task is *tied* to the thread that started it.

2. A thread can only start executing a new task if that task is descended from all the tied tasks that are currently tied to that thread.

OpenMP has a relaxed-consistency shared-memory model. All threads have access to memory and are allowed to maintain their own *temporary views* of it. This temporary view allows a compiler to cache variables accessed by a thread and thereby avoid going to memory for every reference to a variable.

Some simple OpenMP C code is shown in Fig. 2.5. This code uses tasks to perform a parallel post-order traversal of a binary tree data structure. The `taskwait` directive ensures that the child nodes are processed before the parent node.

```
void postorder_traverse(tree_node *p) {
    if (p->left)
      #pragma omp task
          postorder_traverse(p->left);
    if (p->right)
      #pragma omp task
          postorder_traverse(p->right);
    #pragma omp taskwait
    process(p);
}
```

Figure 2.5: An example of some OpenMP C code

2.4 Models for parallel computations

Chapter 5 includes a discussion of the efficiency of OpenMP programs and task-based programs in general. This discussion requires a model of parallel computation.

This section develops a simple model of parallel computations and their execution on a parallel architecture. It generalises the model used by Blumofe et al. [7], so that it can incorporate the computations produced by OpenMP programs. This model consists of executing *computations* according to *execution schedules*. Blumofe et al. used this model to show that Cilk programs could all be executed in a time- and space-efficient way.

We define the execution time and space for computations in this model. We also define the concept of a *scheduling algorithm* and what it means for a scheduling algorithm to be time or space efficient.

2.4.1 Parallel computations

A *parallel computation* is a directed acyclic graph (N, E) where N is a set of nodes, labelled by instructions, and E represents dependencies between instructions. We write $\sqsubset_E$ for the transitive closure of E. A simple computation is shown in Fig. 2.6.

Instructions take unit time and may include access to data. A computation's data is represented by a set V of *variables*. We denote the set of instructions that access a variable $v \in V$ by $\mathcal{A}_v \subseteq N$.

For any $p \in \mathbb{N}$ we define a *p-way ordering* of a computation (N, E) to be a tuple of partial functions $\tilde{\omega} = \omega_1, \ldots, \omega_p : \mathbb{N} \rightharpoonup N$ satisfying three conditions:

1. (order respecting): $\omega_k(i) \sqsubset_E \omega_l(j) \implies i < j$ whenever $\omega_k(i)$ and $\omega_l(j)$ are defined
2. (injective): $\omega_k(i) = \omega_l(j) \implies i = j \wedge k = l$ whenever $\omega_k(i)$ and $\omega_l(j)$ are defined
3. (surjective): $\forall n \in N\ \exists k, i\ .\ \omega_k(i) = n$.

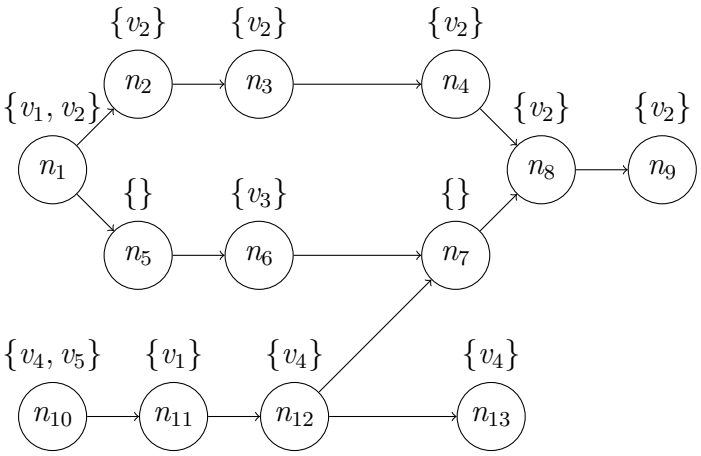


Figure 2.6: A parallel computation with instructions $n_1, \ldots, n_{13}$ and variables $v_1, \ldots, v_5$

The last two properties mean that $\tilde{\omega}$ gives a unique index for every instruction. These indices can be found using the function $\omega^{-1} : N \to \mathbb{N}$ defined uniquely (as a form of inverse to $\tilde{\omega}$) by the requirement

$$\forall n \in N \,.\, \exists k \leq p \,.\, \omega_k(\omega^{-1}(n)) \;=\; n$$

Note that most parallel languages do not support every possible computation. We call such restrictions *computation restrictions*. By restricting the patterns of dependencies and variable accesses, such languages are able to provide certain guarantees about the execution behaviour of their computations.

2.4.2 Execution schedules

The physical parallelism of an architecture is represented by a fixed number of *worker threads*, and its memory is represented by a set L of *locations*.

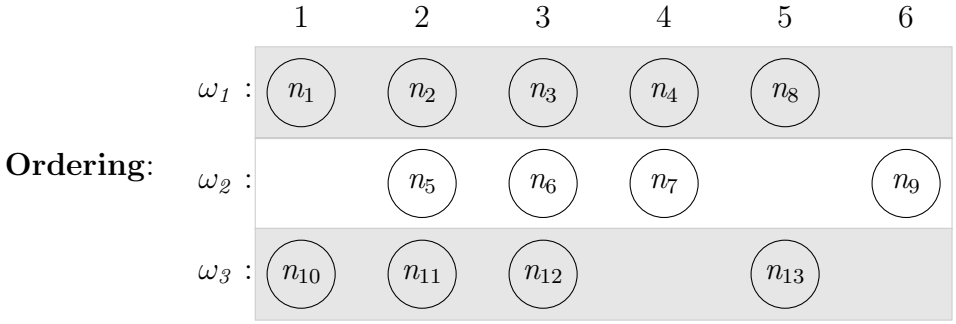
An *execution schedule*[1] X for a parallel computation c $(= (N, E))$ on a parallel architecture with p worker threads consists of:

(i) $\tilde{\omega}$, a p-way ordering of c

(ii) $\ell : V \to L$, a storage allocation function

The ordering $\tilde{\omega}$ determines which instruction is executed by each thread for a particular step in the execution, so that at time-step i the nodes $\omega_1(i), \ldots, \omega_p(i)$ are executed. Note that if $\omega_k(i)$ is undefined then we say that worker thread k is *stalled* at step i. The total

[1] We refer to this as a schedule for consistency with previous work, even though it includes a storage allocation component.



Locations:

l_1	l_2	l_3	l_4
v_1	v_2	v_3	v_4
		v_5	

Figure 2.7: A valid 3-thread execution schedule for the computation from Fig. 2.6 using four locations $(l_1, \ldots, l_4)$

function ℓ determines which memory location each variable is stored in. We define the *live range*[2] of a variable v as the interval:

$$\text{live}_X(v) = [\min_{n \in \mathcal{A}_v} \omega^{-1}(n), \max_{n \in \mathcal{A}_v} \omega^{-1}(n)]$$

For an execution schedule to be *valid* the following condition must hold for all distinct variables $v, u \in V$:

$$\ell(v) = \ell(u) \implies \text{live}_X(v) \cap \text{live}_X(u) = \{\}$$

An example schedule for the computation from Fig. 2.6 is shown in Fig. 2.7.

2.4.3 Execution time and space

The *execution time* of a computation c $(= (N, E))$ under a p-thread execution schedule X, $T_{(c,p)}(X)$, is defined as the number of execution steps to execute all the computation's instructions:

$$T_{(c,p)}(X) = \max_{n \in N} \omega^{-1}(n)$$

When the choice of computation is obvious, we will often refer to this as simply $T_p(X)$. We also denote the minimum execution time of a computation c using p threads by $T_{(c,p)}$.

$$T_{(c,p)} = \min_{Y} T_{(c,p)}(Y)$$

Note that T_1 is equal to $|N|$, since a single worker thread can only execute one instruction per step. We use T_∞ to denote the length of the longest instruction path

[2]Our definition treats liveness as from first access to last access and hence live ranges reduce to a simple interval.

(called the *critical path*) in the computation, since even with arbitrarily many threads, each instruction on this path must execute sequentially. We call a p-thread schedule with an execution time of T_p *time optimal*. We call $\frac{T_1}{T_\infty}$ the *average parallelism* of the computation.

The *space* required by a computation c under a p-thread execution schedule X, $S_{(c,p)}(X)$, is defined as the number of locations used by the schedule (i.e. the size of the image of ℓ). We denote the minimum space needed to execute the computation on a single thread as:

$$S_{(c,1)} = \min_{Y} \; S_{(c,1)}(Y)$$

This is also the minimum space needed to execute the computation on any number of threads. We call a schedule that only requires S_1 execution space *space optimal*.

2.4.4 Scheduling algorithms and restrictions

The definitions of execution time and space given above allow us to compare different schedules. However, in reality these schedules must be computed (often during the execution of a computation) by some kind of algorithm. It is these algorithms that we are really interested in comparing. A *scheduling algorithm* is an algorithm that maps computations to valid execution schedules of those computations for any given number of threads. We represent these algorithms as functions $\mathcal{X}$ that map a number of threads p and a computation c to a p-thread execution schedule for c: $\mathcal{X}_p(c)$.

Most parallel languages place restrictions on which scheduling algorithms are allowed. We call these restrictions *scheduling restrictions*[3]. For instance, most parallel languages do not permit scheduling algorithms that depend on the "future" of the computation. In other words, the choice of which instructions should be executed at a particular time-step (and where variables first used at that time-step should be allocated) should not depend on instructions whose prerequisites are executed at a later time-step. This is a common restriction because, in practice, the nature of these "future" instructions may depend on the prerequisites that are yet to be executed.

2.4.5 Optimal scheduling algorithms

We denote the execution time of running a computation c on p threads using the scheduling algorithm $\mathcal{X}$ as $T_{\mathcal{X}}(c,p) = T_{(c,p)}(\mathcal{X}_p(c))$. Similarly, the execution space of running a computation c on p threads using the scheduling algorithm $\mathcal{X}$ is denoted $S_{\mathcal{X}}(c,p) = S_{(c,p)}(\mathcal{X}_p(c))$.

[3]Again, although we call these scheduling restrictions, they may also include restrictions in how variables are allocated to locations.

The definitions of execution time and space given above can be used to define preorders between scheduling algorithms pointwise (e.g. one algorithm is less than another if the schedules it produces use less space than the ones produced by the other algorithm for all computations on any number of threads). In general, with either the time or the space ordering, the set of all scheduling algorithms becomes a downward-directed set (i.e. all finite subsets have lower bounds). This means that we can produce *time-optimal* or *space-optimal* scheduling algorithms that have the best time or space performance across all computations.

However, in the presence of scheduling restrictions, the set of all allowed scheduling algorithms may not form a downward-directed set. This means that optimal scheduling algorithms that have the best time or space performance across all computations may not exist.

2.4.6 Efficient scheduling algorithms

We wish to define what it means for a schedule to be efficient. By efficient we mean that the effect on execution time or space of increasing the number of threads is in accordance with that expected by the programmer.

In the case of time we would like linear speedup for any number of threads, however this is not possible in general. So instead we say that a scheduling algorithm $\mathcal{X}$ is *time efficient* iff:

$$T_{\mathcal{X}}(c,p) = O(\frac{T_{\mathcal{X}}(c,1)}{p}) \qquad \text{whenever } p \leq \frac{T_1}{T_\infty}$$

In the case of space, we expect space to only increase linearly with the number of threads. We say that a scheduling algorithm $\mathcal{X}$ is *space efficient* iff:

$$S_{\mathcal{X}}(c,p) = O(p \times S_{\mathcal{X}}(c,1))$$

Note that these conditions only provide guarantees about how time and space will change with the number of threads. For good performance we also require guarantees about performance of the scheduling algorithm in the single-threaded case. For this reason we also define what it means for a scheduling algorithm to be *absolutely time (space) efficient*. We say that a scheduling algorithm is *absolutely time (space) efficient* iff it is time (space) efficient and all its single-threaded schedules are within a constant factor of optimal. Equivalently, a scheduling algorithm $\mathcal{X}$ is absolutely time efficient iff it obeys the following condition (a similar condition is obeyed by absolutely space-efficient scheduling algorithms):

$$T_{\mathcal{X}}(c,p) = O(\frac{T_{(c,1)}}{p}) \qquad \text{whenever } p \leq \frac{T_1}{T_\infty}$$

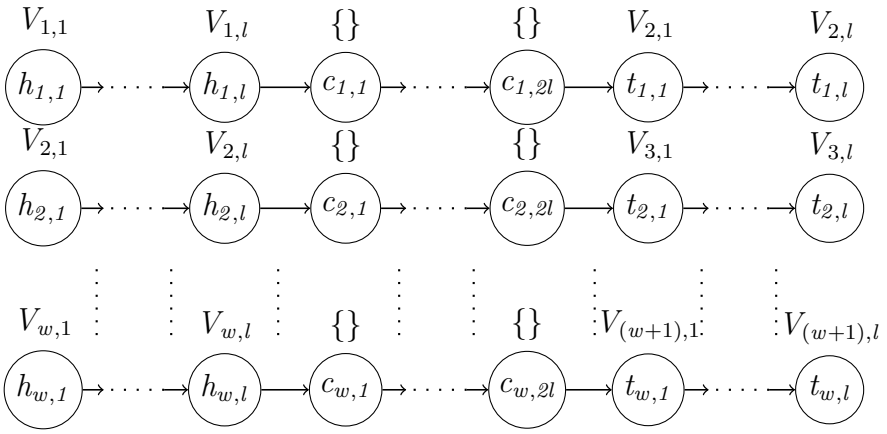


Figure 2.8: Inefficient Computation $\mathfrak{C}(w, l, s)$ (where $V_{x,y} = \{v_{x,y,1}, \ldots, v_{x,y,s}\}$)

Early work on scheduling directed acyclic graphs by Brent [9] and Graham [30] shows that this condition holds for any time-optimal scheduling algorithm. Eager et al. [24] and Blumofe et al. [7] show that any greedy scheduling algorithm (i.e. an algorithm that only allows a thread to stall if there are no ready instructions) also obeys this condition.

Any space-optimal schedule only requires S_1 space, so any space-optimal scheduling algorithm is obviously absolutely space efficient.

Note that if there are no scheduling restrictions, then it is always possible to create both absolutely time-efficient scheduling algorithms and absolutely space-efficient scheduling algorithms. However, it is not possible to create scheduling algorithms that are simultaneously both absolutely space efficient and absolutely time efficient.

2.4.7 Inherently inefficient example

It is not possible to create scheduling algorithms that are simultaneously both absolutely space efficient and absolutely time efficient for all possible computations. This can be demonstrated by showing that for certain classes of computation there are no schedules which meet the conditions for both absolute time efficiency and absolute space efficiency when $p > 1$.

Consider the class of computation $\mathfrak{C}(w, l, s)$ (where $w > 1$, $l > 1$ and $s > 0$) shown in Fig. 2.8. These computations access $(w + 1)ls$ variables $(v_{(1,1,1)}, \ldots, v_{(w+1,l,s)})$ and consist of w "chains", each a linear chain of $4l$ instructions divided into three sections:

1. A header of l instructions $h_{x,1}, \ldots, h_{x,l}$. Each header instruction $h_{x,y}$ accesses variables $v_{(x,y,1)}$ through $v_{(x,y,s)}$.

2. A core of $2l$ instructions $c_{x,1}, \ldots, c_{x,2l}$, which access no variables.

3. A tail of l instructions $t_{x,1}, \ldots, t_{x,l}$. Each tail instruction $t_{x,y}$ accesses variables $v_{(x+1,y,1)}$ through $v_{(x+1,y,s)}$.

For any computation $\mathfrak{C}(w, l, s)$ it can be shown that $S_1 = s$. This can be achieved by a schedule that executes the core of each chain in order and interleaves the headers and tails of adjacent chains, so that $t_{x,y}$ is executed immediately before $h_{x+1,y}$.

It can be shown (see Appendix A) for any p-thread schedule X of the computation $\mathfrak{C}(w, l, s)$ where $p > 1$ that:

$$T_p(X) = O(\frac{T_1}{p}) \implies S_p(X) = \Omega(l \cdot s \cdot p)$$

This means that the space required for an efficient schedule cannot be bounded by a function of s and p alone.

Informally, the reason for this bound is that any absolutely time-efficient schedule must execute multiple chains at once. This means that the headers of some chains must execute before the tails of their preceding chains, causing the lifetimes of all the variables accessed by these headers to overlap.

2.5 Logic programming

Chapter 6 describes an analysis of OpenMP programs' task-local memory usage. We develop an intuitive and novel presentation of this analysis using a generalisation of logic programming. This section describes logic programming and the generalisation used in presenting our analysis.

2.5.1 Logic programming

Logic programming is a paradigm where computation arises from proof search in a logic according to a fixed, predictable strategy. It arose with the creation of Prolog [43]. In a traditional logic programming system, a logic program represents a deductive knowledge base, which the system will answer queries about.

Syntax

A (traditional) *logic program* P is a set of rules of the form

$$A \longleftarrow B_1, \ldots, B_k$$

where $A, B_1, \ldots, B_k$ are *atoms*.

An *atom* is a formula of the form $F(t_1, \ldots, t_k)$ where F is a *predicate symbol* and $t_1, \ldots, t_k$ are *terms*. A is called the *head* and $B_1, \ldots, B_k$ the *body* of the rule. Logic programming languages differ according to the forms of terms allowed. We give a general explanation below, but our applications will only consider Datalog-style terms consisting of variables and constants. A logic program defines a model in which *queries* (syntactically

bodies of rules) may be evaluated. We write $ground(P)$ for the ground instances of rules in P.

Note that we do not require P to be finite. Indeed the program analyses we propose in Chapter 6 naturally give infinite such P, but Section 6.6 shows these to have an equivalent finite form.

Interpretations and models and immediate consequence operator

To *evaluate* a query with respect to a logic program we use some form of reduction process (SLD-resolution for Prolog, bottom-up model calculation for Datalog), but the *semantics* is simplest expressed model-theoretically. We present the theory for a general complete lattice $(\mathfrak{L}, \sqsubseteq)$ of truth values (the traditional theory uses $\{false \sqsubseteq true\}$). We use $\sqcup$ to represent the join operator of this lattice and $\sqcap$ to represent the meet operator of this lattice. Members of $\mathfrak{L}$ may appear as nullary atoms in a program.

Given a logic program P, its *Herbrand base* $\mathcal{HB}_P$ is the set of ground atoms that can be constructed from the predicate symbols and function symbols that appear in P. A *Herbrand interpretation* I for a logic program P is a mapping of $\mathcal{HB}_P$ to $\mathfrak{L}$; interpretations are ordered pointwise by $\sqsubseteq$.

Given a ground rule $r = (A \longleftarrow B_1, \ldots, B_k)$, we say a Herbrand interpretation I *respects* rule r, written $I \models r$, if $I(B_1) \sqcap \cdots \sqcap I(B_k) \sqsubseteq I(A)$.

A Herbrand interpretation I of P is a *Herbrand model* iff $I \models r \quad (\forall r \in ground(P))$. The least such model (which always exists for the rule-form above) is the canonical representation of a logic program's semantics.

Given logic program P we define the *immediate consequence operator* T_P from Herbrand interpretations to Herbrand interpretations as:

$$\big(T_P(I)\big)(A) = \bigsqcup_{(A \longleftarrow B_1, \ldots, B_k) \in ground(P)} I(B_1) \sqcap \cdots \sqcap I(B_k)$$

Note that I is a model of P iff it is a pre-fixed point of T_P (i.e. $T_P(I) \sqsubseteq I$). Further, since the T_P function is monotonic (i.e. $I_1 \sqsubseteq I_2 \Rightarrow T_P(I_1) \sqsubseteq T_P(I_2)$), it has a least fixed point, which is the least model of P.

2.5.2 Negation and its semantics

It is natural to consider extending logic programs with some notion of negation. This leads to the idea of a *general logic program* which has rules of the form $A \longleftarrow L_1, \ldots, L_k$ where L is a *literal*. A literal is either an atom (*positive literal*) or the negation of an atom (*negative literal*).

The immediate consequence operator of a general logic program is not guaranteed to be monotonic. This means that it may not have a least fixed point, so that the canonical

model of logic programs cannot be used as the canonical model of general logic programs. It is also one of the strengths of adding negative literals: support for non-monotonic reasoning.

Non-monotonic reasoning grew out of attempts to capture the essential aspects of common sense reasoning. It resulted in a number of important formalisms, the most well known being:

- The circumscription method of McCarthy [52], which attempts to formalise the common sense assumption that things are as expected unless otherwise specified.
- Reiter's Default Logic [65], which allows reasoning with default assumptions.
- Moore's Autoepistemic Logic [54], which allows reasoning with knowledge about knowledge.

A classic example of non-monotonic reasoning is the following:

$$\begin{aligned}
&\texttt{fly}(X) \longleftarrow \texttt{bird}(X), \neg\texttt{penguin}(X)\\
&\texttt{bird}(X) \longleftarrow \texttt{penguin}(X)\\
&\texttt{bird}(\texttt{tweety}) \longleftarrow\\
&\texttt{penguin}(\texttt{skippy}) \longleftarrow
\end{aligned}$$

It seems obvious that the "intended" model of the above logic program is:

$$\{\texttt{bird}(\texttt{tweety}),\ \texttt{fly}(\texttt{tweety}),\ \texttt{penguin}(\texttt{skippy}),\ \texttt{bird}(\texttt{skippy})\}$$

Two approaches to defining such a model are to *stratify programs* and to use *stable models*.

Stratified programs

One approach to defining a standard model for general logic programs is to restrict our attention to those programs that can be *stratified*.

A predicate symbol F is *used* by a rule if it appears within a literal in the body of a rule. If all the literals that it appears within are positive then the use is positive, otherwise the use is negative. A predicate symbol F is *defined* by a rule if it appears within the head of that rule.

A general logic program P is *stratified* if it can be partitioned $P_1 \cup \cdots \cup P_k = P$ so that, for every predicate symbol F, if F is defined in P_i and used in P_j then $i \leq j$, and additionally $i < j$ if the use is negative.

Any such stratification gives the *standard model*[4] of P as M_k below:

$$\begin{aligned} M_1 &= \text{ The least fixed point of } T_{P_1} \\ M_i &= \text{ The least fixed point of } \lambda I.\ \left(T_{P_i}(I) \sqcup M_{i-1}\right) \end{aligned}$$

Stable models

Stable models (Gelfond et al. [28]) give a more general definition of standard model using *reducts*. For any general logic program P and Herbrand interpretation I, the *reduct* of P with respect to I is a logic program defined as:

$$\mathcal{R}_P(I) = \{\ A \longleftarrow red_I(L_1), \ldots, red_I(L_k) \mid\ (A \longleftarrow L_1, \ldots, L_k) \in ground(P)\ \}$$

$$\text{where } red_I(L) = \begin{cases} L & \text{if } L \text{ is positive} \\ \hat{I}(L) & \text{if } L \text{ is negative} \end{cases}$$

where $\hat{I}$ is the natural extension of I to ground literals.

A *stable model* of a program P is any interpretation I that is the least model of its own reduct $\mathcal{R}_P(I)$.

Unlike the standard models of the previous sections, a general logic program may have multiple stable models or none. For example, both $\{p\}$ and $\{q\}$ are stable models of the general logic program having two rules: $(p \longleftarrow \neg q)$ and $(q \longleftarrow \neg p)$. A stratified program has a unique stable model.

The stable model semantics for negation does not fit into the standard paradigm of logic programming. Traditional logic programming hopes to assign to each program a single "intended" model, whereas stable model semantics assigns to each program a (possibly empty) set of models. However, the stable model semantics can be used for a different logic programming paradigm: *answer set programming*. Answer set programming treats logic programs as a system of constraints and computes the stable models as the solutions to those constraints. Note that finding all stable models needs a backtracking search rather than the traditional bottom-up model calculation in Datalog.

2.5.3 Implication algebra programming

In Chapter 6 we use logic programs to represent stack-size constraints using a multi-valued logic. To represent operations like addition on these sizes it is convenient to allow operators other than negation in literals—a form of *implication algebra* (due to Damasio et al. [18])—to give *implication programs*.

[4] Apt et al. [3] show that this standard model does not depend on which stratification of P is used.

Literals are now terms of an algebra $\mathfrak{A}$. A *positive* literal is one where the formula corresponds to a function that is monotonic (order preserving) in the atoms that it contains. Similarly, *negative* literals correspond to functions that are anti-monotonic (order reversing) in the atoms they contain. We do not consider operators which are neither negative nor positive (such as subtraction).

Implication programs and their models

An *implication program* P is a set of *rules* of the form $A \longleftarrow L_1, \ldots, L_k$ where A is an atom, and $L_1, \ldots, L_k$ are positive literals.

Given an implication program P, we extend the notion of *Herbrand base* $\mathcal{HB}_P$ from the set of atoms to the set, $\mathcal{HL}_P$, of all ground literals that can be formed from the atoms in $\mathcal{HB}_P$. A *Herbrand interpretation* for an implication program P is a mapping $I : \mathcal{HB}_P \to \mathfrak{L}$ which extends to a valuation function $\hat{I} : \mathcal{HL}_P \to \mathfrak{L}$.

Given rule $r = (A \longleftarrow L_1, \ldots, L_k)$, now a Herbrand interpretation I *respects* rule r, written $I \models r$, if $\hat{I}(L_1) \sqcap \cdots \sqcap \hat{I}(L_k) \sqsubseteq I(A)$. Definitions of Herbrand model, immediate consequence operator etc. are unchanged.

General implication programs and their models

General implication programs extend implication programs by also allowing negative literals. The concepts of stratified programs and stable models defined in Section 2.5.2 apply to general implication programs exactly as they do to general logic programs.

Chapter 3

EMCC: An extensible C compiler

This chapter describes the design and implementation of the EMCC compiler, a C compiler which supports extensions to its front- and middle-ends.

Most previous work on extensibility in compilers has focused on extensions to the front-end of the compiler. Extensions are added to the AST of the language, this extended AST is then lowered into the original AST in the front-end before being passed on to the middle-end. This means that all analyses and optimisations used to implement the extension must be carried out on the AST. However, ASTs are not a suitable representation for many optimisations. They contain syntactic artefacts which can add a lot of unnecessary complexity to an analysis' implementation.

The problem with supporting language extensions exclusively through extensions to the front-end is that the analyses (and optimisations) applied to the extension are completely separate from those applied to the base language. This means that if an optimisation of the extension requires an existing analysis of the base language, that analysis must be reimplemented so that it can be applied in the front-end. More importantly, it means that there is no "semantic integration" of the extension with the rest of the language. The analyses and optimisations of the base language remain completely oblivious to the extension and any semantic information it may contain. If our extensions are to have parity with the rest of the language, we must be able to extend the middle-end of the compiler.

Extending the middle-end of a compiler, requires us to be able to extend the compiler's Intermediate Languages (ILs). In traditional compilers this is difficult because the front-ends, back-ends and optimisations of the compiler are all implemented directly in terms of the ILs. There is no abstraction of the ILs to allow a front-end to target multiple ILs, or allow an optimisation to operate on any IL that provides the right operations.

Difficulty in extending the middle-end can cause people to try to encode their extensions within the existing IL inappropriately. For example, for years libraries like PThreads[57] were used to add atomic operations to C/C++. Rather than add atomic

constructs to the IL, this encoded the atomic operations within the existing IL in the form of function calls. The safety of these libraries relied on assumptions about how optimisations would treat calls to unknown procedures. However, these assumptions turned out to be incorrect within GCC resulting in bugs in code using those libraries, as discussed by Boehm[8].

Difficulty in extending the middle-end can also prevent people using efficient techniques not supported within the existing IL. For example, language designers targeting the LLVM framework have requested improved support for precise garbage collection in LLVM, notably the Rust project[55] from Mozilla. While the LLVM IL has support for precise garbage collection, it forces all garbage collection roots to be spilled from registers when a collection may occur. This greatly reduces the efficiency of languages using such collectors. Adding support for GC roots in registers would require additional instructions in the IL and additional restrictions about how other instructions could be manipulated. Adding these features would require all optimisations in LLVM to be checked to ensure these restrictions were obeyed. The amount of work required to make these changes has meant that Rust has continued to use a much less efficient conservative garbage collector, despite wanting to change to a precise collector for multiple years.

The rest of this chapter discusses the design and implementation of the EMCC compiler, a C compiler which supports extensions to its front- and middle-ends. Section 3.1 discusses related work. Section 3.2 gives a general overview of the design of EMCC. Section 3.3 describes some design patterns to support extensibility in EMCC. Section 3.4 describes how the design of EMCC allows the front-end to be extended. Section 3.5 describes how the design of EMCC allows the middle-end to be extended.

3.1 Related Work

In this section we first discuss the extensibility of mainstream compilers, and then discuss related work on designing extensible compilers and languages.

3.1.1 Mainstream compilers

The two most widely used open source compilers are GCC [68] and LLVM [45].

GCC

GCC has a traditional compiler design, with front-ends for multiple languages all targeting an Intermediate Language, called GIMPLE. This IL is then optimised before being lowered to another IL, called RTL. The RTL representation is then passed to target-specific code-generators. Each instruction in RTL carries with it the corresponding target-specific

assembly instruction, so the lowering from GIMPLE to RTL is parameterised by the target, while the RTL optimisations are still independent of the target.

The front-ends of GCC have hand-written lexers and parsers, and are not particularly easy to extend. However, for C and C++ GCC has its own language extension (since adopted by many other compilers) called attributes. These allow arbitrary expressions to be attached at various places in the AST, in order to pass additional information to the compiler. For example, the `deprecated` attribute can be attached to function definitions:

```
int old_fn() __attribute__((deprecated));
```

and the compiler will emit a warning if the function is used. GCC propagates these attributes through to the GIMPLE IL, so they can be used by optimisations.

There is support for adding new kinds of simple expression to GIMPLE. However other changes are much more difficult, requiring changes to all GIMPLE optimisations as well as the target-specific lowering from GIMPLE to RTL. Similarly, extensions to RTL require changes to all RTL optimisations. Adding new optimisations based on either GIMPLE to RTL is well supported: there is even a plugin mechanism.

LLVM

LLVM is framework consisting of an IL and a collection of optimisations and back-ends for that IL. It was designed to support link-time optimisation and JIT compilation by using its IL as a form of byte code.

Clang is a C/C++ front-end that targets the LLVM IL and is closely associated with the LLVM project. It has a hand-written lexer and parser, and its semantic analysis is tightly coupled with its parser. This makes it very difficult to extend the parser. However, like GCC, it has good support for creating new attributes, which can be used for language extensions.

LLVM is designed to make it easy for people to build new compilers and optimisations, by standardising on a single simple IL. However, this makes it very difficult to extend the IL. Not only the optimisations in LLVM, but also all the code written by third-parties using LLVM, are based directly on the IL. This means that changes to the IL cause incompatibilities with code based on previous versions of LLVM, and are avoided if at all possible. To mitigate this, there is some support for added new intrinsic functions by providing a basic description of their effects, but this system is very limited.

3.1.2 Extensible compilers

There is much literature on creating extensible compilers, especially to support extension of the compiler by its users. This work is mostly focused on extending the front-end of compilers by extending the syntactic and semantic analysis phases of the compiler.

$\mathbf{Expr}_1 \rightarrow \mathbf{Expr}_2 + \mathbf{Term}$	$[\mathbf{Expr}_1.\texttt{value} = \mathbf{Expr}_2.\texttt{value} + \mathbf{Term}.\texttt{value}]$
$\mathbf{Expr} \rightarrow \mathbf{Term}$	$[\mathbf{Expr}.\texttt{value} = \mathbf{Term}.\texttt{value}]$
$\mathbf{Term}_1 \rightarrow \mathbf{Term}_2 * \mathbf{Factor}$	$[\mathbf{Term}_1.\texttt{value} = \mathbf{Term}_2.\texttt{value} * \mathbf{Factor}.\texttt{value}]$
$\mathbf{Term} \rightarrow \mathbf{Factor}$	$[\mathbf{Term}.\texttt{value} = \mathbf{Factor}.\texttt{value}]$
$\mathbf{Factor} \rightarrow$ "("$\mathbf{Expr}$")"	$[\mathbf{Factor}.\texttt{value} = \mathbf{Expr}.\texttt{value}]$
$\mathbf{Factor} \rightarrow$ `integer`	$[\mathbf{Factor}.\texttt{value} = \texttt{strToInt}(\texttt{integer.str})]$

Figure 3.1: A simple Attribute Grammar

Extensible syntax

There has been a lot of work on allowing the syntax of languages to be extended. This has mostly focused on various forms of extensible grammar. Examples include the extensible PEG grammar in Xtc [31], the extensible LL grammar of Camlp4 [20], the extensible GLR grammar in Xoc [17] and the extensible LALR grammar in Polyglot [59].

Extensible semantic analysis and translation

Most work on extending the semantic analyses of a compiler is related to the notion of *Attribute Grammars* [41]. Attribute grammars define values (called *attributes*) associated with nodes on a tree – for compilers the AST. These attributes are defined in terms of production rules which describe how the attribute is computed in terms of other attributes.

For example, expression nodes may have a `type` attribute, and the `type` attribute of a addition expression (e.g. `x + y`) might be `int` or `float` depending on the `type` attributes of its operands. Attributes, like `type`, which are computed and passed up the tree are called *synthesised* attributes, whilst attributes that are passed down the tree (for example the typing environment) are called *inherited* attributes. A simple example of an attribute grammar is shown in Fig. 3.1.

A good example of a system using attribute grammars is the JustAdd framework [32]. JustAdd is a framework for creating compilers with extensible front-ends. It provides a library for defining attribute grammars over an AST and for creating AST rewriters based on those attributes. The semantic analysis of the compiler can be defined using these grammars, and extensions to that analysis can be made by adding additional attributes to the grammar. The AST rewriters can be then used to remove the extension's nodes from the AST before it is passed to the middle-end.

Other systems have used notions very similar to attribute grammars. The Xoc [17]

compiler uses *lazy attributes* which are attributes attached to AST nodes which are computed on demand. The Polyglot [59] compiler treats AST nodes as extensible objects and creates fields to represent the semantic analyses of the compiler. These fields are very similar to attributes, and creating these objects using mixin inheritance is similar to how extensions are added to an attribute grammar.

Another interesting example is MPS [36]. MPS is a *structured editor* that supports creating language extensions using a system related to attribute grammars. Structured editors work directly on the AST of a language, rather than editing text and then parsing it. MPS allows users to define language extensions with a system similar to an attribute grammar specification, and these extensions can then be used within the editor. This system avoids the need to fully support extensible syntax.

Extensible middle-ends

One compiler which has support for extending its middle-end is the CoSy [2] compiler framework. Optimisations and analyses in CoSy are performed by *engines*. These engines must declare how they are going to access the IL. For example, they must declare which kinds of nodes they may write to. This system was designed in order to allow these engines to be safely run in parallel, or even speculatively, but it also makes it much easier to extend the IL since you can easily determine which optimisations might be affected by the change.

Another example is the SUIF [73] compiler. SUIF allows new IL nodes to be created by subclassing from existing nodes. There are also abstract node classes which create a predefined hierarchy of abstractions, allowing some analyses and optimisations to work on ILs including new nodes.

3.1.3 Extensible languages

There has been a large body of work around the creation of *extensible languages*. These are languages which allow users to add new features to them. Zingaro [77] gives a good summary of some of this work. The mechanisms for extensibility are often macro systems, for example hygienic macros in Lisp [42] or the macro system in Scala [11]. Another example, is the Delite framework [10], built on top of the Scala macro system, which supports creating language extensions to improve performance using heterogeneous parallelism, including extensible transformations on an "IL" which is then translated back into Scala. XLR [19] is an extensible language that allows users to specify AST rewrite rules to implement new features. The Seed7 [53] language supports extensibility through an extensible syntax and call-by-name functions.

3.2 Design overview

EMCC is implemented using the OCaml programming language (see Section 2.1). OCaml's excellent support for complex data structures makes it ideal for implementing a compiler, and its module and object systems are both very useful for allowing extensibility.

EMCC is really a library for creating compilers combined with a "driver" program. The library can either be used directly to create a new compiler that includes some language extensions, or the extensions can be wrapped into a plug-in and executed by the driver program.

Fig. 3.2 shows the main components of the EMCC library.

`Cabs` contains a definition of our Abstract Syntax Tree. This is a set of mutually recursive datatypes which describe the syntax of preprocessed C programs.

`Lexer` is a lexer which converts a preprocessed C source file into a stream of tokens. It is implemented using the `ocamllex` tool which comes with the OCaml distribution. `ocamllex` is a traditional lexer generator, based on Lex [49], which generates a lexer based on a specification made up of regular expressions.

`Parser` is a parser which converts a stream of tokens from `Lexer` into the AST defined by `Cabs`. It is implemented using the Menhir [63] parser generator. Menhir is a traditional LR parser generator, like Yacc [38] or Bison [22], which generates a parser based on a context-free grammar.

`Translate` converts the AST defined in `Cabs` into an IL. This translation is built using a functor that accepts a module describing how to create a given IL (see Section 3.4.2). The translation also checks the C program for errors.

`Generate` provides the main back-end for EMCC; it converts an IL back into the AST defined in `Cabs` so that it can be printed as C source code. This source code can then be run through a standard C compiler to produce the final output. Like `Translate`, this conversion is defined as a functor that accepts a module describing how a given IL can be lowered to the `Cabs` AST for output.

`IL` defines some basic facilities for creating and manipulating ILs.

`CIL` defines a default IL including a module that describes how this IL can be created using `Translate` and a module that describes how this IL can be lowered using `Generate`.

`Properties` provides support for *properties* (see Section 3.3.1).

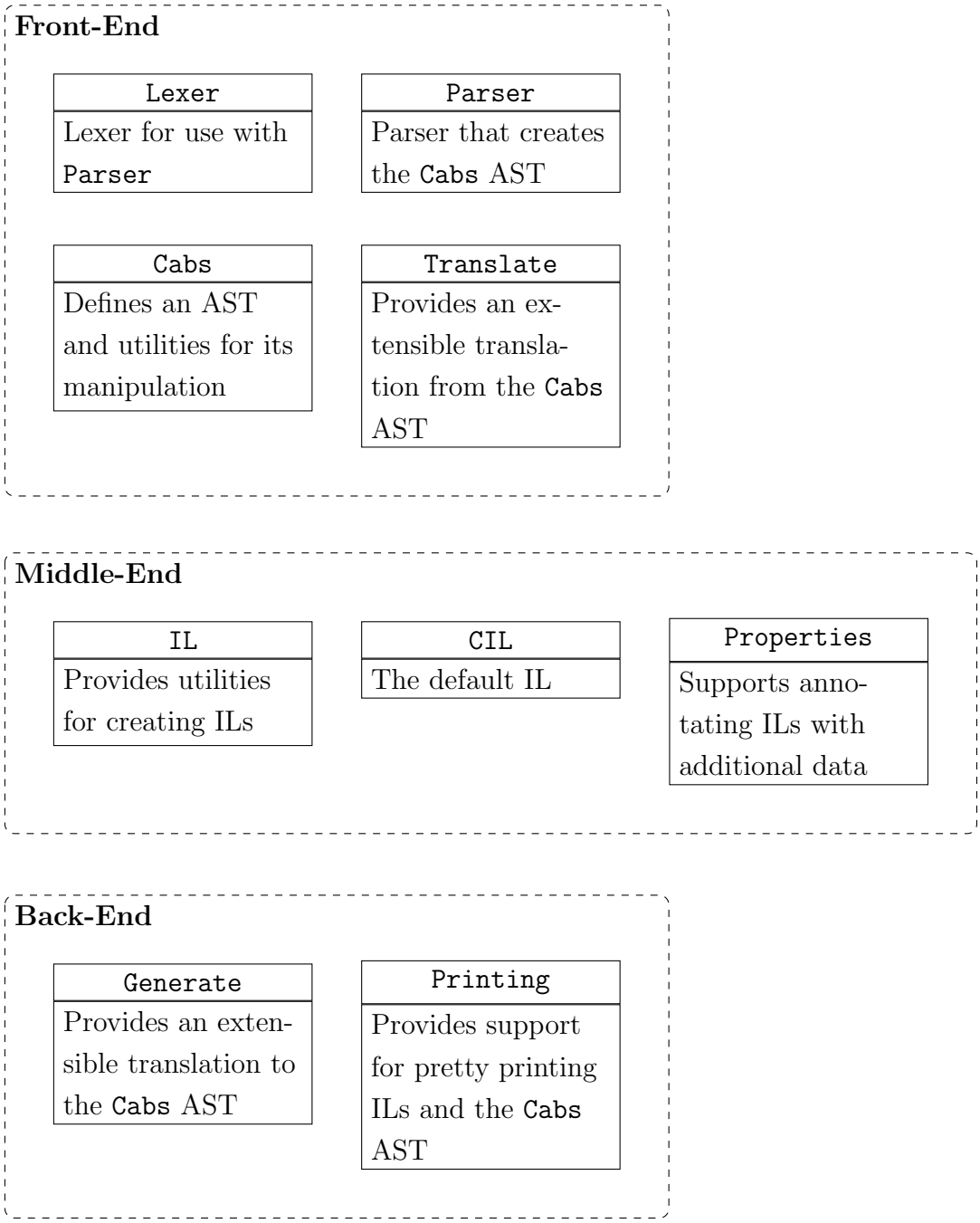


Figure 3.2: Components of EMCC

```
type 'a prop_class
type property
val definePropClass: unit -> 'a prop_class
val createProperty: 'a prop_class -> 'a -> property
val matchProperty: 'a prop_class -> property -> 'a option
val findProperty: 'a prop_class -> property list -> 'a
val addProperty: property -> property list -> property list
val removeProperty: 'a prop_class -> property list -> property list
```

Figure 3.3: The signature of the `Properties` module

We developed EMCC starting from the C Intermediate Language [56] project, which is a C front-end. Most of the modules have been completely rewritten since then, but the `Lexer`, `Parser` and `Cabs` modules have remained mostly unchanged.

Note that the current version of EMCC only supports a back-end that generates low-level C code. This means that extensions to the back-end are really made by extending the back-end of another C compiler and then using that to compile the output C code. This design should not be confused with preprocessors which transform extensions down to their base language. We use C as a portable assembly language and expect all machine-independent optimisations to be performed by our compiler rather than during the final translation of the output C. We intend to support more traditional back-ends in the future.

3.3 Patterns for extensibility

We use a number of design patterns to support extensibility in EMCC. This section describes two of these patterns: *properties* and *visitors*.

3.3.1 Properties

Sometimes we only need to add a small extension to an existing IL. For these cases EMCC supports *properties*. Properties are an extensible sum type that allows nodes of the ILs to be tagged with arbitrarily typed data. The signature of the `Properties` module is shown in Fig. 3.3.

Properties can be used to attach additional data to an existing data type. Fig. 3.4 shows a simple example which uses properties to add an `addr_taken` field to the `variable` data type.

In OCaml properties can be implemented using hash tables or extensible variant types, which were added to OCaml in version 4.02 by the author.

```
open Properties
type variable =
  { ...
    mutable properties: property list;
    ... }

let addr_taken : bool prop_class = definePropClass ()

let set_addr_taken v b =
  let props = removeProperty addr_taken v.properties in
  let p = createProperty addr_taken b in
  let props = addProperty p props in
    v.properties <- props

let get_addr_taken v =
  findProperty addr_taken v.properties
```

Figure 3.4: An example using properties

Properties are basically an instance of an *extensible algebraic datatype*, which have been proposed before as a useful tool for implementing extensible compilers by Zeneger et al. [75].

3.3.2 Visitors

Many analyses and optimisations involve traversing a tree (or graph) of IL nodes. For example, marking all the local variables used in a function requires us to traverse the IL nodes which represent the function to find all the variable usages. For extensibility, we would to express these optimisations in a way that does not rely on the underlying structure of the nodes. We can use *visitors* to make this easier. A visitor is a class that encodes the recursive tree structure of a set of data types—in our case IL nodes. The class contains a method for each of the data types will recursively call the other methods on all of its child nodes.

Fig. 3.5 shows an example of a visitor class for a small set of recursive data types. Fig. 3.6 uses the visitor from Fig. 3.5 to create a function that counts the number of minus operations in an expression. This function is implemented by creating a class that inherits from `visitor` and overrides its `op` method to increment a count. Note how the new `op` method calls the inherited `op` method (using the special `super` variable) to recursively visit its child nodes.

```
type var = string
and expr =
  Const of int
| Var of var
| Op of op
and op =
  Plus of expr * expr
| Minus of expr * expr

class visitor = object (self)
  method var v = ()
  method expr = function
    Const _ -> ()
  | Var v -> self#var v
  | Op o -> self#op o
  method op = function
    Plus(e1, e2) -> self#expr e1; self#expr e2
  | Minus(e1, e2) -> self#expr e1; self#expr e2
end
```

Figure 3.5: An example of a visitor class

```
class count_minuses = object
  val mutable count = 0
  method count = count
  inherit visitor as super
  method op o =
    super#op o;
    match o with
      Minus _ -> count <- count + 1
    | _ -> ()
end

let minuses e =
  let v = new count_minuses in
    v#expr e;
    v#count
```

Figure 3.6: An example using the visitor from Fig. 3.5

3.4 Extensible front-end

3.4.1 Extensible syntax

Extensible front-ends must include some method of extending the syntax of the language. However, extending grammars is complex and the extensions do not compose easily. Instead EMCC uses a more flexible version of the attributes, pragmas and built-in functions used by GCC and LLVM to support syntax extensions. EMCC treats these constructs as *quotations*. Quotations are syntactic constructs which are not parsed with the rest of the source code. Quotations are treated normally by the lexer, however the parser simply leaves them as sequences of lexical tokens. This allows them to be handled by extensions during the translation phase.

EMCC supports three types of *quotation*:

Pragmas are compiler directives of the form:

```
#pragma name arguments
```

A pragma's arguments are terminated by a newline. Pragmas can appear anywhere in a program where a statement or declaration would be allowed. Within a function body a pragma is attached to the statement that it precedes.

Attributes are attached to functions, variables and types during declarations. For example:

```
extern void foobar (void) __attribute__((section ("bar")));
```

An attribute's arguments must have balanced parentheses and quotation marks. They can appear at any point within a declaration, and various rules determine where in the AST they are attached.

Built-in Functions look like ordinary functions whose names start with `__builtin_`. For example:

```
__builtin_types_compatible_p(typeof (x), long double)
```

A built-in function's arguments must have balanced parentheses and quotation marks. They are treated just like any other kind of expression.

During translation (in the `Translate` module), the quotations' bodies can be parsed using simple stream parsers. OCaml provides good support for creating these and some useful ones (e.g. one that parses the same expression syntax as `Parser`) are included in EMCC.

3.4.2 Extensible semantic analysis and translation

The semantic analysis of the AST and its translation into an IL are handled in EMCC by the `Translate` module. This analysis and translation is implemented using a mechanism similar to attribute grammars. The `Translate` module can be thought of as two components: a set of recursive functions which apply a grammar to the AST, and a grammar which implements the semantic analysis required to check the correctness of a C program. The translation is used by creating a grammar which inherits from the one in `Translate` and using the functions in `Translate` to apply it to an AST. The representation of an AST node in the target IL is represented as an attribute in these grammars.

These grammars are implemented as a set of functions which take an *environment* object and objects representing the node's children and produce an object representing the node, and possibly a modified environment object. The inherited attributes of the grammar are represented by the methods of the environment objects, whilst the synthesised attributes are represented by the methods of the node objects. Note that, in these grammars the passing around of the environment object is hard-coded so there are limits to how inherited attributes can be used.

These grammars are wrapped in a module and passed to a functor in the `Translate` module which generates the functions that will apply the grammar to an AST. The semantic analysis grammar in `Translate` is actually a set of classes, rather than a set of functions, so that they can be inherited by other grammars.

There are two kinds of extensions to this translation that we wish to support: new AST nodes and new attributes. Fig. 3.7 shows part of a simple language extension to add a built-in function that evaluates to my name. It also adds a new attribute on expressions `is_name`, which is true for the expression node created for the new name built-in.

This example includes an `is_name` method in its expression class, which inherits from the expression class in the `Translate.Grammar` module. It also defines a `name_builtin` class which inherits from the `string_literal` class specialised by my name and overriding its `is_name` method to be true. The extension would also need to register the name of the built-in function and register `name_builtin` as the function for handling it.

3.5 Extensible middle-end

3.5.1 Modular interfaces

In order to allow the compiler's middle-end to be extended, it is important to have a modular design with clear interfaces between the middle-end and each of the front-end, back-end and optimisations. These interfaces must also themselves be extensible, to allow extensions to the middle-end to be consumed by back-ends and produced by front-ends.

```
class type expr =
  object
    inherit Translate.Grammar.expression
    ...
    method is_name : bool
  end

class string_literal (env : environment) (s : string) : expr =
  object
    inherit Translate.Grammar.string_literal env
    ...
    method is_name = false
  end

class name_builtin (env : environment) : expr =
  object
    inherit string_literal env "Leo White"
    method is_name = true
  end

let name_builtin e = new name_builtin env
```

Figure 3.7: Example of a language extension

In EMCC we achieve this modularity by using OCaml functors throughout the design. The front-end, back-end and all analyses and optimisations are implemented as functors. The input to these functors describes what must be exposed by a middle-end in order to use these components. This allows the middle-end to be anything that can provide the required interfaces for translation and generation.

The extensibility in these interfaces is achieved using techniques like property lists and visitors (see Section 3.3). Fig. 3.8 shows a simple analysis which uses a visitor class to determine whether a local variable has its address referenced.

The `ADDRESSABLE` signature defines the interface required by the analysis. The `AddressTaken` implements the analysis parameterised by a module `M` which implements `ADDRESSABLE`. Note that the methods of the visitor class must be private in order to be used with these functors because non-private methods cannot be ignored. The `visitor` class of the `AddressTaken` module overrides the `expr` method with one which checks if an expression takes the address of a given variable. Then the `isAddressTaken` function instantiates this class and uses its `func` method to check all the expressions within a function definition.

3.5.2 CIL: The default IL

The library also provides a default IL called CIL. CIL resembles a very simple subset of C, and is derived from the C Intermediate Language [56][1].

CIL is actually made up of two separate ILs: the expression language and the control-flow language. The expression language includes side-effect-free expressions, variables and types. The control-flow language is used to describe function definitions, including statements that change the state of variables. This division makes it easy to change the control flow representation without having to change the expression representation, and vice-versa.

We provide two versions of the control-flow language: one that represents function bodies as a control-flow graph, and one that represents them as a tree of simple control structures.

The CIL expression language has two related type systems. Each expression and variable has both a *type* which describes which operations are valid on that expression or variable, and a *representation* which describes how the value of that expression or variable is represented in memory.

The types are used to ensure that invalid expressions cannot be created. This is done by encoding the types of the IL within OCaml's own type system using Generalised Algebraic Datatypes (GADTs).

Fig. 3.9 shows an example of how GADTs are used in CIL's definition. This example illustrates how the type parameter of the `expr` type is used to ensure that the `PlusII`

[1]Not to be confused with Microsoft's Common Intermediate Language used in their .NET framework

```
module type ADDRESSABLE = sig
  type func
  type expr
  type lval
  type variable
  val isAddress: expr -> bool
  val getAddress: expr -> lval
  val isVariable: lval -> bool
  val getVariable: lval -> variable
  val equalVariable: variable -> variable -> bool
  class visitor : object
    method private expr: expr -> unit
    method private func: func -> unit
  end
end
module AddressTaken (M : ADDRESSABLE) = struct
  class visitor (v: M.variable) : object
    method result : bool
    method func : M.func -> unit
  end = object
    inherit M.visitor
    val mutable result = false
    method result = result
    method! private expr (e: M.expr) =
      if (M.isAddress e) then
        let lv = M.getAddress e in
        if (M.isVariable lv) then
            let v' = M.getVariable lv in
            if (M.equalVariable v v') then
              result <- true
  end
  let isAddressTaken (f: M.func) (v: M.variable) =
    let vis = new visitor v in
      vis#func f;
      vis#result
end
```

Figure 3.8: An example of a modular analysis

```
type typ = [ `Integer | `Float | ··· ]

type 'ty constant =
| CInt : int_lit -> [< typ > `Integer] constant
| CFloat : float_lit -> [< typ > `Float] constant
    ⋮

type 'ty expr =
    ⋮
| Const : 'ty constant -> 'ty expr
    ⋮
| BinOp : ('op1,'op2,'ty) binop * 'op1 expr * 'op2 expr -> 'ty expr
    ⋮

type ('op1,'op2,'result) binop =
    ⋮
| PlusII : ([`Integer], [`Integer], [< typ > `Integer]) binop
    ⋮
```

Figure 3.9: An excerpt from `CIL` using GADTs

(addition of two integers) binary operation is only applied to expressions representing integers.

The representations are used when generating code from the IL. Representations are also parametrised by the types that they can be used to represent. For instance, it is not possible to create an integer expression with a representation that is used for representing structures.

Rather than implement the interface used by the `Translate` module directly, CIL provides a functor for creating such implementations. The argument to this functor is a `MachineDescription` module which describes how C's types should be represented and other machine-dependent parts of the C standard. This allows EMCC to target multiple architectures.

3.6 Conclusion

In this chapter we have described the design and implementation of the EMCC compiler, a C compiler which supports extensions to its front- and middle-ends. This compiler has been successfully implemented in OCaml and is complete enough to compile large C code bases (including the C standard library).

EMCC has an extensible front-end that takes advantage of OCaml's class system to implement a system based on Attribute Grammars. These grammars are used to allow the semantic analysis phase of the front-end to be extended. EMCC does not have an extensible parser, but supports syntax extensions by allowing new attributes, pragmas and built-in functions to be defined.

EMCC has an extensible middle-end which abstracts the details of the IL using the OCaml module system. In particular, the front-end, the back-end and the analyses and transformations of the middle-end are implemented as functors. Design patterns such as properties and visitors are used to make implementing these functors easier. EMCC includes a default IL which uses GADTs to ensure that it is correctly typed.

This compiler has been used to create an OpenMP implementation, which is described in Chapter 5.

Chapter 4

Run-time library

Not every aspect of a high-level language is implemented in the machine code generated by the compiler's back-end. Some features are typically implemented with the help of a run-time library. These run-time libraries implement higher-level features than those provided by the architecture's machine instructions. A classic example is garbage collection. Like compilers, these run-time libraries translate an input language (the library's API) into an output language (instructions executed on the architecture). As with compilers, a single run-time library may be capable of translating multiple input languages (different APIs targeted by different compiler back-ends) into multiple output languages (instructions for different architectures).

In order for run-time libraries to support new architectures and new language features they must be extensible. This extensibility must run through the whole library. Architectures with similar high-level features may also perform very differently with different low-level algorithms and data structures. To support as many architectures as possible, it is important that the library can be customised to use different instructions, algorithms and data structures depending on the architecture.

In this chapter we outline the design and implementation of such an extensible run-time library. In Chapter 5 we implement OpenMP with EMCC, and the run-time library that we implement in this chapter is focused on providing the functionality required by that implementation. In Chapter 7 we extend our OpenMP implementation to support heterogeneous architectures, using the Cell Broadband Engine [14] as our example. For this reason, it is important that our library can target both the `x86_64` architecture used in Chapter 5 and the Cell architecture used in Chapter 7. This provides us with a good example of the need for extensibility in run-time libraries.

Section 4.1 describes how we use the library's build system to increase its modularity, and thus increase its extensibility. Section 4.2 gives an overview of the data structures and algorithms provided by our run-time library. Section 4.3 describes how we abstract different kinds of atomic operation, so that the implementations of concurrent data structures

and algorithms described in Section 4.2 can run on a large variety of different architectures.

4.1 Modular build system

A key aspect of extensibility is modularity. It allows us to easily change which implementation of a component is used for a particular instance of our library. Since performance is important for a run-time library, any such customisation must occur at compile time. The simplest way to change which implementation of a component is used at compile-time is through the build system. This means providing a *build script* that chooses which components are required, and which implementations of these components should be used.

We can check for errors in these build scripts by having each component implementation provide a description of the interface that it implements. In our case we describe these interfaces as a list of the symbols that it requires and the symbols that it defines. Using these lists we can ensure that a build script is not missing a component, or choosing an implementation that does not correspond to the desired component. This can be thought of as a kind of poor man's module system, where the modules are groups of header files and source files which provide definitions of particular symbols. If two "modules" provide the same set of symbols[1] then they can be thought of as having the same "module type".

In addition to choosing which components to use in an instance of our library, we also wish to allow one component to be parametrised in terms of another. For example, we may want one tree data structure using lock-based lists and another using lock-free lists, so that the tree data structure is parameterised by the list implementation it uses.

We can implement this using C macros. By always referring to a component in the C code through a C macro, the build system can change which implementation is used by changing the definition of the C macro. To support this, a component implementation must provide a list of macros which need to be defined, then uses of the implementation within the build script must include a list of component implementations to be assigned to these macros. These parameterised components are analogous to a functor in a module system.

We implemented a simple build system with this design using a simple shell script, which could easily be integrated into a configuration script. By specifying a different build script for different architectures, we are able to customise our library at compile-time to use the most appropriate collection of data-structures and algorithms for that particular architecture.

[1] We do not check the actual C types of these symbols, leaving this as a problem for the C compiler.

4.2 Library overview

This section gives an overview of the facilities provided by our run-time library.

4.2.1 Thread teams

The library provides facilities for creating teams of threads. These threads could be implemented by any mechanism that can support a full C execution context (user-level threads, operating system threads, processes, etc.). All the threads in a team start executing the same function, and each is given an index within that team so that different work can be assigned to different threads. The thread which creates the team also executes the function, and when all the threads have finished it returns from the thread team creation function and continues its execution.

To allow the threads to communicate with each other, each thread is allocated a block of shared memory. Threads can look-up the address of the shared memory of another thread in order to communicate with it.

4.2.2 Memory allocator

The data structures and algorithms provided by the library require dynamically allocated shared memory. This is provided in the form of a simple memory allocator. It only allocates cache-aligned blocks of a single size (intended to be the cache-line size). These blocks are accessed using the atomic operations described in Section 4.3. By restricting data to the size of a single cache-line the library encourages algorithms which have good locality and make this locality explicit. It also greatly simplifies the design of the allocator.

Each thread in a team allocates blocks from its own pool. This pool consists of a linked list of previously allocated nodes and a page of fresh memory for allocating new nodes if the list is empty. Each list only has access to its own free list, so that blocks can be allocated without synchronisation. However, each thread also has a public free list whose items are moved onto its private free list when the private list is empty. Any thread may deallocate a memory block, however it must be deallocated onto the free lists of the thread which created it.

4.2.3 Concurrency primitives

The library provides various concurrency primitives that are used to implement OpenMP. These include mutexes, barriers and atomic counters. Some of these primitives are exposed by OpenMP directly as library functions.

4.2.4 Concurrent data structures

Implementing OpenMP requires a variety of concurrent data structures. The library provides lock-free implementations of all these data structures based on the atomic operations described in Section 4.3.

The most important data structure for implementing OpenMP is a single-producer multiple-consumer dequeue. This is based on the design of Hendler et al. [33]. It allows one thread to add and remove items from one end of the dequeue, like a stack, while allowing multiple threads to remove items from the other end. By creating one such dequeue per-thread, threads can fetch their own most recently created task for execution, or steal the least recently created task from another thread. Similar designs of a single-producer multiple-consumer stack and a single-producer multiple-consumer queue are also provided by the library.

The library also provides a tree data structure using reference-counted memory blocks that form an upwardly linked tree. Each block in the tree contains a pointer to its parent, and each such pointer is included in the reference count of the parent.

4.2.5 Worksharing primitives

The library uses the data structures and concurrency primitives described above to provide primitives for implementing OpenMP's worksharing constructs. In particular, it provides support for OpenMP's `for` construct which dynamically divides the iterations of a loop amongst the threads in a team.

4.2.6 Task primitives

The library also uses the data structures and concurrency primitives described above to provide primitives for implementing OpenMP's task-based parallelism as described in Chapter 5. This includes maintaining per-thread task pools, support for calling the continuations that represent suspended tasks, and control of of the C stack using the C standard library's `setjmp` and `longjmp` functions. Note that these primitives are specific to the continuation-based approach used by our OpenMP implementation, they could not be used as the run-time library targeted by other OpenMP implementations.

4.3 Atomics

The atomic instructions provided by a particular architecture can vary widely. We would like to be able to abstract away these differences so that a single implementation of a concurrent data structure can be used across all the different architectures supported by the library.

Operation	Description
`read`	An atomic load operation.
`write`	An atomic store operation.
`load_linked`	A strong LL operation with *acquire* memory ordering.
`store_conditional`	A strong SC operation with *release* memory ordering.
`compare_and_swap`	A CAS operation with both *acquire* and *release* memory ordering.
`swap`	An swap operation with both *acquire* and *release* memory ordering.
`fence`	Ensures all previous operations are visible to all threads.

Table 4.1: Some Atomic Operations

When choosing which atomic operations the library should provide, we wanted to be able to support two common styles of instruction set, namely:

1. Those instruction sets, for instance x86 and PowerPC, whose atomic instructions operate on a word at a time.

2. Those instruction sets, like the Synergistic Processing Element of the Cell Broadband Engine [14], whose atomic instructions operate on an entire cache line.

In order to support these two situations efficiently, we use a simple programming model which has both a cache-based aspect and a word-based aspect. In this model atomic operations are performed by first *loading* a cache line with the `load_cache` statement, then performing the operations themselves on offsets of that cache line. If the atomic operations need to make changes to the cache line then the `write_cache_aborted` function must be called after the operations have completed to check that they were successful. If `write_cache_aborted` returns true then the operation was unsuccessful and no changes have taken place.

Even though `write_cache_aborted` may indicate that none of the atomic operations since the last `load_cache` have completed, the operations on a cache line do not happen all at the same time. An individual atomic operation may become visible to other threads before `write_cache_aborted` has been called. Similarly, changes to the cache line by other threads may become visible after `load_cache` has returned.

A sample of the atomic operations provided by the library are detailed in Table. 4.1.

Using these atomic operations we are able to create lock-free implementations of many useful data structures. As an example, a simple spin-lock is shown in Fig. 4.1.

When implementing this programming model using word-based atomic instructions the `load_cache` and `write_cache_aborted` operations become essentially no-ops. Similarly, on an architecture with cache-line based atomic instructions operations like `swap_uint` are implemented by ordinary reads and writes.

```
void acquireLock(cache_addr lock)
{
  uint old;
  do
  {
    load_cache(lock);
    old = swap_uint(SPIN_LOCK_INDEX, LOCKED);
  } while((old == LOCKED) || write_cache_aborted());
}

void releaseLock(cache_addr lock)
{
  do
  {
    load_cache(lock);
    write_uint(SPIN_LOCK_INDEX, UNLOCKED);
  } while(write_cache_aborted());
}
```

Figure 4.1: A simple spin-lock

The flexibility of this approach allows this programming model to also be implemented using per-cache-line locks, by implementing `load_cache` as an acquire operation and `write_cache_aborted` as a release operation. This gives us a fine-grained lock-based implementation of our data-structures for free.

Note that some of the supported architectures implement the atomic operations using weak load-linked/store-conditional. This means that, although the design of our data structures is lock-free, on these architectures the implementations are only really obstruction-free. Similarly, an architecture using locks to implement its atomic primitives would produce data structures that were not non-blocking.

The emphasis on cache lines in our atomic model encourages a focus on reducing the number of cache lines accessed and prevents false sharing. This can help improve the locality of our algorithms, which in turn improves their performance.

The cache-line aspect of our model is very similar to Software Transactional Memory (STM) [66], with `load_cache` similar to the start of a transaction and `write_cache_aborted` similar to a commit operation. However, our model does not require transactions that consist entirely of reads to perform a commit operation. This means that it would be difficult to implement our model directly using STM.

4.4 Conclusion

In order for run-time libraries to support new architectures and new language features they must be extensible. In this chapter we have described the design and implementation of an extensible run-time library to support our OpenMP implementation (see Chapter 5).

We have described how our run-time library uses a modular build system to enable component implementations to be selected at compile-time. It also supports components parametrised by other components.

We have also outlined how our library abstracts the details of atomic operations on a particular architecture. This enables our (lock-free) data structures to have a single implementation which works on a wide variety of architectures. In particular, we support the x86_64 architecture which we use for our initial OpenMP implementation in Chapter 5 and the Cell Broadband Engine architecture which we use for our heterogeneous extensions to OpenMP in Chapter 7.

Chapter 5

Efficient implementation of OpenMP

This chapter describes our implementation of OpenMP(see Section 2.3) as an extension in EMCC. We particularly focus on the implementation of task-based parallelism (see Section 2.2), because previous implementations of OpenMP's task system have struggled to compete with the performance of other task-based systems like Cilk [62, 60].

Task-based parallelism in OpenMP is an interesting example of the need for extensible compilers. OpenMP was originally designed for scientific applications on multi-processor systems. It provided static parallelism which did not require complicated program transformation, and could be implemented in the front-end of the compiler or with a simple preprocessor. Then OpenMP 3.0 added support for task-based parallelism, as part of an effort to evolve towards supporting more mainstream applications. However, implementing task-based parallelism efficiently requires much more involved program transformation than the simple static parallelism OpenMP originally supported. These transformations are best implemented in the middle-end of the compiler, but current implementations of OpenMP are still being implemented in the front-end of the compiler. This has prevented their performance from competing with that of Cilk and other task-based programming languages [60]. It should be said that, even with these less efficient implementations, task-based parallelism can still dramatically improve the performance of some algorithms in OpenMP [5].

Efficient implementation of task-based parallelism requires tasks to be as lightweight as possible. The resources used by a set of tasks should scale linearly with the number of live tasks. Lightweight tasks require an efficient representation of the current state of a task. An important part of a task's state is its task-local variables (analogous to the local variables of a traditional procedural language). Existing implementations of OpenMP handle task-local variables by allocating a stack for each task in the program. However, giving each task its own stack is very inefficient. This inefficiency forces these implementations to use load-based inlining (see Section 2.2), which can severely restrict the parallelism of a program.

On recent architectures with 64-bit address spaces and support for memory overcommit, it may not seem that large amounts of unused space are a problem. However, per-thread stacks produce both excessive and fragmented memory usage which can result in many page and TLB misses, significantly damaging performance. Support for memory overcommit is also controversial and not very portable, and it is common for architectures to allow users to disable it.

Section 5.1 gives a theoretical demonstration that OpenMP tasks can be scheduled in a space-efficient way without affecting time efficiency. It also shows that using per-thread stacks is not space efficient, and is likely to be inefficient in common cases. Section 5.2 gives an overview of how OpenMP is implemented efficiently using EMCC and our customisable run-time library. Section 5.3 contains experimental results for various benchmarks. Related work is discussed in Section 5.4.

5.1 Efficiency of task-based computations

Both Cilk and OpenMP 3.0 are extensions to the C programming language that support task-based parallelism.

Cilk places various restrictions on where tasks can be created and how tasks synchronise with each other (e.g. a task cannot outlive its parent). These restrictions provide certain guarantees about the behaviour of tasks in Cilk. Blumofe et al. [7] used these guarantees to show that Cilk programs could be scheduled in a way that was guaranteed to be space efficient, without affecting their time efficiency. This space guarantee, combined with fast implementations of task creation and destruction, mean that Cilk never has to resort to load-based inlining.

OpenMP's support for task parallelism places fewer restrictions than Cilk on how tasks may behave (e.g. OpenMP tasks can outlive their parents). OpenMP lacks the guarantees about task behaviour that Blumofe et al. used to show that all Cilk programs can be scheduled in a space-efficient way without affecting time efficiency. Without a guarantee of space efficiency, many current implementations of OpenMP allocate activation frames from per-task stacks, and fall back on load-based inlining to ensure space efficiency. These implementations have struggled to compete with the performance of Cilk [62, 60], especially for benchmarks with fine-grained tasks.

In this section, we show that OpenMP tasks can be scheduled in a space-efficient way without affecting time efficiency.

5.1.1 Execution model

Task-based parallel programming languages, such as OpenMP and Cilk, share a basic underlying task-based computation model. This model consists of *threads* executing *tasks*.

Tasks are sequences of instructions to be executed by a thread. When a computation begins there is a set of initial tasks to be executed. These tasks can in turn *spawn* more tasks, creating a directed forest of tasks (which we call the *task tree*). A task can also perform a *sync* operation, which prevents it from being executed until all of the tasks that it has spawned have been finished.

We define a number of terms relating to the current state of a task. If some of a task's instructions have been executed then we say that the task has *started*. If all of the instructions have been executed we say that the task has *finished*. If a task has been spawned but has not yet been started we say that the task is *waiting*. If a thread executes instructions from one task and then executes instructions from a different task, even though the first task has not finished, we say that the first task has been *suspended*. If the execution of a suspended task resumes on a different thread than the one that had previously been executing it, we say that the task has *migrated*. If a thread is suspended at a sync operation, and some of its child tasks have not yet finished, we say that the task is *blocked*. We write $\prec$ for the tree relation between tasks formed by the spawn edges (e.g. $\tau \prec \tau'$ means that τ' is descended from τ). If the completion of a task depends on the completion of all its child tasks (i.e. every spawn is followed by a sync) then we say that those tasks are *nested*.

In terms of the model of parallel computation described Section 2.4, a task-based computation is a parallel computation that obeys some computation restrictions. The computation must be partitioned into a set of tasks $\mathcal{T}$, where each task τ is a directed path $n_0, \ldots, n_k$ through the computation. Only three kinds of edges are permitted in a task-based computation:

1. *Continue edges* – The edges within a task's path
2. *Spawn edges* – The edges that represent spawns. They proceed from the parent task into the first instruction n_0 of its child.
3. *Join edges* – The edges that represent syncs. They proceed from the final instructions n_k of all the child tasks to the sync instruction of the parent task.

We define the lifetime of a task τ with instructions $n_0, \ldots, n_k$ as:

$$\text{lifetime}_X(\tau) = [\omega^{-1}(n_0), \omega^{-1}(n_k)]$$

There are also two scheduling restrictions placed on scheduling algorithms in task-based systems. First, since the computations are not known in advance, the scheduling algorithms cannot rely on information about the "future" of a computation. Second, the scheduling algorithm cannot use preemption, which means that a thread can only stall a task at specific points in that task's sequence of instructions. We call these points *task scheduling points*.

5.1.2 Memory model

The memory model of these task-based languages is based on that of single-threaded languages like C. The variables in a task-based computation can be divided into three kinds:

1. Global variables, which are allocated for the duration of the program. Note that this includes the *thread-local variables* supported by many C compilers, which are global variables with local versions for each operating system thread.

2. Automatically allocated task-local variables, which are allocated for the duration of a code block or function call. These would typically be allocated on the stack in a single-threaded program.

3. Dynamically allocated space on the heap, which can be allocated and deallocated at any point in the program.

It is generally considered the programmer's responsibility to control the amount of dynamically allocated memory (3) live at any point in the program; accordingly we discuss this no further. Similarly, since global variables (1) are always allocated for the duration of the program, they are not relevant to a discussion of space efficiency.

For these reasons, we only consider task-local variables (2). Since all the variables associated with a code block are allocated at the same time and deallocated at the same time as part of an *activation frame*, they are represented within our model of parallel computations by a single variable v that is accessed by the first and last instructions of the code block.

5.1.3 Scheduling tasks efficiently

We would like to use scheduling algorithms that are both absolutely time efficient and absolutely space efficient (see Section 2.4.6). However, the scheduling restrictions of task-based systems mean that there are no absolutely space-efficient scheduling algorithms. This means that we will have to settle for space-efficient schedules whose single-threaded schedules, while not optimal, perform within the expectations of a programmer.

To do this we will first define some kinds of scheduling algorithms for task-based computations. A *depth-first* scheduling algorithm is one that assigns work to threads from the task tree in a depth-first manner. This means that once a thread has started executing tasks from a sub-tree of the task tree it will continue to execute tasks from that sub-tree until there are none available.

There are two kinds of depth-first scheduling algorithm that are of particular interest. A *post-order* scheduler will assign work to threads so that a thread will continue to execute a task until it reaches a task spawn or the task finishes, and when a thread reaches a task

spawn it continues by executing the spawned task. A *pre-order* scheduler will assign work to threads so that a thread will continue to execute a task until it reaches a task sync or the task finishes.

Single-threaded schedules from either a pre-order or post-order scheduling algorithm are the schedules most likely to match a programmer's expectation of how much space their program will require. We will write $S_{\mathcal{D}}$ for the minimum execution space for a computation using a post-order scheduling algorithm and $S_{\mathcal{W}}$ for the minimum execution space for a computation using a pre-order scheduling algorithm. We consider any scheduler whose single-threaded schedules use either $S_{\mathcal{D}}$ or $S_{\mathcal{W}}$ space to be performing within the expectations of the programmer.

We can create pre-order scheduling algorithms (see Appendix B) that are space efficient and whose single-threaded schedules use $S_{\mathcal{W}}$ space. These scheduling algorithms can also be made greedy, so that they are also absolutely time efficient. This shows that we can create schedules that are both time and space efficient.

The scheduling restrictions on task-based systems prevent us from creating a post-order scheduling algorithm that is absolutely time efficient, space efficient and whose single-threaded schedules use $S_{\mathcal{D}}$ space. However, it is possible (see Appendix B) to create a post-order scheduling algorithm that is absolutely time efficient, space efficient and whose single-threaded schedules use $O(S_{\mathcal{D}} + S_{\mathcal{W}})$ space. Since in practice the space requirement will usually be very close to $S_{\mathcal{D}}$, and there are practical advantages to using a post-order schedule (see Section 5.2.1), these algorithms are likely to be the most useful in practice.

Intuitively, both the above proofs work because, in a depth-first schedule, all live tasks are either being executed or one of their children is being executed. The tasks whose children are being executed would also have been live when the single-threaded version was executing that child task. This means that each live variable can be associated with a thread, and each thread is associated with no more live variables than the single-threaded version would have been when executing the same task.

5.1.4 Scheduling OpenMP tasks efficiently

If an OpenMP program contains no tied tasks, then the above results for general task-based computations still apply.

Unfortunately, the restrictions on tied tasks prevent the creation of an absolutely time-efficient scheduling algorithm. This is because in general it is impossible to tell in advance whether it would be quicker to begin executing a new tied task or wait for another thread to become free to execute it. The nature of tied tasks also means that any scheduling algorithm that schedules tied tasks in post-order can cause a complete loss of speedup, because if the parent task is tied it cannot migrate to another thread.

However, the time efficiency of an OpenMP scheduling algorithm only depends on decisions about whether a thread, with no other available tasks, should start executing the descendant of a task tied to another thread, or stall and wait for more tasks to become available.

These decisions are not affected by the requirements of a pre-order depth-first scheduler. This means that, given an OpenMP scheduler, we can create a new scheduler that is space efficient and whose single-threaded schedules use $S_{\mathcal{W}}$ space, and that is as time efficient as the original scheduler.

This means that OpenMP tasks, including tied tasks, *can* be executed in a space-efficient way without affecting time efficiency. As mentioned in the previous section, we prefer to schedule untied tasks in post-order, so in practice we prefer to use a scheduling algorithm that schedules its tied tasks in pre-order and its untied tasks in post-order. The above proofs still hold for such a schedule.

5.1.5 Inefficiency of stack-based implementations

Previously reported OpenMP implementations [1, 71, 72] have allocated task-local memory by allocating a whole stack for each task in an OpenMP computation. It can be shown (see Appendix C) that maintaining space efficiency in such a system can cause a complete loss of speed-up.

Intuitively, the limit on the number of stacks, which is required to maintain space efficiency, is also a limit on the number of *stolen* tasks (i.e. tasks executing on a different thread than their parents) that can be live at any point. This inevitably limits parallelism for some kinds of computation.

Not only is allocating each task its own stack theoretically inefficient in the worst case, it is likely to be inefficient for common task-based patterns. The inefficiency of this system can be seen by considering how much of its stack a task is actually likely to use. For any program where the tasks have a low *stack depth* (i.e. they do not allocate many activation frames at once), increasing the amount of parallelism will increase the inefficiency of a stack-based implementation.

Task-based programs express parallelism in two ways:

Recursively Creating child tasks which in turn create more child tasks etc.

Iteratively Creating many child tasks from a single parent.

Recursive tasks that are not nested are also a type of iteration, since they amount to a tail-call.

Recursive task patterns produce a large *task depth* (i.e. they have a deep task tree), which forces each individual task to have a low stack depth. This is because, if the program is run on a single-thread, all these tasks must be able to fit their activation

frames onto a single stack. Even in the case of iterative task patterns, where task depth is low, unless the tasks are executing deep recursive functions they are still likely to have a low stack depth.

We conclude that a system using separate stacks per task is only likely to be space efficient for iterative task patterns where the individual tasks execute deep recursive functions.

5.1.6 Scheduling overheads

In our discussion so far we have overlooked the fact that some memory must always be allocated to represent a task that has been created but not started. In a post-order schedule, the total amount of this scheduling data that is allocated at any point is bounded by the task depth, which in turn is bounded by $S_{\mathcal{D}}$. However, in a pre-order schedule there is no such bound. Since tied tasks become serialised when executed in post-order, this means that sometimes we must still choose between space efficiency and time efficiency. However, this is only possible for programs that use a tied task to create a huge number of tasks, and in these cases it is likely that briefly executing the tied task in post-order will not actually affect the time efficiency.

5.2 Implementing OpenMP

This section describes how we implement OpenMP as an extension for EMCC, focusing on the efficient implementation of OpenMP tasks. The other aspects of OpenMP are implemented in much the same way as in previous OpenMP implementations, and their performance is not affected by our approach to task-based parallelism.

5.2.1 Efficient task-based parallelism

Representing tasks

In order to keep the tasks in our implementation as lightweight as possible we represent suspended tasks as simple continuations. These consist of a pointer to the current function being executed by the task, an integer representing which suspension point within the function the task is suspended at, and a pointer to the activation frame for that function.

Resuming suspended tasks involves calling these continuations. We call a continuation by calling the function pointer with the suspension point and activation frame as arguments. This requires us to transform functions into a form which accepts the suspension point and activation frame as arguments.

The activation frames must contain the arguments, return addresses and returned values of function calls in our implementation. Since C does not allow programs to

control where these values are stored, we cannot use C's procedure calling mechanism and must instead transform them to use a continuation-passing style. This means that the return address of a function is stored as a continuation within its activation frame, which is called when the function is finished with the return value as an argument.

Separating suspendable and non-suspendable procedures

Since tasks can only be suspended at task synchronisation points, any code that does not contain such a point is guaranteed to execute to completion on a single thread without suspending. This means that if a procedure is not *suspendable* (i.e. does not contains a task synchronisation point) then it will never be used as the function pointer within a continuation. So non-suspendable procedures do not need to be transformed to use our continuation calling convention, and can be left as regular C procedures.

We use a simple inter-procedural static analysis to determine whether or not a procedure is suspendable, and only transform procedures marked as suspendable.

Activation frames

The work in Section 5.1 showed that OpenMP tasks can be scheduled so that they are space efficient without affecting time efficiency, but that giving each task its own stack cannot achieve this efficiency.

Note that OpenMP does not guarantee that tasks are nested (unlike Cilk). This prevents us from using one stack per-thread solutions like the one used in TBB [64] (which are not space efficient anyway), or a cactus stack implementation like the one suggested by Lee et al. [46].

This means that we must, in general, allocate activation frames using dynamic allocation from the heap. In our implementation we allocate frames using the `malloc` function provided by the C standard library.

As with the continuation transformation, only suspendable procedures need to dynamically allocate their activation frames. Non-suspendable procedures continue to use the regular C stack for their local variables.

Live variable analysis

Rather than operate directly on the variables in the heap-allocated activation frame, we keep their values in normal local variables and copy them into the activation frame when crossing a task suspension point. In order to detect which variables need to be written to the activation frame across a particular suspension point, we perform a live-variable analysis. This means we will only store a value if it may be needed later.

Lazy task creation

While our task representation requires us to support a continuation-based calling mechanism, we can still take advantage of C's procedure calling mechanism in the most common cases by using a technique called *lazy task creation*. This technique relies on using a post-order scheduler and works on the assumption that most of the time a parent task will still be available for execution after its child task has completed.

When a new task is created, the program attempts to execute it as a normal procedure. However, it will also store enough information about the parent task to allow it to be stolen by another thread. If the parent task is still available when the child task returns, the program simply continues as normal.

Similarly, when a suspendable procedure is called it is treated as a normal procedure, but enough information is stored to allow the calling procedure (which must also be suspendable) to be suspended if the task is stolen. If the task is not stolen while the procedure is executing then the program simply continues as normal.

This means that only when executing a task that has been stolen do we have to use the continuation-based calling mechanism.

We implement lazy task creation using the *fast/slow clones* method, which was designed for use in Cilk-5 [27]. This means that we actually create two versions of every suspendable procedure. The first time a suspendable procedure is called the *fast* version is used, but if that procedure is suspended then it is the *slow* version that is resumed. The fast versions are called and return using the regular procedure calling mechanisms, while the slow versions are called and return using the continuation-based mechanism.

Serial-parallel reciprocity

Allocating (suspendable) procedures' activation frames on the heap requires a non-standard calling convention. However, OpenMP is supposed to be compatible with standard C, so we must also be able to provide versions of these procedures that use the standard stack-based calling conventions. The same problem arises due to function pointers – if we call a procedure indirectly we do not know which calling convention it uses.

In Cilk this problem is avoided by using the type system to prevent these two situations, however OpenMP does not place such restrictions on the use of suspendable procedures. Other solutions to this problem, such as those discussed by Lee et al. [46], rely on the heap-based frames being allocated on a cactus stack, which is not possible for OpenMP.

To avoid this problem, we create wrapper procedures that call the suspendable procedures using the correct calling convention. Since the activation frames of the procedures that called the wrapper are on the C call stack, and cannot be properly suspended, the task is forced to become tied to the thread executing it. These activation frames also

prevent the stack from being unwound, so if more than one of these tasks is on a single stack only the topmost one is available to be resumed.

This solution risks both deadlock and space inefficiency. To prevent this, we restrict the scheduling algorithm, only allowing a thread to resume tasks that are deeper in the task tree than the tasks "stuck" on its stack. Unfortunately this may reduce the parallelism of the program, however it is a very similar condition to the one used by Intel's Threading Building Blocks (TBB) [64] to prevent its tasks from being space inefficient, and TBB has been shown to perform very well when compared with published OpenMP implementations.

5.2.2 Implementing OpenMP in EMCC

OpenMP IL

In order to implement OpenMP in EMCC, we create a new IL to represent an OpenMP program. This IL is based on the flowgraph version of `CIL` extended with graph edges to represent OpenMP constructs. It also adds representations for suspendable procedures—which use a different calling convention from regular procedures.

The front-end

Since OpenMP only uses pragmas to extend the C syntax, the EMCC lexer and parser do not need to be extended at all. For semantic analysis, we register handlers in `Translate` for each of the OpenMP pragmas. These handlers produce nodes representing the appropriate OpenMP construct, including its translation into the OpenMP IL. The handlers also extend the environment so that we can associate each OpenMP construct with its local variables.

Fig. 5.1 shows some example code used to handle the `task` construct The `task_construct` class extends the `block` class used for handling structured blocks. The IL node for the construct is returned by the `output` method. This IL node is constructed using the `Task` constructor and contains the IL node representing its child block. The `task_construct` function uses the `task_construct` class to handle task constructs. It also modifies the environment using the `enter_task` function.

The middle-end

Our middle-end performs a number of analyses and transformations, including:

Outlining of the bodies of `parallel` and `task` constructs into their own function definitions. This is necessary so that they can be referred to by function pointers. These constructs already have their own lists of local variables and control-flow

```
class task_construct blk ··· = object
  inherit block
  ···
  method output = Task(blk#output, ··· )
end

let task_construct env blk ··· =
  let tk = new task_construct blk in
  let env = enter_task env in
    (tk, env)
```

Figure 5.1: Code to handle the `task` construct

graphs within our IL, so it is very simple to treat them as function definitions when generating C code.

Suspendable procedure analysis is performed to detect which procedure definitions contain task suspension points or calls to other suspendable procedures. These are transformed into suspendable procedure definitions, including a list of all possible suspension points.

Live variable analysis is performed on suspendable functions to annotate all suspension points with the variables that must be maintained in the activation frame during a possible suspension.

Reaching definition analysis is performed to further refine the lists of variables which must be copied to activation frames. This allows us to only copy values which may have changed since the last suspension point.

Each of these is implemented as a functor that expects functions and visitors for inspecting and translating different parts of the IL.

The back-end

We use the `Generate` module to convert the OpenMP IL into C. The OpenMP constructs are handled using a combination of low-level C code and calls into our OpenMP run-time library.

5.2.3 Implementing OpenMP in our run-time library

The high-level OpenMP constructs are implemented using the run-time library described in Chapter 4. For example, the `parallel` construct is translated into a call to the run-time to create a team of threads executing the body of the construct (which has been

outlined into its own function definition). Similar support is provided for the worksharing constructs.

The scheduler used for task-based parallelism is also implemented in the run-time library. When a task is spawned, a continuation for the current task is registered with the scheduler. When a thread has finished all its tasks (or had its remaining tasks stolen), it calls a function in the run-time library which will find a new task to execute and then begin its execution. This function also handles the removal of dead stack frames from the thread's stack when tasks have been stolen from the thread.

5.3 Evaluation

5.3.1 Benchmarks

We evaluated our implementation using six benchmarks from the Barcelona OpenMP Tasks Suite [23].

Alignment Aligns all protein sequences from an input file against every other sequence. It uses an iterative pattern, with a parallel `for` loop containing an inner loop that spawns tasks.

Health Simulates the Colombian Health Care System. It uses a recursive divide-and-conquer pattern, with each task spawning multiple tasks in a loop and then waiting for them to finish.

N Queens Computes all solutions to the n-queens problem. It uses a recursive divide-and-conquer pattern, with each tasks spawning multiple tasks in a loop and then waiting for them to finish.

Sort Sorts numbers with a fast parallel variation of merge sort. It uses two parallel algorithms: one for merging and another for sorting. They both use recursive divide-and-conquer patterns, with each task spawning a fixed number of tasks and then waiting for them to finish.

Sparse LU Computes an LU matrix factorisation over sparse matrices. It uses an iterative pattern, with a single task spawning all the other tasks from within a `for` loop.

Strassen Multiplication of large dense matrices using hierarchical decomposition. It uses a recursive divide-and-conquer pattern, with each tasks spawning a fixed number of tasks and then waiting for them to finish.

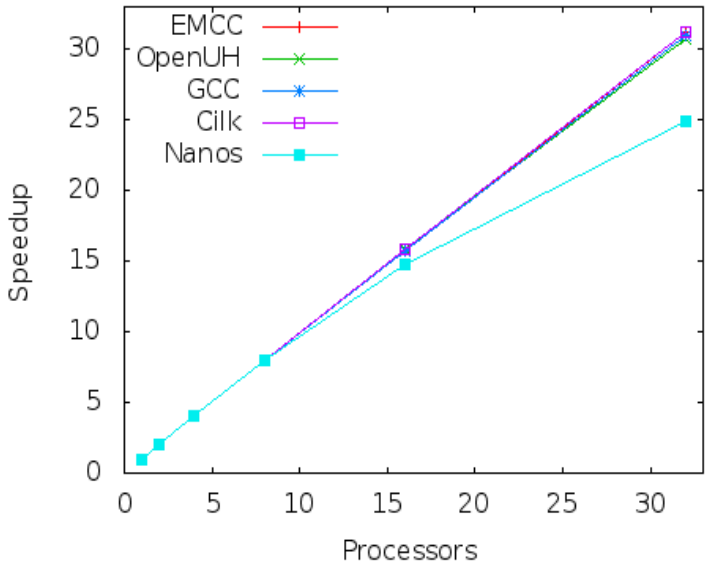


Figure 5.2: Alignment

We chose the Barcelona OpenMP Task Suite over the NAS OpenMP Parallel Benchmarks because the NAS benchmarks have not yet been updated to take advantage of task-based parallelism in OpenMP.

In addition to EMCC, we also ran the benchmarks using the GCC, NANOSv4 [71] and OpenUH [1] implementations of OpenMP. Each of these implementations uses a stack per-thread, although OpenUH is a relatively lightweight implementation, using coroutines to implement its tasks. We also converted the benchmarks to the Cilk language and ran them using Cilk version 5. The benchmarks were run on a server with 32 AMD Opteron processors (model 6134).

5.3.2 Results

Figs. 5.2–5.7 show the speed-ups for each benchmark.

Alignment is an embarrassingly parallel algorithm, and showed roughly linear speed-up on all implementations. Both Health and Strassen showed noticeably worse speed-up on the heavyweight implementations GCC and NANOSv4. N Queens and Sort both showed noticeably better speed-up for the two implementations that allocate their activation records on the heap (i.e. EMCC and Cilk). The Sparse LU result is something of an anomaly; we do not know what causes the poor performance of Cilk and also expected the heavyweight implementations to perform better on an algorithm with an iterative pattern.

Overall these results show that using the heap to allocate activation records can no-

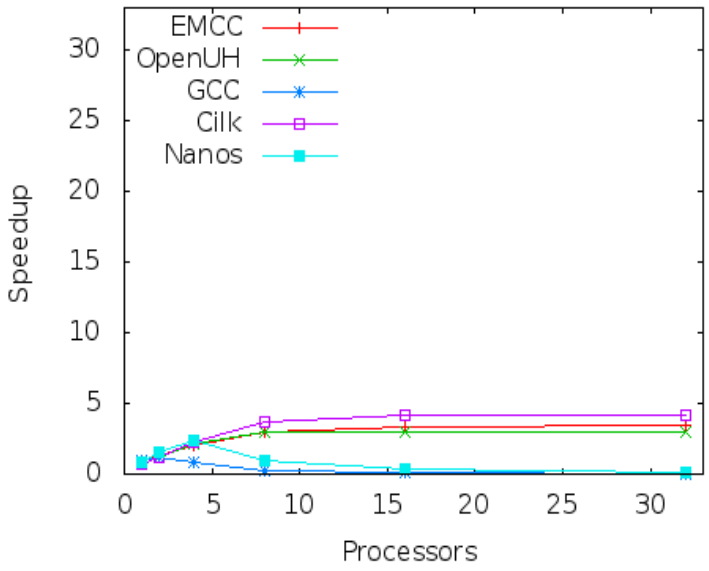


Figure 5.3: Health

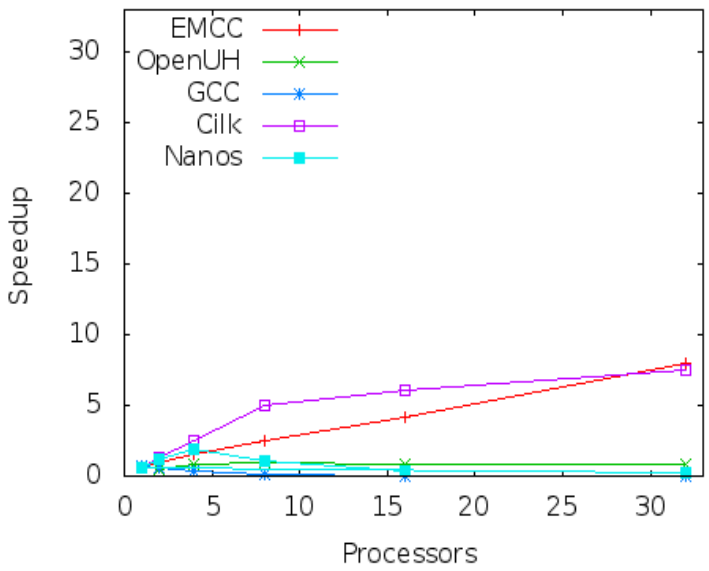


Figure 5.4: N Queens

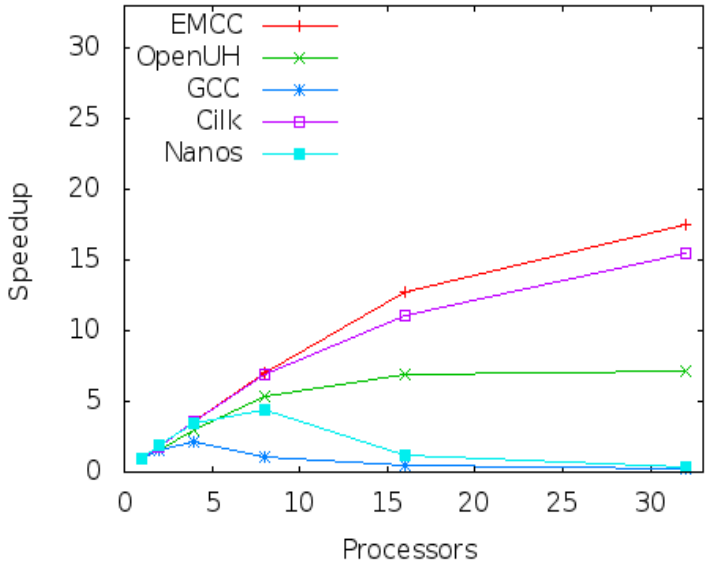


Figure 5.5: Sort

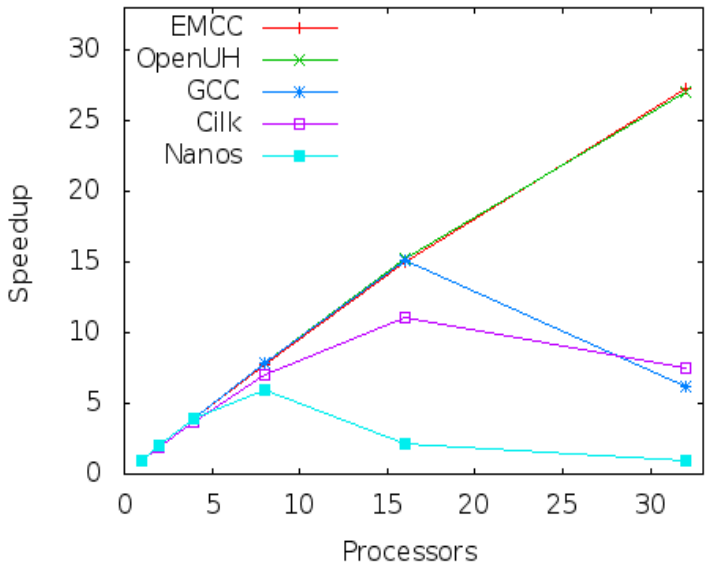


Figure 5.6: Sparselu

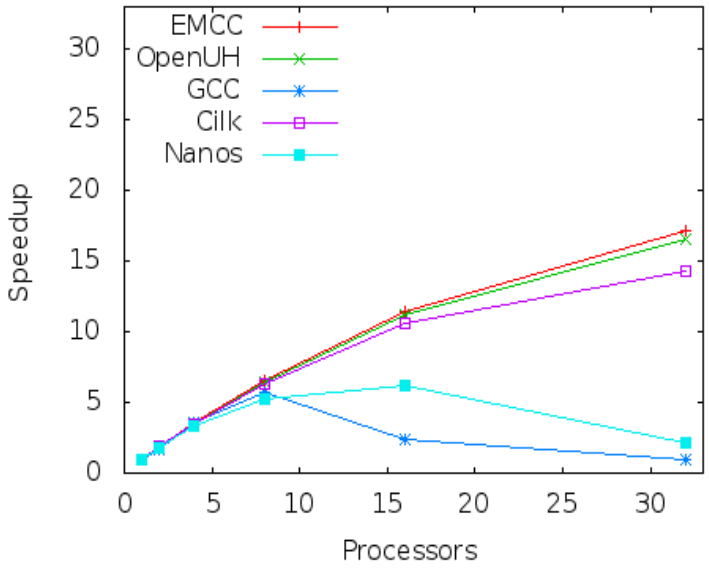


Figure 5.7: Strassen

ticeably improve performance. They also show the superiority of a lightweight implementation.

5.4 Related work

Our demonstration that OpenMP tasks can be space efficient without affecting time efficiency is a generalisation of the demonstration by Blumofe et al. [7] that there are absolutely time- and space-efficient scheduling algorithms for Cilk programs. The original demonstration only allowed computations where all tasks were nested and each task only allocated a single activation frame. Under those restrictions $S_1 = S_{\mathcal{W}} = S_{\mathcal{D}}$, so that our result in Section 5.1.3 implies the stronger result that there are scheduling algorithms that are both absolutely time efficient and absolutely space efficient.

The Nanos Mercurium OpenMP compiler [6] is derived from the Open64 compiler. It uses the Open64 front-end to produce an AST including OpenMP constructs and translates that into an AST without OpenMP constructs. It uses a macro template system to specify these translations. The implementation of task-based parallelism in Nanos [71] uses user-level threads to represent tasks, each with its own call stack, and relies on inlining to maintain space efficiency.

The GNU Compiler Collection [68] implementation of OpenMP converts OpenMP constructs into library calls as part of the process of turning the AST into the Single Static Assignment form used by its middle-end optimisations. The GCC implementation of task-based parallelism treats all tasks as tied tasks, and any tasks that are not stolen

are inlined. This means that each task gets its own stack. This design greatly reduces the available parallelism in some programs, which explains GCC's poor performance in many task benchmarks.

The closest implementation of OpenMP to our own is the OpenUH compiler [50], which is based on the Open64 compiler. OpenUH does include OpenMP constructs in the early stages of its middle-end. This enables it to perform loop optimisations on OpenMP parallel loops, which can't be done after the OpenMP constructs have been replaced by calls into a run-time library. The OpenMP constructs are translated late enough to make an implementation of task-based parallelism similar to ours possible, however the OpenUH implementation of task-based parallelism [1] still gives each task its own stack and relies on inlining to maintain space efficiency.

5.5 Conclusion

In this chapter we have described an efficient implementation of OpenMP using our EMCC compiler. Unlike previous implementations of OpenMP, we avoid giving each task its own stack for local variables. This makes tasks more lightweight and allows us to avoid the load-based inlining which has prevented previous implementations from obtaining performance similar to other task-based systems like Cilk.

To show that we do not require load-based inlining, we presented a theoretical demonstration that OpenMP tasks can be scheduled in a space-efficient way without affecting time efficiency. We also showed that using per-thread stacks is not space efficient, and is likely to be inefficient in common cases.

Finally, we provided experimental results, using our implementation, that demonstrate the effectiveness of our approach.

Chapter 6

Optimising task-local memory allocation

In Chapter 5, we showed that allocating activation records from per-thread stacks requires load-based inlining to remain space efficient, and that this inlining can seriously degrade time performance. To avoid this our OpenMP implementation allocates activation records from the heap using dynamic allocation. However, dynamic allocation is more expensive than stack allocation, and in practice most of these activation records will be allocated and deallocated in stack order.

This chapter describes an optimisation that allows multiple tasks to share a single stack. In general, two concurrent tasks sharing a stack would require time-consuming synchronisation between the tasks and would require garbage collection to avoid wasting a potentially unbounded amount of space. However, in some cases a parent task may safely share its stack with some of its child tasks. Consider the OpenMP function shown in Fig. 6.1. Both tasks only require a bounded amount of space, and they both must finish before the parent task (the one which executed the `work` function) finishes. This means that their stack frames could safely be allocated from the parent task's stack (by using different offsets within it). We say that the child tasks' stacks can be *merged* with their parent task's stack.

The stacks of the child tasks created by the spawn instructions in Fig. 6.1 can always safely be merged. Other spawn instructions create child tasks whose stacks can safely be merged in most, but not all, instances. Consider the post-order tree traversal OpenMP function shown in Fig. 6.2. There is no guarantee that the first child task will finish before the second child task begins and they both use unbounded stack space, so they cannot generally be merged. However, our OpenMP implementation executes tasks in post-order: when a thread encounters a spawn instruction it will suspend its current task and begin executing the newly created task. After that new task has finished it will resume its original task (assuming it has not been stolen for execution on another thread). This

```
void add_tree(struct tree_node *root) {
    #pragma omp task untied     // OpenMP spawn
    {   tree_node *p = root;
        while (p) {
            left_sum += p->value;
            p = p->left;
        }
    }
    #pragma omp task untied     // OpenMP spawn
    {   tree_node *q = root;
        while (q) {
          right_sum += q->value;
          q = q->right;
        }
    }
    #pragma omp taskwait        // OpenMP sync
}
```

Figure 6.1: OpenMP example—where spawned stacks can be merged.

```
void postorder_traverse( struct tree_node *p ) {
    if (p->left)
      #pragma omp task untied     // OpenMP spawn
          postorder_traverse(p->left);
    if (p->right)
      #pragma omp task untied     // OpenMP spawn
          postorder_traverse(p->right);
    #pragma omp taskwait      // OpenMP sync
    process(p);
}
```

Figure 6.2: OpenMP example—stack merge is often possible subject to a cheap test.

means that, if the parent task has not been stolen, the first child task in Fig. 6.1 will definitely finish before the second child task begins.

We can merge spawn instructions like the second one in Fig. 6.1 as long as their parent task has not been stolen. This can be checked at run-time cheaply and without synchronisation. We say that such spawn instructions are *merged guarded*, while spawn instructions that can always be merged are *merged unguarded*.

To support this optimisation the compiler must perform an analysis to determine sets M of spawn instructions whose stacks can safely be merged (the *merged set*), and $U \subseteq M$ of spawn instructions whose stacks can safely be merged unguarded (the *unguarded set*).

Note that previous work on predicting the stack usage of programs, for example by Campbell [12], has not considered languages with parallelism and has focused on inferring upper bounds for a fixed program. Whereas this work must allow for parallelism and is focused on optimising the stack usage rather than predicting it.

In order to express this analysis concisely, we develop a generalisation of logic programming (described in Section 2.5). We use a multi-valued logic, with the values representing possible stack sizes.

First we use a *program* in this logic to represent finding the sizes of stacks for a particular pair of merged set and unguarded set. Then, using the notion of a *stable model* which was developed as a semantics for negation in logic programming, we are able to extend this program to express the whole analysis.

By showing a stratification result about the program representing the analysis, we show that the analysis has a single solution and can be solved in polynomial time.

6.1 Model of OpenMP programs

In order to describe our analysis we require a model of OpenMP programs.

6.1.1 OpenMP programs

We represent OpenMP programs as a triple $(\mathcal{F}, \mathit{body}, \mathcal{S})$ where $\mathcal{F}$ is the set of function names, *body* is a function that maps function names to their flowgraph (CFG), and $\mathcal{S} \subseteq \mathcal{F}$ gives the entry points to the program.

We make various assumptions: function names are unique, program flowgraphs are disjoint and the bodies of tasks have been *outlined* into their own separate functions. (For example, Fig. 6.1 would be treated as three function definitions, one for the `work` function and one each for the two task bodies.) We assume that every function is call-graph reachable from $\mathcal{S}$ and that every node in a flowgraph is reachable within its associated function.

Each flowgraph is a tuple (N, E, s, e) with nodes N, edges E, entry node s and exit node e. For a given function $f \in \mathcal{F}$ we write $\mathit{start}(f) = s$, $\mathit{end}(f) = e$, $\mathit{Nodes}(f) = N$ and $\mathit{Edges}(f) = E$. Our analysis is not concerned with detailed intraprocedural execution, so control flow is considered non-deterministic along edges in E, and local variables are summarised by their total size, $\mathit{frame}(f)$.

Flowgraph nodes n are labelled with instructions $\mathit{instr}(n)$. These form four classes: calls, spawns, syncs and local computation. Given $f \in \mathcal{F}$ we write $\mathit{Calls}(f)$ (resp. $\mathit{Spawns}(f)$, $\mathit{Syncs}(f)$) for the subset of $\mathit{Nodes}(f)$ labelled with function calls (resp. task spawns, task syncs). Additionally, provided $\mathit{instr}(n)$ calls or spawns function g, we write $\mathit{func}(n) = g$.

6.1.2 Paths, synchronising instructions and the call graph

Paths

A *path* through a function f is an edge-respecting sequence of nodes $(n_0, \ldots, n_k)$ in $\mathit{body}(f)$. The set of all paths between nodes n and m is

$$\mathit{Paths}(n, m) = \{(l_0, \ldots, l_k) \mid l_0 = n \wedge l_k = m \wedge \forall\, 0 \leq i < k.\ (l_i, l_{i+1}) \in \mathit{Edges}(f)\}$$

Notation: $\quad \mathit{Paths}(n, \_) = \bigcup_m \mathit{Paths}(n, m) \qquad \mathit{Paths}(\_, n) = \bigcup_m \mathit{Paths}(m, n)$

Synchronising instructions

A *synchronising instruction* is one whose execution necessarily involves the execution of a sync instruction. These are either sync instructions themselves or calls to functions with a synchronising instruction on every possible path. We define the sets of synchronising instructions, one for each function, as the smallest sets closed under the rules:

$$\begin{aligned}
\mathit{Synchronising}(f) &\supseteq \mathit{Syncs}(f) \\
\mathit{Synchronising}(f) &\supseteq \Big\{ n \in \mathit{Calls}(f) \mid g = \mathit{func}(n) \wedge \\
&\qquad \forall\, (m_0, \ldots, m_k) \in \mathit{Paths}(\mathit{start}(g), \mathit{end}(g)) \\
&\qquad\quad \exists\, 0 \leq i \leq k.\ m_i \in \mathit{Synchronising}(g) \Big\}
\end{aligned}$$

Unsynchronised paths

An unsynchronised path is a path that may pass through no synchronising instructions. We define the set of unsynchronised paths between two instructions of a function f as follows:

$$\mathit{Upaths}(n, m) = \{(l_0, \ldots, l_k) \in \mathit{Paths}(n, m) \mid \forall\, 0 < i < k.\ l_i \notin \mathit{Synchronising}(f)\}$$

Notation: $\quad \mathit{Upaths}(n, \_) = \bigcup_m \mathit{Upaths}(n, m) \quad \mathit{Upaths}(\_, n) = \bigcup_m \mathit{Upaths}(m, n)$

Call graph

The call graph is a relation *CallGraph* on instructions:

$$\mathit{CallGraph}(n, m) \quad \overset{\text{def}}{\Leftrightarrow} \quad m \quad \in \quad \mathit{Calls}(f) \ \cup \ \mathit{Spawns}(f) \qquad \text{where } f \quad = \quad \mathit{func}(n)$$

We use this relation on spawn and call instructions to order merged sets M and unguarded sets U by dominance (rooted in $\mathcal{S}$, the set of program entry points).

6.2 Stack sizes

The safety of merging stacks depends on the potential size of those stacks at different points in a program's execution. We represent the potential size of a stack by $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$, writing $\sqsubseteq$ for its usual order $\leq_\mathbb{N}$ extended with $(\forall z \in \mathbb{N}^\infty)\ z \sqsubseteq \infty$. Note that $(\mathbb{N}^\infty, \sqsubseteq)$ is a complete lattice. To emphasise this, we will often represent 0 by the symbol $\bot$ and ∞ by the symbol $\top$. The join of this lattice ($\sqcup$) is max and the meet ($\sqcap$) is min.

We use this lattice as the basis for implication programs, using literals of the form:

$$L ::= \neg\, L \mid {\sim} L \mid L + L \mid A$$

We use the usual addition operator extended such that $(\forall z \in \mathbb{N}^\infty)\ z + \infty = \infty + z = \infty$.

There are natural definitions for both implication and difference operators on this lattice[1]:

$$\forall z_1, z_2 \in \mathbb{N}^\infty.\quad z_1 \to z_2 \ \overset{\text{def}}{=}\ \begin{cases} z_2 & \text{if } z_2 \sqsubset z_1 \\ \top & \text{otherwise} \end{cases}$$

$$\forall z_1, z_2 \in \mathbb{N}^\infty.\quad z_1 \setminus z_2 \ \overset{\text{def}}{=}\ \begin{cases} z_1 & \text{if } z_2 \sqsubset z_1 \\ \bot & \text{otherwise} \end{cases}$$

Both operators can be used to define pseudo-complement operations:

$$\forall z \in \mathbb{N}^\infty.\quad \neg z \ \overset{\text{def}}{=}\ z \to \bot$$
$$\forall z \in \mathbb{N}^\infty.\quad {\sim} z \ \overset{\text{def}}{=}\ \top \setminus z$$

To distinguish them we will call $\neg$ the *complement* and $\sim$ the *supplement*.

The complement gives $\top$ when applied to 0, and $\bot$ otherwise. We use it conveniently to mean "equals zero". The supplement gives $\bot$ when applied to ∞, and $\top$ otherwise. We use it conveniently to mean "is not ∞". Note that both are anti-monotonic, so they form negative literals in our implication programs.

[1] This follows from $\mathbb{N}^\infty$ being a bi-Heyting algebra—both it and its dual are Heyting algebras

6.3 Stack size analysis using implication programs

This section formulates the stack size analysis as an implication program in a logic using $\mathbb{N}^\infty$ as logic values. Although predicates in the implication program are written as having parameters, these parameters are all constants rather than run-time variables as could be found in Prolog. We emphasise this by writing parameters within $\langle\,\rangle$ instead of ( ). The framework is monotonic in that only conjunction (min), disjunction (max) and sum are used (we address the benefits in expressiveness and efficiency of using *general* implication programs in Section 6.4).

We do not analyse OpenMP programs in isolation, but rather in a context of a choice of merged set M and unguarded set U . Hence the result of analysing an OpenMP program is an implication program $P_{(M,U)}$.

Only some choices of M and U are safe and of these we wish to choose a 'best' solution (Section 6.3.3). Finally, we show how a context-sensitive variant of the analysis naturally follows (Section 6.3.5).

This section focuses on ease of expression and does not address efficiency, or even computability (note that the analyses here can produce infinite logic programs—Section 6.6 shows that these are equivalent to finite logic programs).

We represent the amount of stack space that may be required by a function at different points in its execution by four separate values:

Total Size An upper bound on the total amount of stack space that may be used during a function's execution. This includes the space used by any child functions that it calls, and the space used by any child tasks that it spawns whose stacks have been merged.

Post Size An upper bound on the amount of stack space that the function may use after it returns[2]. This size represents how the function may interfere with functions or tasks executed after it has finished. It includes the space used by any merged child tasks that it spawns whose execution may not have completed when the function returns.

Pre Size An upper bound on the amount of stack space that the function may use while an existing child task is still executing. This size represents how the function may interfere with tasks spawned before it started executing. It is similar to the total size, but includes neither tasks whose stacks are merged guarded nor any space used after the execution of a sync instruction.

[2]In task-based systems like Cilk this value is always zero because all tasks wait for their children to complete, but this is not the case in OpenMP.

```
void foo(...)
{
#pragma omp task
    bar(...);

#pragma omp taskwait

#pragma omp task
    baz(...);
}
```

Size	*Value*
Total	$\mathit{frame}(\texttt{foo}) + \big(\mathit{frame}(\texttt{bar}) \sqcup \mathit{frame}(\texttt{baz})\big)$
Post	$\mathit{frame}(\texttt{baz})$
Pre	$\mathit{frame}(\texttt{foo}) + \mathit{frame}(\texttt{bar})$
Through	0

Figure 6.3: Example of different stack sizes.

Through Size An upper bound on the amount of stack space that the function may use after it returns, while an existing child task is still executing. This size represents how the function may simultaneously interfere with tasks spawned before it started executing, and functions or tasks executed after it has finished. It is similar to the post size, but includes neither tasks whose stacks are merged guarded nor space used after the execution of a sync instruction.

For example, consider the program in Fig. 6.3. If we assume that the spawns of `bar()` and `baz()` are merged unguarded then the sizes are as shown on the right-hand side. We also extend these size definitions to apply to individual instructions, for instance the *total size* of a call instruction is an upper bound on the total amount of stack space that may be used during that call's execution.

We represent these sizes with the predicate symbols `TotalSize`, `PostSize`, `PreSize` and `ThroughSize` parameterised with function names or instruction nodes. The next two subsections describe the rules that make up $P_{(M,U)}$.

6.3.1 Rules for functions

Total size

Each function's total size must be greater than its stack frame plus the total size of any of its individual instructions. We can represent this by the following rule family:

$$[f \in \mathcal{F},\ n \in \mathit{Nodes}(f)] \quad \texttt{TotalSize}\langle f \rangle \longleftarrow \mathit{frame}(f) + \texttt{TotalSize}\langle n \rangle$$

The notation here $[f \in \mathcal{F}]$ represents a meta-level 'for all', in that one rule is generated for every function f (and in this case for each node n).

The above rules ensure that a function's total size is greater than the total size of any of its instructions executing on their own. A function's total size must also be greater

than any combination of its instructions that may use stack space simultaneously. This can be represented by the following rule family:

$$[f \in \mathcal{F},\ n \in \mathit{Nodes}(f),\ (m_0, \ldots, m_k) \in \mathit{Upaths}(\_, n)]$$
$$\texttt{TotalSize}\langle f \rangle \longleftarrow \mathit{frame}(f) + \texttt{PostSize}\langle m_0 \rangle + \sum_{0<i<k} \texttt{ThroughSize}\langle m_i \rangle + \texttt{PreSize}\langle m_k \rangle$$

Post size, pre size and through size

A function's post size must be greater than the post size of any combination of its instructions that may use stack space simultaneously, and which lie on an unsynchronised path to the function's exit. A function's pre size must be greater than its stack frame plus the pre size of any combination of its instructions that may use stack space simultaneously, and which lie on an unsynchronised path from the function's entry. A function's through size must be greater than the through size of any combination of its instructions that may use stack space simultaneously, and which lie on an unsynchronised path from the function's entry to its exit. These observations encode directly as rule families:

$$[f \in \mathcal{F},\ (n_0, \ldots, n_k) \in \mathit{Upaths}(\_, \mathit{end}(f))]$$
$$\texttt{PostSize}\langle f \rangle \longleftarrow \texttt{PostSize}\langle n_0 \rangle + \sum_{0<i\leq k} \texttt{ThroughSize}\langle n_i \rangle$$

$$[f \in \mathcal{F},\ (n_0, \ldots, n_k) \in \mathit{Upaths}(\mathit{start}(f), \_)]$$
$$\texttt{PreSize}\langle f \rangle \longleftarrow \mathit{frame}(f) + \sum_{0\leq i<k} \texttt{ThroughSize}\langle n_i \rangle + \texttt{PreSize}\langle n_k \rangle$$

$$[f \in \mathcal{F},\ (n_0, \ldots, n_k) \in \mathit{Upaths}(\mathit{start}(f), \mathit{end}(f))]$$
$$\texttt{ThroughSize}\langle f \rangle \longleftarrow \sum_{0\leq i\leq k} \texttt{ThroughSize}\langle n_i \rangle$$

6.3.2 Rules for instructions

Call instructions

Since all call instructions use the stack of the caller, their sizes must be greater than the corresponding size of the functions they call. This is represented by the following rule

family:

$$
\begin{aligned}
&[f \in \mathcal{F},\ n \in \mathit{Calls}(f)] \\
&\quad \texttt{TotalSize}\langle n\rangle \longleftarrow \texttt{TotalSize}\langle \mathit{func}(n)\rangle \\
&\quad \texttt{PreSize}\langle n\rangle \longleftarrow \texttt{PreSize}\langle \mathit{func}(n)\rangle \\
&\quad \texttt{PostSize}\langle n\rangle \longleftarrow \texttt{PostSize}\langle \mathit{func}(n)\rangle \\
&\quad \texttt{ThroughSize}\langle n\rangle \longleftarrow \texttt{ThroughSize}\langle \mathit{func}(n)\rangle
\end{aligned}
$$

Spawn instructions

For any *merged* spawn instruction, the spawned task may use the stack of the caller and may be deferred until some point after the spawn instruction has completed. This means that both the total size and post size of the instruction must be greater than the total size of the spawned task. If the spawn instruction is merged *unguarded* then the pre size and through size of the instruction must also be greater than the size of the spawned task. This leads to the following rule families:

$$
\begin{aligned}
[n \in M] \qquad & \texttt{TotalSize}\langle n\rangle \longleftarrow \texttt{TotalSize}\langle \mathit{func}(n)\rangle \\
& \texttt{PostSize}\langle n\rangle \longleftarrow \texttt{TotalSize}\langle \mathit{func}(n)\rangle \\
[n \in U] \qquad & \texttt{PreSize}\langle n\rangle \longleftarrow \texttt{TotalSize}\langle \mathit{func}(n)\rangle \\
& \texttt{ThroughSize}\langle n\rangle \longleftarrow \texttt{TotalSize}\langle \mathit{func}(n)\rangle
\end{aligned}
$$

6.3.3 Optimising merged and unguarded sets

A solution to our analysis is a pair (M, U) of merged set M and unguarded set U. Our analysis must choose the "best" safe solution. We now explore: (i) which solutions are safe, and (ii) which safe solution is the "best".

Which solutions are safe?

Using the implication program $P_{(M,U)}$, we can now decide whether a particular solution (M, U) is a safe choice of merged and unguarded sets. There are two situations that we consider unsafe:

1. A child task using its parent task's stack after that parent task has finished.

2. Two tasks simultaneously using unbounded amounts of the same stack.

In situation 1 the parent task may delete the stack after it has finished while the child task is still using it. In situation 2 both tasks may try to push and pop data onto the top of the stack concurrently, which our optimisation does not support (it would require

synchronisation). Note that it would be safe if one of the tasks only required a bounded amount of space because then that much space could be reserved on the stack in advance.

Situation 1 is equivalent to spawning a function with a non-zero post size. To avoid this situation, under the least model of $P_{(M,U)}$ for a safe solution (M, U), the following family of formulae must all evaluate to $\top$:

$$[f \in \mathcal{F},\ n \in \mathit{Spawns}(f)] \quad \neg\, \texttt{PostSize}\langle \mathit{func}(n) \rangle$$

Note that here the $\neg$ operator conveniently means "equals zero".

Situation 2 is equivalent to some of a task's child tasks using unbounded stack space whilst at the same time the parent task (and possibly some of its other child tasks) also uses unbounded stack space. To avoid this situation, under the least model of $P_{(M,U)}$ for a safe solution (M, U), the following family of formulae must all evaluate to $\top$:

$$[f \in \mathcal{F}, \quad n_0 \in \mathit{Nodes}(f), \quad (n_0, \ldots, n_k, m_0, \ldots, m_l) \in \mathit{Upaths}(n_0, \lrcorner)]$$

$$\sim \left(\sum_{0<i\leq k} \begin{array}{l} \texttt{ThroughSize}\langle n_i \rangle \\ + \texttt{PostSize}\langle n_0 \rangle \end{array} \prod \sum_{0\leq i<l} \begin{array}{l} \texttt{ThroughSize}\langle m_i \rangle \\ + \texttt{PreSize}\langle m_l \rangle \end{array} \right)$$

These formulae mean that the tasks spawned by instructions $n_0, \ldots, n_k$, and the instructions $m_0, \ldots, m_l$ which may execute simultaneously with them, cannot both use unbounded stack space ($\sim$ means "is not unbounded").

If both of these conditions are met then we say that a solution (M, U) is a safe choice for merged and unguarded sets.

Which safe solution is the "best"?

Our aim is to merge as many stacks at run time as we can, and for as many as possible of those merges to be unguarded. It is also more important to increase the total number of stacks merged than to increase the number of stacks merged unguarded. Hence we order solutions lexicographically:

$$(M, U) \sqsubseteq (M', U') \quad \Leftrightarrow \quad M \subset M' \vee (M = M' \wedge U \subseteq U')$$

We would like to choose as the result of our analysis the greatest safe solution according to this ordering. However, not every program has a unique greatest safe solution. Every program does have a unique set of maximal safe solutions, whose members are each either greater than or incomparable with all other safe solutions. In order to chose the best solution from the set of maximal safe solutions, we must use heuristics.

One simple heuristic is preferring to merge spawns that are further from the root of the run-time call graph, because they are likely to be executed more often. We can approximate this using the static call graph by preferring maximal solution (M, U) over

maximal solution (M', U') if, letting $Lost = M' \setminus M$ and $Gained = M \setminus M'$, we have that every node $n \in Lost$ dominates (in *CallGraph* with respect to paths starting at $\mathcal{S}$) every node $m \in Gained$. Note that this is a heuristic for choosing between maximal solutions, rather than an ordering on all solutions, because the reasoning behind it assumes that there are no safe solutions greater than (M, U) or (M', U').[3]

Even with this heuristic programs may still have several equally preferred safe solutions. We call such solutions *optimal*. In Section 6.3.5 we discuss context sensitivity; the context-sensitive version of our analysis has only a single optimal solution.

6.3.4 Finding an optimal solution

Finding the greatest safe solution according to both the ordering on solutions and our call-graph heuristic is a kind of *constraint optimisation problem* (COP).

A traditional COP consists of a constraint problem (often represented by a set of variables with domains and a set of constraints on those variables) and an objective function. The aim is to optimise the objective function while obeying the constraints. In our case, the safety conditions are our constraint problem, and instead of an objective function we have the ordering on solutions and our call-graph heuristic.

Many COPs are inherently non-monotonic: as the variables are increased the value of the objective function increases, until a constraint is broken – which is equivalent to the objective function being zero. This is true of finding an optimal solution for our analysis: we prefer solutions which merge more spawn instructions, but as more spawn instructions are merged the sizes increase, and as the sizes increase the solution becomes more likely to be unsafe.

COPs are usually solved using some form of backtracking search. This tries to incrementally build solutions, abandoning each partial candidate as soon as it determines that it cannot possibly be part of a valid solution. Such an approach can easily be adopted for finding the optimal solution to our analysis: keep merging more spawn instructions until it is unsafe, then backtrack and try merging some different spawn instructions.

The search space of a COP is exponential in the number of variables, and our problem requires us to recompute the stack sizes for each solution that we try. A naive search could be very expensive, however there are two simple methods for improving our search:

1. We can use the stack sizes to prune the search tree. For instance, if the current solution causes two tasks to have unbounded size and their spawn instructions have an unsynchronised path between them, then there is no point in trying a solution that merges both of them unguarded.

[3]Including this heuristic as part of the ordering on all solutions can lead to cycles in the ordering.

2. Instead of recomputing the stack sizes for each possible solution, we can start from the stack sizes of a similar solution and just compute the changes.

We shall see in Section 6.4 that this approach can be encoded as a general implication program.

6.3.5 Adding context-sensitivity

It is clear from our safety conditions that whether a spawn can be safely merged is *context-sensitive*. By context-sensitive we mean that it does not just depend on the details of the function that contains it, but also on the details of the function that called that function, and the details of the function that called that second function, and so on.

While the safety conditions are context-sensitive, the optimisation and analysis described so far are context-insensitive. This means that some stacks will not be merged even though it would be safe to do so, because it would not have been safe if the function had been called from a different context.

In order to allow more spawn points to be merged at run time, we can make the optimisation and analysis context-sensitive. This involves making the behaviour of functions depend on the context that called them.

In our model we achieve this by creating multiple versions of the same function for different contexts, but in practice we simply add extra arguments containing information about the calling context.

A recursive program may have an infinite number of contexts, however we are only interested in the restrictions placed on a function by its context. These restrictions can be represented by four boolean values (see Section 6.4.1), so we can also represent our context by four boolean values.

Making our optimisation context-sensitive is very cheap; other than the extra context arguments it only requires a few additional logic operations before some calls and stack frame allocations. A simple analysis can detect and remove unused or unnecessary context arguments.

The aim of making the optimisation context-sensitive is to separate run-time function calls when they are called from contexts which require them to merge fewer spawns. This means that the context that a call is in depends on the stack sizes of related instructions, but the stack sizes of instructions depend on the contexts that they are given. This recursive relationship is also non-monotonic: as stack sizes increase more calls are assigned more restrictive contexts, but as more calls are placed in more restrictive contexts stack sizes decrease.

This situation is very similar to the one that exists between stack sizes and the merged and unguarded sets. Similarly it can be resolved using a backtracking search and it can be encoded as a general implication program.

6.4 The analysis as a general implication program

This section describes how to represent the context-sensitive version of our analysis as a single general implication program—the idea is that meta-level constraints on U and M are now expressed within the logic using negation.

6.4.1 Stack size restrictions

We represent the safety conditions, within this general implication program, as various restrictions on individual stack sizes. There are four kinds of restriction:

1. Restricting the post size to 0. This is equivalent to making the *complement* of the post size $\top$.

2. Restricting the post size to be not unbounded. This is equivalent to making the *supplement* of the post size $\top$.

3. Restricting the pre size to be not unbounded. This is equivalent to making the *supplement* of the pre size $\top$.

4. Restricting the through size to be 0. This is equivalent to making the *complement* of the through size $\top$.

We place these restrictions on instructions using the predicates `CompPostSize`, `SuppPostSize`, `SuppPreSize` and `CompThroughSize`. We do not need to have explicit predicates to place these restrictions on functions, because we use these restrictions as the contexts for functions. Each function f is replaced by 16 versions of the function $f_{(cr,sr,sg,cgr)}$, one for each possible combination of restrictions.

Note that these restrictions can only affect stack sizes by preventing or guarding merges. So a function whose pre size is restricted may still have unbounded pre size if that unboundedness is caused by ordinary recursive calls, rather than by recursive spawns.

6.4.2 Restriction rules

The `CompPostSize` restriction is placed on functions that are spawned, to prevent our first safety condition from being broken. It is propagated by the rule family:

$$[f \in \mathcal{F}, \quad \gamma \in (\{\mathtt{T}\} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}), \quad (n_0, \ldots, n_k) \in \mathit{Upaths}(\_, \mathit{end}(f_\gamma))]$$
$$\mathtt{CompPostSize}\langle n_0 \rangle \longleftarrow \top$$

The other restrictions are used to prevent our second safety condition from being broken. They are propagated by the rule families:

$$[f \in \mathcal{F}, \quad \gamma \in (\mathbb{B} \times \{\mathtt{T}\} \times \mathbb{B} \times \mathbb{B}), \quad (n_0, \ldots, n_k) \in \mathit{Upaths}(\llcorner, \mathit{end}(f_\gamma))]$$
$$\mathtt{SuppPostSize}\langle n_0 \rangle \longleftarrow \top$$

$$[f \in \mathcal{F}, \quad \gamma \in (\mathbb{B} \times \mathbb{B} \times \{\mathtt{T}\} \times \mathbb{B}), \quad (n_0, \ldots, n_k) \in \mathit{Upaths}(\mathit{start}(f_\gamma), \llcorner)]$$
$$\mathtt{SuppPreSize}\langle n_k \rangle \longleftarrow \top$$

$$[f \in \mathcal{F},\ \gamma \in (\mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \{\mathtt{T}\}),$$
$$(n_0, \ldots, n_k) \in \mathit{Upaths}(\mathit{start}(f_\gamma), \mathit{end}(f_\gamma)),\ 0 \leq i \leq k]$$
$$\mathtt{CompThroughSize}\langle n_i \rangle \longleftarrow \top$$

The `CompThroughSize` restriction is used to prevent loops of instructions from using unbounded stack space. It is enforced by the rule:

$$[f \in \mathcal{F},\ \gamma \in (\mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}),\ n \in \mathit{Nodes}(f_\gamma),\ (m_0, \ldots, m_k) \in \mathit{Upaths}(n, n)]$$
$$\mathtt{CompThroughSize}\langle m_0 \rangle \longleftarrow \top$$

The `SuppPreSize` restriction is used to prevent spawn instructions from being merged unguarded if they are unbounded and preceded by a merged spawn instruction which is also unbounded. It is enforced by the rule family:

$$[f \in \mathcal{F},\ \gamma \in (\mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}),\ n \in \mathit{Nodes}(f_\gamma),\ (m_0, \ldots, m_k) \in \mathit{Upaths}(\llcorner, n)]$$
$$\mathtt{SuppPreSize}\langle m_k \rangle \longleftarrow \sim\!\sim \mathtt{PostSize}\langle m_0 \rangle$$

The `SuppPostSize` restriction is used to prevent spawn instructions from being merged if they are unbounded and followed by a call to a function that may use unbounded stack space (even if all its spawns are merged guarded). Note that we do not prevent a spawn from being merged due to a later unguarded spawn, because we prefer to make the later spawn guarded. This restriction is enforced by the rule family:

$$[f \in \mathcal{F},\ \gamma \in (\mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}),\ n \in \mathit{Nodes}(f_\gamma),\ (m_0, \ldots, m_k) \in \mathit{Upaths}(n, \llcorner),$$
$$g = \mathit{func}(m_k),\ cr \in \mathbb{B},\ sr \in \mathbb{B},\ crg \in \mathbb{B}]$$
$$\begin{aligned} \mathtt{SuppPostSize}\langle m_0 \rangle \longleftarrow\ & \sim\!\sim \mathtt{PreSize}\langle g_{(cr,sr,\mathtt{T},crg)} \rangle\ , \\ & \mathit{lit}(cr,\ \mathtt{CompPostSize}\langle m_k \rangle)\ , \\ & \mathit{lit}(sr,\ \mathtt{SuppPostSize}\langle m_k \rangle)\ , \\ & \mathit{lit}(crg,\ \mathtt{CompThroughSize}\langle m_k \rangle) \end{aligned}$$

where $\mathit{lit}(b, A)$ is a macro for $\begin{cases} \neg\neg\, A & \text{if } b = \mathtt{T} \\ \neg\, A & \text{if } b = \mathtt{F} \end{cases}$

Note that the *lit* macro used in generating the above rules converts the restriction predicates into booleans that can be used as the contexts for functions.

We apply the supplement restrictions to spawns via complement restrictions using the following rules. This is equivalent to preventing unbounded sizes by forcing those sizes to be zero (i.e. preventing the stacks from merging).

$$
\begin{array}{l}
[f \in \mathcal{F},\ \gamma \in (\mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}),\ n \in \mathit{Spawns}(f_\gamma),\ g = \mathit{func}(n)] \\
\quad \texttt{CompPostSize}\langle n \rangle \longleftarrow\ \texttt{SuppPostSize}\langle n \rangle\ , \\
\qquad\qquad\qquad\qquad \sim\!\sim \texttt{TotalSize}\langle g_{(\texttt{T,T,F,F})} \rangle \\
\quad \texttt{CompPreSize}\langle n \rangle \longleftarrow\ \texttt{SuppPreSize}\langle n \rangle\ , \\
\qquad\qquad\qquad\qquad \sim\!\sim \texttt{TotalSize}\langle g_{(\texttt{T,T,F,F})} \rangle
\end{array}
$$

We refer to these rules as the *bounding rules*.

6.4.3 Other rules

The rules for the stack sizes of spawn instructions are as follows:

$$
\begin{array}{l}
[f \in \mathcal{F},\ \gamma \in (\mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}),\ n \in \mathit{Spawns}(f_\gamma),\ g = \mathit{func}(n)] \\
\quad \texttt{TotalSize}\langle n \rangle \longleftarrow \texttt{TotalSize}\langle g_{(\texttt{T,T,F,F})} \rangle\ ,\ \neg\, \texttt{CompPostSize}\langle n \rangle \\
\quad \texttt{PostSize}\langle n \rangle \longleftarrow \texttt{TotalSize}\langle g_{(\texttt{T,T,F,F})} \rangle\ ,\ \neg\, \texttt{CompPostSize}\langle n \rangle \\
\quad \texttt{PreSize}\langle n \rangle \longleftarrow\ \texttt{TotalSize}\langle g_{(\texttt{T,T,F,F})} \rangle\ ,\ \neg\, \texttt{CompPostSize}\langle n \rangle\ , \\
\qquad\qquad \neg\, \texttt{CompPreSize}\langle n \rangle\ ,\ \ \neg\, \texttt{CompThroughSize}\langle n \rangle \\
\quad \texttt{ThroughSize}\langle n \rangle \longleftarrow\ \texttt{TotalSize}\langle g_{(\texttt{T,T,F,F})} \rangle\ ,\ \neg\, \texttt{CompPostSize}\langle n \rangle\ , \\
\qquad\qquad\qquad \neg\, \texttt{CompPreSize}\langle n \rangle\ ,\ \ \neg\, \texttt{CompThroughSize}\langle n \rangle
\end{array}
$$

The remaining stack size rules are based on those in Section 6.3 and are shown in Fig. 6.4 and Fig. 6.5.

6.4.4 Extracting solutions

Given a stable model of the rules described in this section, we can extract a solution that is equivalent to the solution that we would have obtained using the methods suggested in Section 6.3.3. The merged set M and unguarded merge set U are given by:

$$
\begin{array}{rcl}
M & = & \{n \mid \texttt{TotalSize}\langle g_{(\texttt{T,T,F,F})} \rangle \sqsubseteq \texttt{PostSize}\langle n \rangle,\ g = \mathit{func}(n)\} \\
U & = & \{n \mid \texttt{TotalSize}\langle g_{(\texttt{T,T,F,F})} \rangle \sqsubseteq \texttt{ThroughSize}\langle n \rangle,\ g = \mathit{func}(n)\}
\end{array}
$$

$$[f \in \mathcal{F},\ \gamma \in (\mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}),\ n \in \mathit{Nodes}(f_\gamma)]$$

$$\texttt{TotalSize}\langle f_\gamma \rangle \longleftarrow \mathit{frame}(f_\gamma) + \texttt{TotalSize}\langle n \rangle$$

$$[f \in \mathcal{F},\ \gamma \in (\mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}),\ n \in \mathit{Nodes}(f_\gamma),\ (m_0, \ldots, m_k) \in \mathit{Upaths}(\llcorner, n)]$$

$$\begin{aligned}\texttt{TotalSize}\langle f_\gamma \rangle \longleftarrow & \mathit{frame}(f_\gamma) + \texttt{PostSize}\langle m_0 \rangle \\ & + \sum_{0<i<k} \texttt{ThroughSize}\langle m_i \rangle \\ & + \texttt{PreSize}\langle m_k \rangle\end{aligned}$$

$$[f \in \mathcal{F},\ \gamma \in (\mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}),\ (n_0, \ldots, n_k) \in \mathit{Upaths}(\llcorner, \mathit{end}(f_\gamma))]$$

$$\begin{aligned}\texttt{PostSize}\langle f_\gamma \rangle \longleftarrow & \texttt{PostSize}\langle n_0 \rangle \\ & + \sum_{0<i\leq k} \texttt{ThroughSize}\langle n_i \rangle\end{aligned}$$

$$[f \in \mathcal{F},\ \gamma \in (\mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}),\ (n_0, \ldots, n_k) \in \mathit{Upaths}(\mathit{start}(f_\gamma), \llcorner)]$$

$$\begin{aligned}\texttt{PreSize}\langle f_\gamma \rangle \longleftarrow & \mathit{frame}(f_\gamma) + \sum_{0\leq i<k} \texttt{ThroughSize}\langle n_i \rangle \\ & + \texttt{PreSize}\langle n_k \rangle\end{aligned}$$

$$[f \in \mathcal{F},\ \gamma \in (\mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}),\ (n_0, \ldots, n_k) \in \mathit{Upaths}(\mathit{start}(f_\gamma), \mathit{end}(f_\gamma))]$$

$$\texttt{ThroughSize}\langle f_\gamma \rangle \longleftarrow \sum_{0\leq i\leq k} \texttt{ThroughSize}\langle n_i \rangle$$

Figure 6.4: Rules for Functions

$$
\begin{array}{l}
[f \in \mathcal{F},\ \gamma \in (\mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}),\ n \in \mathit{Calls}(f_\gamma),\ g = \mathit{func}(n), \\
\quad cr \in \mathbb{B},\ sr \in \mathbb{B},\ sg \in \mathbb{B},\ crg \in \mathbb{B}] \\
\qquad \texttt{TotalSize}\langle n \rangle \longleftarrow\ \texttt{TotalSize}\langle g_{(cr,sr,sg,crg)} \rangle\ , \\
\qquad\qquad\qquad\qquad \mathit{lit}(cr,\ \texttt{CompPostSize}\langle n \rangle)\ , \\
\qquad\qquad\qquad\qquad \mathit{lit}(sr,\ \texttt{SuppPostSize}\langle n \rangle)\ , \\
\qquad\qquad\qquad\qquad \mathit{lit}(sr,\ \texttt{SuppPreSize}\langle n \rangle)\ , \\
\qquad\qquad\qquad\qquad \mathit{lit}(crg,\ \texttt{CompThroughSize}\langle n \rangle) \\
\qquad \texttt{PreSize}\langle n \rangle \longleftarrow\ \texttt{PreSize}\langle g_{(cr,sr,sg,crg)} \rangle\ , \\
\qquad\qquad\qquad\qquad \mathit{lit}(cr,\ \texttt{CompPostSize}\langle n \rangle)\ , \\
\qquad\qquad\qquad\qquad \mathit{lit}(sr,\ \texttt{SuppPostSize}\langle n \rangle)\ , \\
\qquad\qquad\qquad\qquad \mathit{lit}(sr,\ \texttt{SuppPreSize}\langle n \rangle)\ , \\
\qquad\qquad\qquad\qquad \mathit{lit}(crg,\ \texttt{CompThroughSize}\langle n \rangle) \\
\qquad \texttt{PostSize}\langle n \rangle \longleftarrow\ \texttt{PostSize}\langle g_{(cr,sr,sg,crg)} \rangle\ , \\
\qquad\qquad\qquad\qquad \mathit{lit}(cr,\ \texttt{CompPostSize}\langle n \rangle)\ , \\
\qquad\qquad\qquad\qquad \mathit{lit}(sr,\ \texttt{SuppPostSize}\langle n \rangle)\ , \\
\qquad\qquad\qquad\qquad \mathit{lit}(sr,\ \texttt{SuppPreSize}\langle n \rangle)\ , \\
\qquad\qquad\qquad\qquad \mathit{lit}(crg,\ \texttt{CompThroughSize}\langle n \rangle) \\
\qquad \texttt{ThroughSize}\langle n \rangle \longleftarrow\ \texttt{ThroughSize}\langle g_{(cr,sr,sg,crg)} \rangle\ , \\
\qquad\qquad\qquad\qquad \mathit{lit}(cr,\ \texttt{CompPostSize}\langle n \rangle)\ , \\
\qquad\qquad\qquad\qquad \mathit{lit}(sr,\ \texttt{SuppPostSize}\langle n \rangle)\ , \\
\qquad\qquad\qquad\qquad \mathit{lit}(sr,\ \texttt{SuppPreSize}\langle n \rangle)\ , \\
\qquad\qquad\qquad\qquad \mathit{lit}(crg,\ \texttt{CompThroughSize}\langle n \rangle) \\
\text{where } \mathit{lit}(b, A) \text{ is a macro for } \begin{cases} \neg\neg A & \text{if } b = \texttt{T} \\ \neg A & \text{if } b = \texttt{F} \end{cases}
\end{array}
$$

Figure 6.5: Rules for Call Instructions

6.5 Stratification

We could find stable models for the general implication program using backtracking algorithms similar to those used in answer set programming, based on the DPLL algorithm. However, using stratified models finds them more directly.

It is easy to see that the implication program derived in the previous section cannot be stratified. However looking at the rules we can make the following observations:

1. `CompPostSize` and `CompThroughSize` only depend negatively on other predicates via the *bounding rules*.

2. The bounding rules only apply within a function with context $(\mathtt{T},\mathtt{T},\mathtt{F},\mathtt{F})$ if that function contains an instruction with unbounded `TotalSize`, and such functions have unbounded `TotalSize` with or without the bounding rules. This means that the `TotalSize` of all functions $f_{(\mathtt{T},\mathtt{T},\mathtt{F},\mathtt{F})}$ can be calculated without the bounding rules, and in such a calculation `TotalSize` will only depend negatively on the `CompPostSize` and `CompThroughSize` predicates.

3. For any instruction n, if $\mathtt{SuppPreSize}\langle n\rangle$ equals $\top$ then the value of $\mathtt{SuppPostSize}\langle n\rangle$ will not affect the values of $\mathtt{PreSize}\langle n\rangle$ or $\mathtt{ThroughSize}\langle n\rangle$. This means that the `PreSize` of any function with a context of the form $(cr, sr, \mathtt{T}, crg)$ only depends negatively on the `TotalSize` of functions with context $(\mathtt{T},\mathtt{T},\mathtt{F},\mathtt{F})$ and on the values of `CompPostSize` and `CompThroughSize` calculated without the bounding rules.

4. If $\mathtt{ThroughSize}\langle f_{(cr,\mathtt{T},sg,crg)}\rangle \neq \mathtt{ThroughSize}\langle f_{(cr,\mathtt{F},sg,crg)}\rangle$ then $\mathtt{PostSize}\langle f_{(cr,\mathtt{T},sg,crg)}\rangle = \mathtt{PostSize}\langle f_{(cr,\mathtt{F},sg,crg)}\rangle = \top$. Therefore, for any instruction n, the value of $\mathtt{SuppPreSize}\langle n\rangle$ will not affect the values of $\mathtt{PostSize}\langle n\rangle$. This means that the `PostSize` of any node only depends negatively on the `TotalSize` of functions with context $(\mathtt{T},\mathtt{T},\mathtt{F},\mathtt{F})$ and on the values of `CompPostSize`, `CompThroughSize` and `SuppPostSize`.

This means that we can create a stratifiable general implication program by using five *layers* of the general implication program from the previous section. Each layer is a more accurate approximation of the full set of rules. All negative literals are made to refer to the literals of the previous layer, so that the program can easily be stratified.

These layers work as follows:

1. The first layer calculates the values of `CompPostSize` and `CompThroughSize` ignoring the bounding rules.

2. The second layer calculates the values of `TotalSize` for all functions with context $(\mathtt{T},\mathtt{T},\mathtt{F},\mathtt{F})$.

3. The third layer calculates the values of `SuppPostSize`.

4. The fourth layer calculates the values of `SuppPreSize`.

5. The fifth layer calculates the values of all the remaining predicates.

It can be shown that the stable models of the previous general implication program are equivalent to the stable models of this stratified general implication program. Since stratifiable general implication programs have a unique stable model, this shows that our analysis has a unique solution.

6.6 Complexity of the analysis

The unique stable model of a stratified general implication program $P_1 \cup \cdots \cup P_k$ is the same as its standard model. This standard model can be computed in polynomial time if the least fixed points of each T_{P_i} can be computed in polynomial time.

While some of the rule families of our analysis contain an infinite number of rules this was only for presentation. They can also be expressed by a finite number of rules, using an additional predicate to represent the maximum sum of `ThroughSize` between two instructions:

$$[f \in \mathcal{F}, \quad n, m \in \mathit{Nodes}(f), \quad (l_0, \ldots, l_k) \in \mathit{Upaths}(n, m), \quad \forall\, 0 < i < k]$$
$$\texttt{PathMax}\langle n, m\rangle \longleftarrow \texttt{PathMax}\langle n, l_i\rangle + \texttt{PathMax}\langle l_i, m\rangle + \texttt{ThroughSize}\langle l_i\rangle$$

Since the number of rules is polynomial, and the operations within the rules are all polynomial time, each iteration of T_{P_i} can be computed in polynomial time.

Each possible bounded size that can be assigned to a predicate in P is uniquely determined by a set of (context-sensitive) function names and instruction nodes. Otherwise that size would include a recursive call or an unbounded iteration of spawns, and so would be $\top$. This means that the number of times a predicate can increase its size is proportional to the size of the original OpenMP program, so the least fixed points of each T_{P_i} can be computed in polynomial time.

6.7 Implementation

This analysis is implemented within EMCC as two separate analyses. The first one does a control-flow analysis to work out the unsynchronised paths in each function. The results of this analysis are attached to each spawn and call instruction using properties. The second analysis uses these properties to create the stratified set of rules representing the analysis. The iterated least model of these rules is then calculated, and spawns and calls are marked as merged and unguarded according to its results. The stack size bounds

are also attached to the spawns and calls, so that space can be reserved on stacks where necessary. Both these analyses are implemented as OCaml functors that expect functions for inspecting statements and visitors for traversing the control-flow and call graphs.

Some of the handlers used with the `Generate` module to convert the OpenMP IL into C were overridden so that the back-end produced code using stacks instead of dynamic allocation. These new handlers recognise the properties produced by the analyses and uses them to merge stacks where possible.

The execution time of this (quite naive) implementation seems to be very good, with no noticable difference in compilation times on any of the programs that we have tried. This is to be expected as we have shown the time complexity to be polynomial to the number of spawn sites and suspendable function calls, which will very small in a typical OpenMP program.

6.8 Evaluation

We compared our OpenMP implementation using this optimisation with the original version using heap-based allocation of stack frames (see Chapter 5). We also compare it with a version which uses stacks but does not apply our optimisation.

We compared the implementations using programs from the Barcelona Tasks Suite [23]: *Alignment*, *NQueens* and *Sort*. Alignment uses an iterative pattern with a parallel loop that spawns multiple tasks. The other two use recursive divide-and-conquer patterns, with each task spawning multiple tasks and then waiting for them to finish. The benchmarks were run on a server with 32 AMD Opteron processors.

The speed-ups for each benchmark are shown in Fig. 6.6. These show that the optimised version scales just as well as our heap-based version. The version using stacks without our optimisation does not scale as well because it is forced to restrict parallelism using load-based inlining.

The relative execution time of the optimised and heap versions for each benchmark is shown in Fig. 6.7. Alignment shows no difference between implementations, Sort shows an improvment of 4% using our optimisation, and NQueens shows an improvemnt of 9%.

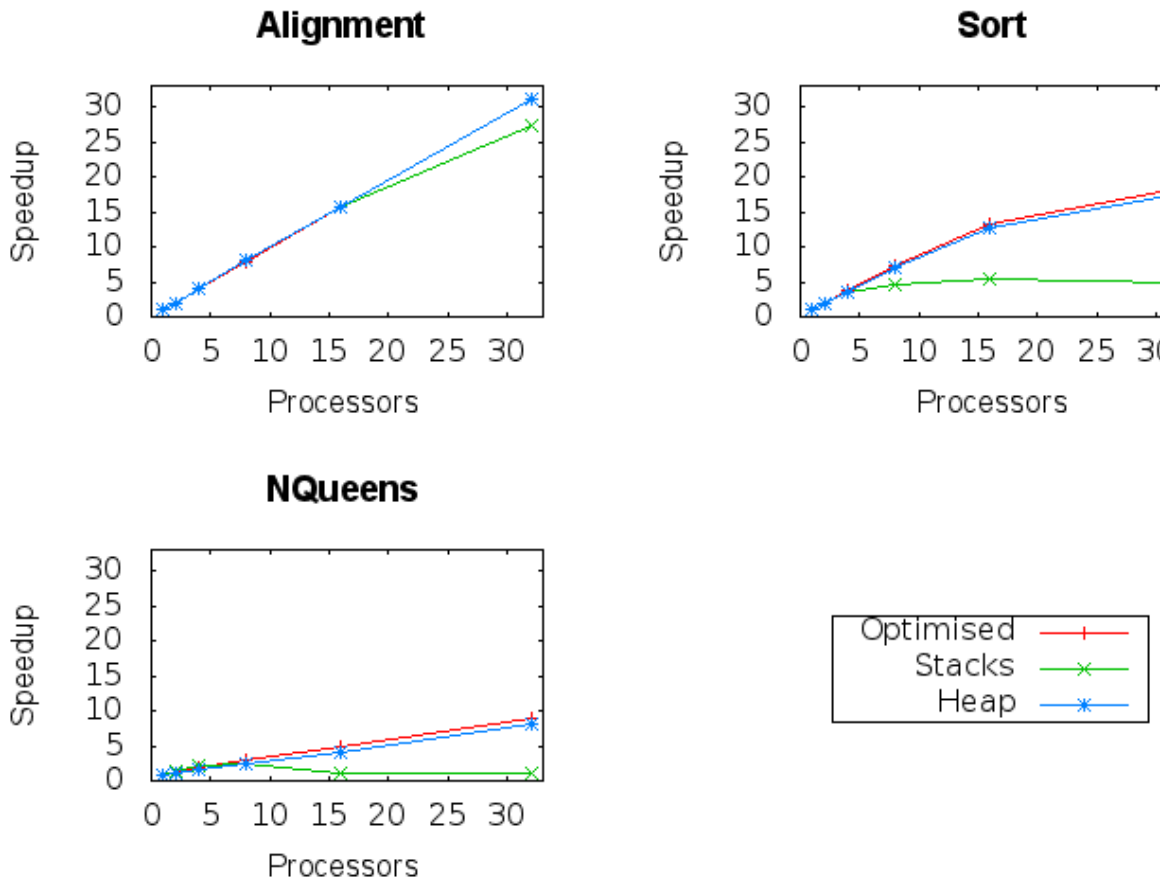


Figure 6.6: Speed-up

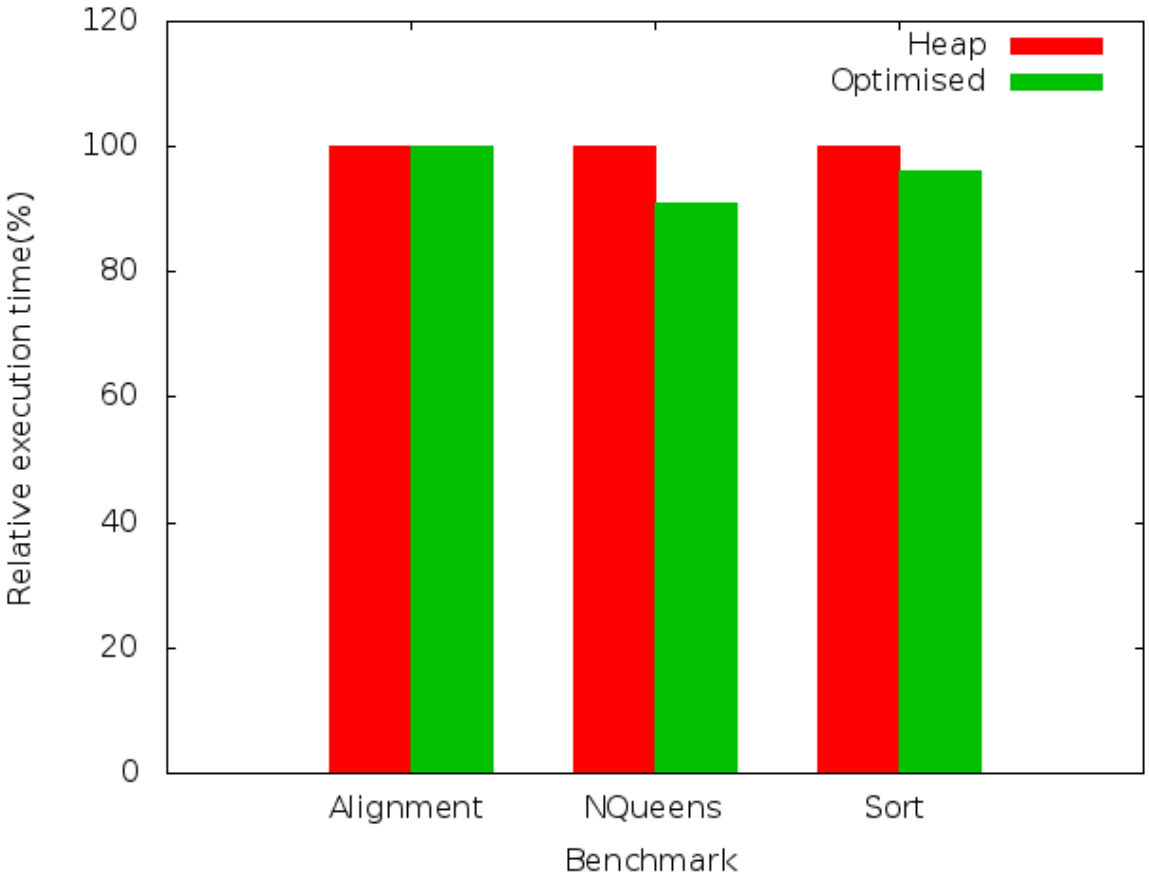


Figure 6.7: Relative execution time

6.9 Conclusion

In this chapter we have described a program analysis for OpenMP to enable tasks to share stacks for task-local memory. We have shown how a novel implication-algebra generalisation of logic programming allows a concise but easily readable encoding of the various constraints.

Using this formalism we were able to show that the analysis has a unique solution and polynomial time complexity.

We implemented this optimisation in EMCC and provided experimental results that show that it provides reasonable improvements in performance compared with the heap-based approach described in Chapter 5.

Chapter 7

Extensions to OpenMP for heterogeneous architectures

In this chapter we use EMCC to implement another language extension: we add extensions to OpenMP to support heterogeneous architectures.

Section 7.1 discusses heterogeneous architectures and the challenges they pose for OpenMP. Section 7.2 presents relevant related work. Section 7.3 describes our proposed extensions. Section 7.4 discusses how the extensions can be implemented using EMCC and our customisable run-time library. Section 7.5 presents experimental results to show some benefits of the extensions.

7.1 Heterogeneous architectures and OpenMP

Modern architectures are becoming more heterogeneous. Power dissipation issues have led chip designers to look for new ways to use the transistors at their disposal. This means multi-core chips with a greater variety of cores and increasingly complex memory systems. These modern architectures can contain a GPU or a number of slave processors, in addition to their CPUs. Developing programs for architectures containing multiple kinds of processors is a new challenge. This chapter describes the design of some extensions to OpenMP to support its implementation on heterogeneous architectures.

There are two main problems with implementing and using OpenMP on a modern heterogeneous architecture:

1. The model assumes that there is a single coherent memory space. Compiler techniques for mapping programs written for a single memory space onto architectures that have partitioned memory systems have had some success [15, 16, 29]. However it seems that these techniques will not solve the problem in the general case, and some sort of extension to OpenMP will be required.

```
void a9(int n, int m, float *a, float *b, float *y, float *z)
{
    int i;
    #pragma omp parallel
    {
        #pragma omp for nowait
        for (i=1; i<n; i++)
            b[i] = (a[i] + a[i-1]) / 2.0;
        #pragma omp for nowait
        for (i=0; i<m; i++)
            y[i] = sqrt(z[i]);
    }
}
```

Figure 7.1: First Example: Two parallel loops

2. There is no mechanism for allocating work to specific processors on an architecture. This problem is orthogonal to the first one: whatever method is used to address partitioned memory systems, if OpenMP continues to use fork/join, loop-based and task-based parallelism then it will require mechanisms for mapping these models onto heterogeneous architectures.

It is the second problem that our extensions attempt to solve. They increase the expressivity of OpenMP to give programmers control over how work is allocated on a heterogeneous architecture.

We start with some illustrative examples of how OpenMP is used in practice, adapted from those found in the OpenMP Version 3.0 Specification [61]. Fig. 7.1 shows how two loops can be divided between the threads in a team. It consists of a `parallel` construct containing two `for` constructs (each annotated with a `nowait` *clause*). Fig. 7.2 shows a how a single thread in a team can traverse a list creating one task for each node. These tasks will then be executed by the other threads in the team.

The work in these examples could be allocated onto a heterogeneous architecture in a number of ways. The simplest allocation would allow the threads in the team to be executed by any of the processors in the architecture. However, it might be more efficient to restrict the threads to a selection of the processors – perhaps those sharing a single memory unit. This would require some form of *thread mapping* extension, to allow the programmer to describe which threads in a team were restricted to which processors.

It is also possible that, in the first example, the processors best suited to execute the first loop differ from those best suited to execute the second. Perhaps there are two groups of processors, and the most efficient solution is to run the first loop on one group while the second loop runs on the other. In either case, the best allocation involves using

```
void process_list_items(node *head)
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            node *p = head;
            while (p) {
                #pragma omp task
                    process(p);
                p = p->next;
            }
        }
    }
}
```

Figure 7.2: Second Example: Using tasks to traverse a list

different threads to execute the two workshares. This would either require two separate teams running in parallel (i.e. *nested parallelism*), or workshares restricted to subsets of the threads in the team (i.e. *subteams*).

The best allocation for the second example might involve allocating the processing tasks to accelerators, while the main loop is executed on the central processor. This would require the `single` workshare to be restricted to the subset of threads executing on the main processor and the `task` constructs to be restricted to the subset of threads executing on the accelerators. Note that nested parallelism could not be used to achieve this allocation.

OpenMP is currently incapable of expressing these possible allocations. In this chapter, we propose a combination of *thread mapping* and *named subteams* to control how work is allocated amongst the different parts of a heterogeneous architecture. We also perform some experiments with a prototype implementation on the Cell Broadband Engine to show the benefit of giving the programmer control over the allocation of work onto a heterogeneous architecture.

7.2 Related work

The need to extend OpenMP to handle the increasing complexity of modern processors has prompted a number of proposals for extensions. We discuss them here in relation to the examples discussed in the previous section.

Device-annotated tasks [4, 26] have been proposed to allow OpenMP tasks to be

```
...
    while (p) {
        #pragma omp task target device(accelerator)
            process(p);
        p = p->next;
    }
...
```

Figure 7.3: Using device-annotated tasks to offload tasks to an accelerator

offloaded to accelerators. The OpenMP syntax is extended to allow a `device` clause to be attached to `task` constructs. This clause takes, as its argument, an identifier that represents a device capable of executing the task. Fig. 7.3 shows how this extension can be used to assign the tasks in our second example to an accelerator. This extension is an effective way of offloading tasks to accelerators, however it does not allow any of a team's threads to run on the accelerators. It also breaks the notion that all work in OpenMP is done by teams of threads, and forces the use of task-based parallelism where fork/join or loop-based parallelism might be more appropriate. These extensions also provide no mechanism to allow the programmer to specify how many instances of a given device should be used or if/when they should be initialised.

Zhang [76] proposes extensions to support thread mapping. The OpenMP execution model is extended to include the notion of a *logical processor*, which represents something on which a thread can run. An architecture is thought of as a hierarchy of these logical processors. The OpenMP syntax is extended by allowing an `on` clause to be used with `parallel` constructs. This clause takes an array of logical processors as its argument, then the team's threads are allocated from each processor in the list in turn. Fig. 7.4 shows how this kind of extension can be used to execute our first example on a selection of the processors in an architecture (in this case the processors `0`, `2` and `4`). However these extensions do not allow different pieces of work in a parallel region to be allocated to different selections of processors. A new team would have to be created each time a different set of processors is required, which is potentially expensive. They also do not allow tasks to be created on one processor for execution on another. Furthermore, these extensions break the OpenMP rule that the thread that encounters a `parallel` construct becomes part of the new team.

Multiple levels of parallelism are already supported in OpenMP with nested `parallel` constructs, however the creation of new thread teams is often prohibitively expensive, and tasks cannot be exchanged between the threads in separate teams. Accordingly Huang et al. [34] propose allowing workshares to be executed by a subteam, as a cheaper alternative

```
void a9(int n, int m, float *a, float *b, float *y, float *z)
{
    omp_group_t g[3];
    omp_group_t procs = omp_get_procs();

    assert(omp_get_num_members(procs) > 5);

    g[0] = omp_get_member(procs, 0);
    g[1] = omp_get_member(procs, 2);
    g[2] = omp_get_member(procs, 4);

    int i;
    #pragma omp parallel on(g)
    {
        ⋮
    }
}
```

Figure 7.4: Using thread mapping to control the allocation of the example from Fig. 7.1

to nested parallelism. It works using an `onthreads` clause for workshares, e.g.

`#pragma for onthreads(`$first$`:`$last$`:`$stride$`)`

where $first$ to $last$ is the range of thread indices and $stride$ is the stride used to select which threads are members of the subteam that will execute the workshare.

Other directive-based programming models, similar to OpenMP, have been created for use with accelerators, especially GPUs. The PGI Accelerator Model [74] consists of directives for executing loops on an accelerator. HMPP [21] uses directives to allow remote procedure calls on an accelerator. These calls can be asynchronous, giving them some of the functionality of OpenMP tasks. Both these models only support a single model of parallelism and can only allocate work to either the main processor or the accelerators.

Low-level programming models to enable programming with accelerators, especially GPUs, have also been created and have seen wide-spread use. The two most popular are the CUDA framework [58] by Nvidia and the OpenCL framework [40] by Apple. These frameworks allow programmers to utilise accelerators, but they are very low-level. This makes them difficult to use and code written in them is not very portable.

7.3 Design of the extensions

This section describes our extensions to the OpenMP execution model and syntax for implementing OpenMP on heterogeneous architectures. These are centred around two complementary extensions: thread mapping and named subteams.

7.3.1 Thread mapping and processors

The current OpenMP execution model consists of teams of threads executing work. We propose extending this model with *thread mapping*. Thread mapping consists of placing restrictions, for each thread, on which parts of an architecture can participate in that thread's execution.

We define an *architecture* as a collection of *processing elements* (a hardware thread, a CPU, an accelerator, etc.), which are capable of executing an OpenMP thread. Each thread is mapped to a subset of these processing elements, called its *processing set*. A thread may migrate between processing elements within its processing set, but will never be executed by an element outside its set. Which subsets of the processing elements in an architecture are allowed as processing sets is implementation-defined.

Processing sets are represented by values of the new type `omp_procs_t`. These values are created using implementation-defined expressions (typically macros or functions). Some examples of possible processing sets and expressions to represent them are:

- Group processors based on their functions (e.g. `MAIN` for the main processors and `ACC` for the accelerators).

- Arrange the processors in a tree and allow any sub-trees of this hierarchy as processing sets. These processing sets could be represented by a variadic function or macro, where a sub-tree is represented by their child indices on the path from the root of the tree (e.g. `TREE(`n`,`m`)` represents the sub-tree that is the mth child of the nth child of the root of the hierarchy).

- Other patterns that specify groups of processors (e.g. `STRIDE(`$n1$`,`$n2$`,`s`)`).

- Allow any set of processing elements as a processing set, and provide a full range of functions to manipulate these sets. (`union_procs(`p`,`q`)`, `intersect_procs(`p`,`q`)`, etc.).

- Expressions that do not change the processing set of a given `omp_procs_t` value, but provide guidance about how groups of threads should be executed on these processing elements (`SCATTER(`p`)`, `COMPACT(`p`)`, etc.).

7.3.2 Subteams

In the current OpenMP execution model tasks and workshares are executed by all of the threads in the team. We propose changing this model to allow tasks and workshares to be restricted to a subset of the threads in a team. To make creating these subsets easier we introduce the notion of *subteams*. Each team is divided into disjoint subteams, which are created when the team is created and remain fixed throughout the team's lifetime. The subset of threads associated with a task or workshare is specified by combining one or more subteams.

Subteams are referenced through the use of *subteam names*. These are identifiers with external linkage in their own namespace (similar to the names used by the `critical` construct). They exist for the duration of the program and can be used by different teams to represent different subteams. Each team maintains its own mapping between subteam names and subteams. Every subteam in a team must be mapped to a different subteam name.

7.3.3 Syntax

The `subteams` clause Thread mapping, subteam creation and the mapping of subteams to subteam names is all done using a single clause for the `parallel` construct:

`subteams(`$name_1$`(`$procs_1$`)[`$size_1$`], `$name_2$`(`$procs_2$`)[`$size_2$`], ...)`

Each argument of the clause creates a new subteam containing $size_i$ threads, which is mapped to the subteam name $name_i$. All the threads in the subteam are mapped to the processing set represented by the `omp_procs_t` expression $procs_i$. If no name is given then the subteam is mapped to a unique unspecified name. If no processing set is given, or the keyword `auto` is used instead, the implementation chooses an appropriate processing set. The first subteam listed is the *master subteam* and contains the master thread of the team. The processing set used with the master subteam must be a superset of the processing set that the encountering thread was mapped to.

The `on` clause Subteams can be used to specify the subset of threads associated with a workshare or `task` construct by annotating that construct with an `on` clause:

`on(`$subteams_1$`, `$subteams_2$`, ...)`

Each argument $subteams_i$ is a subteam name. The threads that are members of the subteams mapped to these subteam names (according to the current team's mapping) are used to execute the task or workshare.

```
void a9(int n, int m, float *a, float *b, float *y, float *z)
{
    int i;
    #pragma omp parallel subteams(st1(PROC_1)[4], st2(PROC_2)[4])
    {
        #pragma omp for nowait on(st1)
        for (i=1; i<n; i++)
            b[i] = (a[i] + a[i-1]) / 2.0;
        #pragma omp for nowait on(st2)
        for (i=0; i<m; i++)
            y[i] = sqrt(z[i]);
    }
}
```

Figure 7.5: Using subteams to divide work between processors in the example from Fig. 7.1

When a workshare or task construct without an `on` clause is encountered, it is associated with the subset of threads defined by the *default subteams set*. This is a set of subteam names, which is stored in a *per-task internal control variable* (these are the variables that control the behaviour of an OpenMP implementation). By default, the default subteams set is the set of subteam names that represent the subteams executing the current piece of work.

In addition to the `on` clause, an `on` construct is also added, of the form:

`#pragma omp on(`$subteams_1$`, `$subteams_2$`, ...)`
structured block

Each argument $subteams_i$ is a subteam name. Threads that are members of the subteams mapped to these subteam names execute the structured block, all other threads ignore the construct. The `on` clause can also be attached to a `parallel` construct as syntactic sugar for a `parallel` construct containing a single `on` construct.

7.3.4 Examples

Fig. 7.5 shows how these extensions can be used to allocate the loops of our first example onto different processors. Two subteams are created and each executes one of the loops. Here `PROC_1` and `PROC_2` are macros representing processing sets (each representing a different processor), and `st1` and `st2` are subteam names. Fig. 7.6 shows how these extensions can be used to assign the tasks of our second example onto accelerators. The `subteams` clause is used to create two subteams. The first subteam contains only the master thread and allows the implementation to choose a suitable processing set. The

```
void process_list_items(node *head)
{
    #pragma omp parallel subteams(main[1],accs(ACC)[5])
    {
        #pragma omp single on(main)
        {
            node *p = head;
            while (p) {
                #pragma omp task on(accs)
                    process(p);
                p = p->next;
            }
        }
    }
}
```

Figure 7.6: Using subteams to offload tasks to an accelerator in the example from Fig. 7.2

second subteam contains five threads mapped to the processing set represented by the macro `ACC`. The `single` construct is associated with the first subteam, which forces the block to be executed by the master thread. The `task` construct is then associated with the second subteam so that all the tasks created by it will be executed by the threads on the accelerators. Here `main` and `accs` are subteam names.

7.4 Implementation

We were able to add support for subteams to our OpenMP implementation without changing the OpenMP IL by using properties. The appropriate handlers in the front-end were extended to detect `subteams` and `on` clauses and add properties to the resulting IL nodes. Similarly, the handlers in the back-end were extended to detect the relevant properties and create the required calls into the run-time library.

Most of the implementation effort for these extensions was in adding support for heterogeneous architectures to our run-time library.

It is worth noting that these extensions were designed before EMCC was developed, but they turned out to be very easy to hook into it using its various extensions mechanisms. This perhaps illustrates the flexibility of EMCC's approach.

7.5 Experiments

To show the benefits of increasing the expressivity of OpenMP to allow the programmer to control how work is allocated to processing elements, a prototype implementation was created for the Cell Broadband Engine [14]. The aim of these experiments is to demonstrate that:

1. Traditional OpenMP programs can easily take advantage of heterogeneous architectures to improve performance

2. Different programs must be allocated onto heterogeneous architectures in different ways, so it is important to let the programmer control this allocation.

The Cell Broadband Engine processor includes one PowerPC Processor Element (PPE) and seven Synergistic Processor Elements (SPEs). The PPE has two hardware threads and accesses main memory through a cache. The SPEs cannot access main memory directly, instead they use 256kB local stores. The SPEs can perform DMA transfers between their local stores and main memory. The prototype implementation supports two processing sets: one mapping threads to the PPE and another mapping threads to the SPEs. The initial thread is mapped to the PPE. We use the Cell Broadband Engine of a Playstation 3 for the experiments.

The prototype implementation uses only simple mechanisms to move memory to/from the SPEs' local memories. Shared variables are accessed through simple software-managed caches in the local stores, while private variables are kept on the local stores within the call stack. Alternatives to simple software-managed caches have been shown to be more effective [15, 16, 29] and would be preferred in a more refined implementation.

The test programs are taken from the OpenMP C implementation of the NAS Parallel Benchmarks [37]. Each program was modified by adding a `subteams` clause to each of its `parallel` constructs. These clauses contain one subteam mapped to the PPE and one mapped to the SPEs. An `on` clause is also added to the `parallel` constructs to allow the parallel region to be executed by just the threads on the SPEs.

The test programs are:

EP Pairs of Gaussian random deviates are generated. The main part of the algorithm is a `for` workshare that performs computation on private data. There is very little communication between threads.

IS A large integer sort is performed. The main part of the algorithm is a `for` workshare that includes regular access to a shared array.

CG A conjugate-gradient method is used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. It contains a series of `for` workshares, some including irregular access to shared arrays.

EP

PPE Threads \ SPE Threads	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0	-	1.23	2.45	3.68	4.90	6.12	7.35	7.31
1	1	2.02	3.01	4.04	4.98	6.03	7.01	**7.98**
2	1.68	2.51	3.34	4.17	5.01	5.85	6.68	7.43
3	1.62	2.17	2.71	3.24	3.76	4.32	4.80	5.31

IS

PPE Threads \ SPE Threads	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0	-	0.07	0.14	0.20	0.27	0.33	0.39	0.36
1	1	0.14	0.20	0.27	0.33	0.39	0.44	0.42
2	**1.42**	0.20	0.27	0.33	0.39	0.45	0.50	0.45
3	1.18	0.27	0.33	0.39	0.45	0.50	0.55	0.49

CG

PPE Threads \ SPE Threads	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
0	-	0.10	0.20	0.30	0.40	0.50	0.6	0.22
1	1	0.21	0.31	0.41	0.51	0.62	0.72	0.19
2	**1.64**	0.31	0.41	0.51	0.62	0.72	0.82	0.21
3	0.5	0.29	0.33	0.37	0.40	0.41	0.43	0.15

Figure 7.7: Speed-ups obtained by using different numbers of PPE and SPE threads

We use these programs because they are traditional OpenMP programs, and they are of quite different natures, which allows us to effectively demonstrate the need for programmer control over the allocation of work on heterogeneous processors.

The speed-ups from adding threads to either the PPE or the SPEs are shown in Fig. 7.7. The greatest speed-up is highlighted in the table for each program.

The best speed-up for EP is obtained using a thread on the PPE and seven SPE threads. This is equivalent to one thread on each processor. This program is inherently very parallel, so the nearly linear relation between speed-up and number of threads is as expected.

The best speed-up for IS is obtained using two threads on the PPE and no SPE threads. This is probably due to the access to shared arrays in the main loop. The simple software cache used by our implementation is an inefficient method for handling

these regular loop accesses, and it is possible that a more refined implementation would actually get better performance from an allocation including SPEs.

The best speed-up for CG is also obtained using two threads on the PPE and none on the SPEs. In this case the access to shared arrays is irregular and very hard to optimise, so this result is unsurprising.

While memory sharing effects arguably dominate these figures, we argue that the size of the disparities shows the importance of allowing programmers to control how work is allocated onto a heterogeneous architecture. The performance of all three test programs is improved by using multiple threads, however the best thread mapping is not the same for all three. Observe that the best thread mapping for EP causes significant loss of performance in IS and CG.

Only the EP benchmark was able to improve performance by using the SPEs, but the other two benchmarks could both have their performance on the SPEs improved by using a more refined mechanism than a simple software-managed cache. However, some programs are simply not amenable to being split across a partitioned memory space, so primitives to express thread mapping are still required.

The EP benchmark also shows that performance can be improved by allowing a single workshare to operate across the different kinds of element on a heterogeneous architecture like the Cell.

7.6 Conclusion

In this chapter we have described the design of extensions to OpenMP to support heterogeneous architectures. These extensions allow the programmer to specify the allocation of work onto the different processing elements of the architecture.

We also described how these language extensions were implemented as extensions to our EMCC OpenMP implementation described in Chapter 5.

We showed benchmark results using this implementation that illustrate the need to allow the programmer to specify how work is allocated onto a heterogeneous architecture (in this case the Cell Broadband Engine).

Chapter 8

Conclusion

8.1 Conclusion

In this thesis we have addressed the problem of creating programming language extensions that take advantage of new architectural features. We have done this through the design and implementation of EMCC, a C compiler that allows extensions of its front-, middle- and back-ends. This allows language extensions to be semantically integrated with the rest of the language.

By allowing extensions to its middle-end, EMCC can support parallel-aware middle-ends. This enables the analysis and optimisation of parallel programs, and the implementation of high-level forms of parallelism.

We have demonstrated the benefits of such a compiler by creating a new implementation of the OpenMP programming language. This implementation uses more complex transformations to implement OpenMP tasks with more lightweight methods than previous implementations, allowing it to match Cilk in terms of performance. This lightweight implementation is based on our theoretical demonstration that OpenMP tasks can be implemented in a space-efficient way without affecting time efficiency.

We demonstrated the benefits of supporting high-level analysis for language extensions, by developing and implementing a new analysis of OpenMP programs, which detects when it is safe for multiple tasks to share a single stack. We developed a generalisation of logic programming to allow a concise but easily readable encoding of this analysis. Using this formalism we were able to show that the analysis has a unique solution and polynomial time complexity.

Finally, we used EMCC to implement extensions to OpenMP to support heterogeneous architectures. These extensions allow the programmer to choose how work is allocated to different processing elements on an architecture.

8.2 Future work

EMCC only supports extending *syntax* through attributes, pragmas and built-in functions. While most extensions do not require the ability to arbitrarily extend the syntax of a language, a more powerful system for syntax extensions might be considered desirable.

While EMCC allows optimisations and translations to be specified through common modular interfaces, we have not addressed the question of what these modular interfaces should be. This is a matter of identifying features that many middle-ends (and ILs in particular) have in common, so that the optimisations and translations can be applied as generally as possible. This could involve identifying common features across very different programming models: do middle-ends for functional languages share common structures with those for object-oriented or procedural languages?

The problem of identifying common modular interfaces is related to the issue of designing general frameworks for specifying optimisations. Frameworks, such as the monotone dataflow analysis frameworks [39], are popular methods of describing optimisations. Making such frameworks available in EMCC, through a modular interface, would allow many common optimisations to be easily encoded.

The current version of EMCC uses C as its back-end, treating it as a portable assembly language. EMCC would benefit from the addition of more back-ends: outputting C code is simple, but relies on using another C compiler. This means that new architectural features must first be made available (at a low-level) through another C compiler before EMCC can be used to create high-level language extensions that exploit that feature. The easiest way to add new back-ends would probably be to use an existing *virtual machine* such as LLVM [45] or JVM [51].

Appendix A

Inefficient schedule proof

A.1 Size lower-bound

For a p-thread schedule X, we define the overlap between the core of a chain x and the core of its successor as:

$$\text{overlap}_X(x) = \begin{cases} \{\} & \text{if } \omega^{-1}(t_{x,1}) \leq \omega^{-1}(c_{(x+1),1}) \\ \left[\omega^{-1}(c_{(x+1),1}),\ \omega^{-1}(t_{x,1})\right] & \text{otherwise} \end{cases}$$

We also define the set of variables that is accessed by the tail of a chain x:

$$V_x = \{v_{(x+1,1,1)}, \ldots, v_{(x+1,l,s)}\}$$

Note that $\forall v \in V_x\ \text{overlap}_X(x) \subseteq \text{live}_X(v)$.

We can now define the set of chains whose tails are being overlapped at a time-step i:

$$\text{exposed}_X(i) = \{x < w \mid i \in \text{overlap}_X(x)\}$$

Since all $v \in V_x$ are live while the chain x is being overlapped, and distinct variables that are live at the same time-step cannot be mapped to the same location:

$$\forall i, (x_1, x_2 < w), v_1 \in V_{x_1}, v_2 \in V_{x_2}\ .$$
$$x_1 \in \text{exposed}_X(i) \wedge x_2 \in \text{exposed}_X(i)$$
$$\implies (v_1 = v_2)\ \vee\ \ell(v_1) \neq \ell(v_2)$$

Therefore

$$\forall i\ .\ S_p(X) \geq ls\ |\text{exposed}_X(i)| \tag{A.1}$$

A.2 Time lower-bound

Observe that, by the definition of $T_p(X)$, for any valid schedule X:

$$T_p(X) \geq \omega^{-1}(t_{w,1}) - \omega^{-1}(c_{1,1})$$

We can expand this into

$$T_p(X) \geq \sum_{x=1}^{w}(\omega^{-1}(t_{x,1}) - \omega^{-1}(c_{x,1})) - \sum_{x=1}^{w-1}(\omega^{-1}(t_{x,1}) - \omega^{-1}(c_{x+1,1}))$$

Since the difference in the second sum of this inequality is always less than or equal to the number of time-steps that the chain is overlapped, we can reduce the inequality to:

$$T_p(X) \geq 2lw - \sum_{x=1}^{w-1}|\text{overlap}_X(x)|$$

Furthermore, since the sum of the number of time-steps each chain is overlapped is equal to the sum of the number of chains overlapped at each time-step:

$$T_p(X) \geq 2lw - \sum_{i=1}^{T_p(X)}|\text{exposed}_X(i)| \tag{A.2}$$

A.3 Combining the bounds

From (A.2) we can place a lower bound on the average number of chains overlapping at each time step:

$$\frac{1}{T_p(X)}\sum_{i=1}^{T_p(X)}|\text{exposed}_X(i)| \geq \frac{2lw}{T_p(X)} - 1$$

Since there must be at least one time-step at which the number of chains overlapped is greater than or equal to the average, we can combine this equation with (A.1) to get a lower bound on the space used:

$$S_p(X) \geq ls\left(\frac{2lw}{T_p(X)} - 1\right) \tag{A.3}$$

A.4 Efficient schedules

Now if we assume that the following condition holds (which is a requirement of the schedules from an absolutely time-efficient schedule when $p \leq w$)

$$T_p(X) = O(\frac{T_1}{p}) = O(\frac{4lw}{p})$$

Substituting this into (A.3) and reducing gives us:

$$\begin{aligned} S_p(X) &\geq ls \left(\frac{2lw}{O(\frac{4lw}{p})} - 1 \right) \\ &\geq ls \left(\Omega(\frac{p}{2}) - 1 \right) \\ &\geq ls \times \Omega(p-1) \end{aligned}$$

Finally, since $p > 1$, we can simplify this to:

$$S_p(X) = \Omega(lsp)$$

Appendix B

Task-based space efficiency proof

B.1 Definitions

B.1.1 Task-local variables

All the variables in a task-based computation are local to a specific task. Since tasks are totally ordered sequences of instructions $n_0, \ldots, n_l$, we can define the local live variables at an instruction n_i as:

$$\text{liveLocalVars}(n_i) = \{v \in V \mid \exists\, j < i.\ n_j \in \mathcal{A}_v \wedge \exists\, k > i.\ n_k \in \mathcal{A}_v\}$$

B.1.2 Spawn descendents

We write spawned(n) for the set of tasks (either one or none) that are spawned by that instruction (i.e. they are the destination of a spawn edge that comes from n).

We define all the tasks descended from a spawn instruction as the set of tasks that are descended from the task spawned by that instruction:

$$\text{spawnDescendents}(n) = \{\tau \mid \exists\, \tau' \in \text{spawned}(n).\ \tau' \prec \tau\}$$

B.1.3 Sync descendents

We write synced(n) for the set of tasks that are synced by that instruction (i.e. they are the orign of a join edge leads to n).

We define all the tasks descended from a sync instruction as the set of tasks that are descended from a task synced by that instruction:

$$\text{syncDescendents}(n) = \{\tau \mid \exists\, \tau' \in \text{synced}(n).\ \tau' \prec \tau\}$$

B.2 Pre-order scheduler efficiency

Note that, in a single-threaded schedule $\mathcal{W}_0$ from a depth-first pre-order scheduler, when a task reaches a sync instruction n all the tasks in synced(n) will be waiting to start. These tasks must be finished before the sync instruction is executed, and since this is a depth-first schedule all their descendents must also be finished before the sync instruction is executed. This means that if n_i is a sync instruction from a task with instructions $n_0, \ldots, n_k$ then:

$$\begin{aligned} &\forall \tau \in \text{syncDescendents}(n_i). \\ &\qquad \text{lifetime}_{\mathcal{W}_0}(\tau) \subseteq [\omega^{-1}(n_{i-1}), \omega^{-1}(n_i)] \end{aligned}$$

therefore

$$\begin{aligned} &\forall \tau \in \text{syncDescendents}(n_i). \quad \forall v \in \text{liveLocalVars}(n_i) \\ &\qquad \text{lifetime}_{\mathcal{W}_0}(\tau) \subseteq \text{live}_{\mathcal{W}_0}(v) \end{aligned}$$

From this we define the *pre-order depth* of an instruction. This is essentially the activation depth when the instruction is executed by a single-threaded schedule from a pre-order scheduler. For an instruction n that is part of the task τ:

$$\begin{aligned} \text{preorderDepth}(n) = &\left\{ v \in V \mid \exists\, m. \begin{array}{c} \tau \in \text{syncDescendents}(m) \\ \wedge \quad v \in \text{liveLocalVars}(m) \end{array} \right\} \\ &\cup \quad \text{liveLocalVars}(n) \end{aligned}$$

Note that $\forall\, n.\ S_{\mathcal{W}} > |\text{preorderDepth}(n)|$.

Now consider a p-thread schedule $\mathcal{W}_p$ a depth-first pre-order scheduler. By the definition of pre-order scheduling, at any point in the execution schedule, every live task is either being executed or suspended at a sync whilst one of that sync's descendents is being executed.

Since task-local variables can only be live while their associated tasks are live, every variable that is live at a particular time-step is associated with either a currently executing instruction or sync instruction with an executing descendent:

$$\begin{aligned} \forall\, i > 0. \quad &\forall v \in \{v \in V \mid i \in \text{live}_{\mathcal{W}_p}(v)\}. \quad \exists\, n \in \{n \mid \omega^{-1}(n) = i\}. \\ &\big(\exists m.\ \tau \in \text{syncDescendents}(m) \ \wedge \ v \in \text{liveLocalVars}(m)\big) \\ &\quad \vee \quad v \in \text{liveLocalVars}(n) \\ &\text{where } n \in \tau \end{aligned}$$

This means that each live variable is a member of the pre-order depth of an executing instruction:

$$\forall i > 0. \qquad \{v \in V \mid i \in \text{live}_{\mathcal{W}_p}(v)\} \subseteq \bigcup_{k \leq p} \text{preorderDepth}(\omega_k(i))$$

Therefore the scheduler can create a schedule $\mathcal{W}_p$ such that $S_p(\mathcal{W}_p) \leq p \times S_{\mathcal{W}}$, so depth-first pre-order schedulers can be space efficient with single-threaded schedules that use $S_{\mathcal{W}}$ space.

B.3 Post-order scheduler efficiency

Note that, in a single-threaded schedule $\mathcal{D}_0$ from a depth-first post-order scheduler, when a task reaches a spawn instruction n it must execute the task in spawned(n), and since this is a depth-first schedule all its descendents must also be finished before the original task is resumed. This means that if n_i is a spawn instruction from a task with instructions $n_0, \ldots, n_k$ then:

$$\begin{aligned} &\forall \tau \in \text{spawnDescendents}(n_i). \\ &\qquad \text{lifetime}_{\mathcal{D}_0}(\tau) \subseteq [\omega^{-1}(n_i), \omega^{-1}(n_{i+1})] \end{aligned}$$

therefore

$$\begin{aligned} \forall \tau \in \text{spawnDescendents}(n_i). \quad \forall v \in \text{liveLocalVars}(n_i) \\ \text{lifetime}_{\mathcal{D}_0}(\tau) \subseteq \text{live}_{\mathcal{D}_0}(v) \end{aligned}$$

From this we define the *post-order depth* of an instruction. This is essentially the activation depth when the instruction is executed by a single-threaded schedule from a post-order scheduler. For an instruction n that is part of the task τ:

$$\begin{aligned} \text{postorderDepth}(n) = \quad & \left\{ v \in V \mid \exists\, m.\ \begin{array}{c} \tau \in \text{spawnDescendents}(m) \\ \wedge \quad v \in \text{liveLocalVars}(m) \end{array} \right\} \\ \cup \quad & \text{liveLocalVars}(n) \end{aligned}$$

Note that $\forall\, n.\ S_{\mathcal{D}} > |\text{postorderDepth}(n)|$.

Now consider a p-thread schedule $\mathcal{D}_p$ a depth-first post-order scheduler. By the definition of post-order scheduling, at any point in the execution schedule, every live task is either: being executed; suspended at a spawn whilst one of that spawn's descendents is being executed; or suspended at a sync whilst one of that sync's descendents is being executed.

Since task-local variables can only be live while their associated tasks are live, every variable that is live at a particular time-step is associated with either a currently executing instruction, a spawn instruction with an executing descendent, or a sync instruction with

an executing descendent:

$$\begin{aligned}\forall\, i > 0.\quad & \forall v \in \{v \in V \mid i \in \text{live}_{\mathcal{D}_p}(v)\}.\quad \exists\, n \in \{n \mid \omega^{-1}(n) = i\}.\\ & \big(\exists m.\ \tau \in \text{spawnDescendents}(m)\ \wedge\ v \in \text{liveLocalVars}(m)\big)\\ & \quad \vee\ \big(\exists m.\ \tau \in \text{syncDescendents}(m)\ \wedge\ v \in \text{liveLocalVars}(m)\big)\\ & \quad\ \vee\quad v \in \text{liveLocalVars}(n)\\ \text{where } n \in \tau\end{aligned}$$

This means that each live variable is a member of either the post-order depth or the pre-order depth of an executing instruction:

$$\begin{aligned}&\forall i > 0.\\ &\quad \{v \in V \mid i \in \text{live}_{\mathcal{D}_p}(v)\} \subseteq \bigcup_{k \le p} \text{postorderDepth}(\omega_k(i)) \cup \text{preorderDepth}(\omega_k(i))\end{aligned}$$

Therefore the scheduler can create a schedule $\mathcal{D}_p$ such that $S_p(\mathcal{D}_p) \le p \times (S_\mathcal{D} + S_\mathcal{W})$, so depth-first post-order schedulers can be space efficient with single-threaded schedules that use $(S_\mathcal{D} + S_\mathcal{W})$ space.

Appendix C

Stack-based inefficiency proof

C.1 Restrictions on sharing stacks

We say that a task τ is *inlined* iff none of the instructions of parent(τ) are scheduled during lifetime$_X(\tau)$. We define the set of inlined tasks in a schedule X as:

$$\text{inlined}_X = \{\ \tau \in \mathcal{T} \mid\ \forall n \in \text{parent}(\tau).\ \omega^{-1}(n) \notin \text{lifetime}_X(\tau)\}$$

A stack-based schedule allocates the variables used by a task τ from a single stack stack(τ). We say that a task τ has a *fresh* stack iff for any other task τ' that uses the same stack either $\tau \prec \tau'$ or lifetime$_X(\tau)$ and lifetime$_X(\tau')$ do not overlap. We define the set of tasks with fresh stacks in a schedule X as:

$$\begin{aligned}\text{fresh}_X = \{\ \tau \in \mathcal{T} \mid\ \forall \tau \not\prec \tau'.\quad & \text{stack}(\tau) = \text{stack}(\tau') \\ & \Rightarrow\ \text{lifetime}_X(\tau) \cap \text{lifetime}_X(\tau') = \{\}\ \}\end{aligned}$$

In a stack-based schedule every task is either inlined or assigned a fresh stack: $\mathcal{T} = \text{inlined}_X \cup \text{fresh}_X$.

C.2 An inefficient example

Consider the task-based computation in Fig. C.1.

Note that the lifetime of every task is included in the lifetimes of all the tasks that aren't descended from it:

$$\forall \tau \not\prec \tau'.\quad \text{lifetime}_X(\tau) \subset \text{lifetime}_X(\tau') \tag{C.1}$$

This means that the lifetime of a task τ_x (where $0 < x < d$) includes the lifetime of its child task τ_{x+1}. This gives us a lower bound on the lifetimes of tasks whose child task is

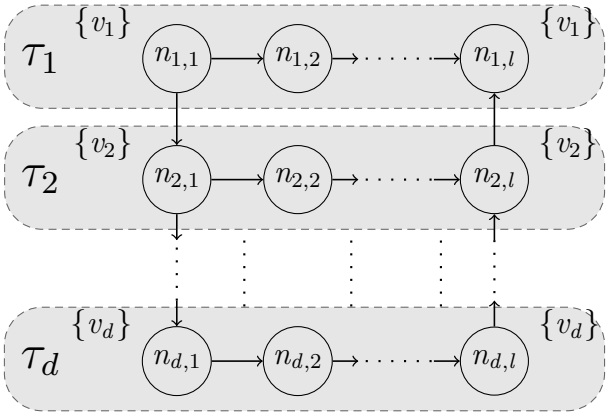


Figure C.1: Inefficient Stack-based Computation

inlined:

$$\begin{aligned} &\forall 0 < x < d. \quad \tau_{x+1} \in \text{inlined}_X \\ &\qquad \Rightarrow \text{lifetime}_X(\tau_x) \supseteq \text{lifetime}_X(\tau_{x+1}) \uplus \{\omega^{-1}(n) \mid n \in \tau_x\} \\ &\forall 0 < x < d. \quad \tau_{x+1} \in \text{inlined}_X \\ \therefore &\qquad \Rightarrow |\text{lifetime}_X(\tau_x)| \geq |\text{lifetime}_X(\tau_{x+1})| + l \end{aligned}$$

Applying this lower bounded inductively and allowing for those tasks whose children are not inlined (and must therefore be in fresh_X), we obtain:

$$\forall 0 < x \leq d. \quad |\text{lifetime}_X(\tau_x)| \geq (d - x - |\{\tau_y \in \text{fresh}_X \mid x < y \leq d\}| + 1) \times l$$

From C.1 and the definition of fresh_X it is clear that:

$$\forall \tau_x, \tau_y \in \text{fresh}_X. \quad \text{stack}(\tau_x) \neq \text{stack}(\tau_y)$$

Combining these two equations, we can see that for any p-threaded schedule X that is stack-based and uses fewer than s stacks, there is a lower bound on the execution time:

$$T_p(X) \geq (d - s)l$$

Bibliography

[1] Addison, C., LaGrone, J., Huang, L., and Chapman, B. OpenMP 3.0 tasking implementation in OpenUH. In *Open64 Workshop at CGO* (2009), vol. 2009.

[2] Alt, M., Assmann, U., and Van Someren, H. Cosy compiler phase embedding with the cosy compiler model. In *Compiler Construction* (1994), pp. 278–293.

[3] Apt, K. R., Blair, H. A., and Walker, A. *Towards a theory of declarative knowledge*. IBM TJ Watson Research Center, 1986.

[4] Ayguade, E., Badia, R. M., Cabrera, D., Duran, A., Gonzalez, M., Igual, F., Jimenez, D., Labarta, J., Martorell, X., Mayo, R., Perez, J. M., and Quintana-Ortí, E. S. A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures. *LNCS 5568* (2009), 154.

[5] Ayguadé, E., Duran, A., Hoeflinger, J., Massaioli, F., and Teruel, X. An Experimental Evaluation of the New OpenMP Tasking Model. 63.

[6] Balart, J., Duran, A., Gonzàlez, M., Martorell, X., Ayguadé, E., and Labarta, J. Nanos mercurium: a research compiler for openmp. In *Proceedings of the European Workshop on OpenMP* (2004), vol. 8.

[7] Blumofe, R. D., and Leiserson, C. E. Space-efficient scheduling of multithreaded computations. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing* (1993), pp. 362–371.

[8] Boehm, H. J. Threads cannot be implemented as a library. In *ACM SIGPLAN Notices* (2005), vol. 40, pp. 261–268.

[9] Brent, R. P. The parallel evaluation of general arithmetic expressions. *Journal of the ACM (JACM) 21*, 2 (1974), 201–206.

[10] Brown, K. J., Sujeeth, A. K., Lee, H. J., Rompf, T., Chafi, H., Odersky, M., and Olukotun, K. A heterogeneous parallel framework for domain-specific languages. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on* (2011), pp. 89–100.

[11] Burmako, E. Scala Macros: Let Our Powers Combine! In *Proceedings of the 4th Annual Scala Workshop* (2013).

[12] Campbell, B. Type-based amortized stack memory prediction.

[13] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., Von Praun, C., and Sarkar, V. X10: an object-oriented approach to non-uniform cluster computing. In *ACM SIGPLAN Notices* (2005), vol. 40, pp. 519–538.

[14] Chen, T., Raghavan, R., Dale, J. N., and Iwata, E. Cell Broadband Engine Architecture and Its First Implementation; A Performance View. *IBM Journal of Research and Development 51*, 5 (2007), 559.

[15] Chen, T., Sura, Z., O'Brien, K., and O'Brien, J. K. Optimizing the Use of Static Buffers for DMA on a CELL Chip. *LNCS 4382* (2007), 314.

[16] Chen, T., Zhang, T., Sura, Z., and Tallada, M. G. Prefetching Irregular References for Software Cache on Cell. In *Proceedings of the Sixth Annual IEEE/ACM International Symposium on Code Generation and Optimization* (2008), p. 155.

[17] Cox, R., Bergan, T., Clements, A. T., Kaashoek, F., and Kohler, E. Xoc, an extension-oriented compiler for systems programming. In *ACM SIGOPS Operating Systems Review* (2008), vol. 42, pp. 244–254.

[18] Damásio, C., and Pereira, L. Antitonic logic programs. *Logic Programming and Nonmotonic Reasoning* (2001), 379–393.

[19] de Dinechin, C. XLR: Extensible language and runtime.

[20] de Rauglaudre, D. The Camlp4 preprocessor. *See: http://caml. inria.fr/camlp4* (2001).

[21] Dolbeau, R., Bihan, S., and Bodin, F. HMPP: A Hybrid Multi-core Parallel Programming Environment. In *First Workshop on General Purpose Processing on Graphics Processing Units* (2007).

[22] Donnelly, C., and Stallman, R. M. *Bison: The YACC-compatible Parser Generator*. Free Software Foundation, 1995.

[23] Duran, A., Teruel, X., Ferrer, R., Martorell, X., and Ayguade, E. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *Proceedings of the 2009 International Conference on Parallel Processing* (Washington, DC, USA, 2009), ICPP '09, IEEE Computer Society, pp. 124–131.

[24] Eager, D. L., Zahorjan, J., and Lazowska, E. D. Speedup versus efficiency in parallel systems. *Computers, IEEE Transactions on 38*, 3 (1989), 408–423.

[25] Faxén, K. F. Wool - a work stealing library. *ACM SIGARCH Computer Architecture News 36*, 5 (2009), 93–100.

[26] Ferrer, R., Beltran, V., Gonzàlez, M., Martorell, X., and Ayguadé, E. Analysis of Task Offloading for Accelerators. *LNCS 5952* (2010), 322.

[27] Frigo, M., Leiserson, C. E., and Randall, K. H. The implementation of the Cilk-5 multithreaded language. *ACM Sigplan Notices 33*, 5 (1998), 212–223.

[28] Gelfond, M., and Lifschitz, V. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic programming* (1988), vol. 161.

[29] Gonzàlez, M., O'Brien, K., Vujic, N., Martorell, X., Ayguadé, E., Eichenberger, A. E., Chen, T., Sura, Z., Zhang, T., and O'Brien, K. Hybrid Access-Specific Software Cache Techniques for the Cell BE Architecture. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques - PACT '08* (2008), p. 292.

[30] Graham, R. L. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics 17*, 2 (1969), 416–429.

[31] Grimm, R. xtc (eXTensible C). *See: http://cs.nyu.edu/rgrimm/xtc/*.

[32] Hedin, G., and Magnusson, E. JastAdd—an aspect-oriented compiler construction system. *Science of Computer Programming 47*, 1 (2003), 37–58. ¡ce:title¿Special Issue on Language Descriptions, Tools and Applications (L DTA'01)¡/ce:title¿.

[33] Hendler, D., Lev, Y., Moir, M., and Shavit, N. A dynamic-sized nonblocking work stealing deque.

[34] Huang, L., Chapman, B., and Liao, C. An Implementation and Evaluation of Thread Subteam for OpenMP Extensions. *Workshop on Programming Models for Ubiquitous Parallelism (PMUP 06), Seattle, WA* (2006).

[35] International Organization for Standardization. *ISO Standard 9126: Software Engineering - Product Quality*. International Organization for Standardization, 2001.

[36] JetBrains. JetBrains Meta Programming System. *See: http://www.jetbrains.com/mps/*.

[37] Jin, H., Frumkin, M., and Yan, J. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. Tech. rep., 1999.

[38] Johnson, S. C. *Yacc: Yet another compiler-compiler*, vol. 32. Bell Laboratories Murray Hill, NJ, 1975.

[39] Kam, J. B., and Ullman, J. D. Monotone data flow analysis frameworks. *Acta Informatica 7*, 3 (1977), 305–317.

[40] Khronos OpenCL Working Group, et al. The opencl specification. *A. Munshi, Ed* (2008).

[41] Knuth, D. E. Semantics of context-free languages. *Mathematical systems theory 2*, 2 (1968), 127–145.

[42] Kohlbecker, E., Friedman, D. P., Felleisen, M., and Duba, B. Hygienic macro expansion. In *Proceedings of the 1986 ACM conference on LISP and functional programming* (1986), pp. 151–161.

[43] Kowalski, R. *Predicate logic as programming language*. Edinburgh University, 1973.

[44] Kruglinski, D. J. *Inside Visual C++*. Microsoft press, 1997.

[45] Lattner, C., and Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on* (2004), pp. 75–86.

[46] Lee, I. Using memory mapping to support cactus stacks in work-stealing runtime systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques* (2010), pp. 411–420.

[47] Leijen, D., Schulte, W., and Burckhardt, S. The design of a task parallel library. In *Acm Sigplan Notices* (2009), vol. 44, pp. 227–242.

[48] Leroy, X. The OCaml programming language. *See: http://caml.inria.fr/ocaml/index.en.html* (1998).

[49] Lesk, M. E., and Schmidt, E. Lex: A lexical analyzer generator, 1975.

[50] Liao, C., Hernandez, O., Chapman, B., Chen, W., and Zheng, W. OpenUH: An optimizing, portable OpenMP compiler. *Concurrency and Computation: Practice and Experience 19*, 18 (2007), 2317–2332.

[51] Lindholm, T., and Yellin, F. *Java virtual machine specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[52] McCarthy, J. Circumscription—a form of non-monotonic reasoning. *Artificial intelligence 13*, 1 (1980), 27–39.

[53] Mertes, T. Seed7. The Extensible Programming Language.

[54] Moore, R. C. Semantical considerations on nonmonotonic logic. *Artificial intelligence 25*, 1 (1985), 75–94.

[55] Mozilla Foundation. The Rust Reference Manual. *See: http://www.rust-lang.org/*.

[56] Necula, G., McPeak, S., Rahul, S., and Weimer, W. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction* (2002), pp. 209–265.

[57] Nichols, B., Buttlar, D., and Farrell, J. *Pthreads programming: A POSIX standard for better multiprocessing*. O'Reilly Media, Inc., 1996.

[58] Nvidia, C. U. D. A. Compute unified device architecture programming guide.

[59] Nystrom, N., Clarkson, M., and Myers, A. Polyglot: An extensible compiler framework for Java. In *Compiler Construction* (2003), pp. 138–152.

[60] Olivier, S. L., and Prins, J. F. Evaluating OpenMP 3.0 Run Time Systems on Unbalanced Task Graphs. In *Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism* (Berlin, Heidelberg, 2009), IWOMP '09, Springer-Verlag, pp. 63–78.

[61] OpenMP Architecture Review Board. OpenMP Application Program Interface. Tech. rep., 2008.

[62] Podobas, A., Brorsson, M., and Faxén, K. F. A comparison of some recent task-based parallel programming models.

[63] Pottier, F., and Régis-Gianas, Y. The Menhir Parser Generator. *See: http://gallium.inria.fr/ fpottier/menhir/*.

[64] Reinders, J. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Incorporated, 2007.

[65] Reiter, R. A logic for default reasoning. *Artificial intelligence 13*, 1 (1980), 81–132.

[66] Shavit, N., and Touitou, D. Software transactional memory. *Distributed Computing 10*, 2 (1997), 99–116.

[67] Sheard, T., and Jones, S. P. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell* (2002), pp. 1–16.

[68] Stallman, R. M. GNU compiler collection internals. *Free Software Foundation* (2002).

[69] Sukha, J. Brief announcement: A lower bound for depth-restricted work stealing. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures* (2009), pp. 124–126.

[70] Supertech Research. Cilk 5.4. 6 Reference Manual, 1998.

[71] Teruel, X., Martorell, X., Duran, A., Ferrer, R., and Ayguadé, E. Support for OpenMP tasks in Nanos v4. In *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research* (2007), pp. 256–259.

[72] Teruel, X., Unnikrishnan, P., Martorell, X., Ayguadé, E., Silvera, R., Zhang, G., and Tiotto, E. OpenMP tasks in IBM XL compilers. In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds* (2008), p. 16.

[73] Wilson, R. P., French, R. S., Wilson, C. S., Amarasinghe, S. P., Anderson, J. M., Tjiang, S. W., Liao, S.-W., Tseng, C.-W., Hall, M. W., Lam, M. S., et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM Sigplan Notices 29*, 12 (1994), 31–37.

[74] Wolfe, M. Implementing the PGI Accelerator Model. In *GPGPU'10: Proceedings of the 3rd Workshop on GPGPUs* (New York, USA, 2010), ACM, pp. 43–50.

[75] Zenger, M., and Odersky, M. Extensible algebraic datatypes with defaults. In *ACM SIGPLAN Notices* (2001), vol. 36, pp. 241–252.

[76] Zhang, G. Extending the OpenMP Standard for Thread Mapping and Grouping. *LNCS 4315* (2008), 435.

[77] Zingaro, D. Modern extensible languages. *McMaster University, Hamilton* (2007).