Concise analysis using implication algebras for task-local memory optimisation

Leo White and Alan Mycroft

Computer Laboratory, University of Cambridge
William Gates Building, 15 JJ Thomson Avenue,
Cambridge CB3 0FD, UK
Firstname.Lastname@cl.cam.ac.uk

Abstract. OpenMP is a pragma-based extension to C to support parallelism. The OpenMP standard recently added support for task-based parallelism but in a richer way than languages such as Cilk. Naïve implementations give each task its own stack for task-local memory, which is very inefficient.
We detail a program analysis for OpenMP to enable tasks to share stacks without synchronisation—either unconditionally or dependent on some cheap run-time condition which is very likely to hold in busy systems.
The analysis is based on a novel implication-algebra generalisation of logic programming which allows concise but easily readable encodings of the various constraints. The formalism enables us to show that the analysis has a unique solution and polynomial-time complexity.
We conclude with performance figures.

1 Introduction

Task-based parallelism is a high level parallel programming model made popular by languages such as Cilk [1]. It uses lightweight cooperative threads called *tasks*, which may *spawn* new tasks and *synchronise* with the completion of the tasks that they have spawned.

OpenMP is a shared-memory parallel programming language that has recently introduced support for task-based parallelism—in a less restricted form than Cilk. OpenMP task implementations have struggled to compete with other task-based systems [2, 3] as they have been too heavyweight, allocating a whole stack for each task and then restricting parallelism at some cut-off to limit memory consumption.

However, in many cases two or more OpenMP tasks could share stacks without any synchronisation. This paper describes the analysis required to implement such an optimisation. It revolves around analysing the stack usage of a program's tasks.

1.1 OpenMP

OpenMP was originally designed for scientific applications on shared-memory multi-processors. 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, the OpenMP Architecture Review Board released OpenMP 3.0 [4], which includes support for task-based parallelism.

The execution model of OpenMP within the parallel sections of a program consists of *teams* of *threads* executing *tasks* and *workshares*. These threads are heavyweight and preemptively scheduled—typically implemented using system threads. Workshares support data parallelism: they divide work amongst the threads in a team; e.g. the `for` workshare allows iterations of a `for` loop to be divided amongst the threads.

Tasks express more dynamic forms of parallelism. Tasks are sequences of instructions to be executed by a thread. They need not be executed immediately, but can be deferred until later or executed by a different thread in the team. When a team of threads 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.

One point to note, in contrast to languages like Cilk, is that OpenMP tasks can outlive the task which spawned them. This breaks a theorem (Blumofe et al. [5]) for Cilk-like languages about existence of time- and space-optimal execution schedules, and complicates our stack size analysis.

1.2 Optimising task-local memory allocation

We develop 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. 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. 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. 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
}
```

Fig. 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);
}
```

Fig. 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. 1 will definitely finish before the second child task begins.

We can merge spawn instructions like the second one in Fig. 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 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*).

1.3 Concise analysis

In order to express our analysis concisely, we develop a generalisation of logic programming. 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.

2 Logic programming: negation and multi-valued logic

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 [6]. This work uses a variant of logic programming where we restrict terms to be variables or constants (the Datalog restriction) but also allow negation and multi-valued logic.

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 $\mathit{ground}(P)$ for the ground instances of rules in P.

Note that we do not require P to be finite. Indeed the program analyses we propose naturally give infinite such P, but Section 8 shows these to have an equivalent finite form.

Interpretations, 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 $\{\mathit{false} \sqsubseteq \mathit{true}\}$). 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 $I \models r \quad (\forall r \in \mathit{ground}(P))$. 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:

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

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.

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

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

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

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

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*[1] of P as M_k below:

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

Stable models *Stable models* (Gelfond et al. [8]) 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:

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

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

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.

2.2 Implication algebra programming

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. [9])—to give *implication programs*.

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

[1] Apt et al. [7] show that this standard model does not depend on which stratification of P is used.

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 $I\colon \mathcal{HB}_P \to \mathfrak{L}$ which extends to a valuation function $\hat{I}\colon \mathcal{HL}_P \to \mathfrak{L}$.

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, canonical semantics, 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 2.1 apply to general implication programs exactly as they do to general logic programs.

3 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[2]:

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

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

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

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

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

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.

4 OpenMP program representation

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. 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 $\mathit{func}(n) = g$.

4.1 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:

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

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

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

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

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).

5 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).

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 5.3). Finally, we show how a context-sensitive variant of the analysis naturally follows (Section 5.5).

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 8 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:

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

#pragma omp taskwait

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

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

Fig. 3. Example of different stack sizes.

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[3]. 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.

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. 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 $P_{(M,U)}$.

[3] 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.

5.1 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:

$$\begin{aligned}
&[f \in \mathcal{F},\ n \in \mathit{Nodes}(f),\ (m_0, \ldots, m_k) \in \mathit{Upaths}(\llcorner\!\lrcorner, n)] \\
&\quad \texttt{TotalSize}\langle f \rangle \longleftarrow \mathit{frame}(f) + \texttt{PostSize}\langle m_0 \rangle \\
&\qquad\qquad + \sum_{0<i<k} \texttt{ThroughSize}\langle m_i \rangle \\
&\qquad\qquad + \texttt{PreSize}\langle m_k \rangle
\end{aligned}$$

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:

$$\begin{aligned}
&[f \in \mathcal{F},\ (n_0, \ldots, n_k) \in \mathit{Upaths}(\llcorner\!\lrcorner, \mathit{end}(f))] \\
&\quad \texttt{PostSize}\langle f \rangle \longleftarrow \texttt{PostSize}\langle n_0 \rangle \\
&\qquad\qquad + \sum_{0<i\le k} \texttt{ThroughSize}\langle n_i \rangle
\end{aligned}$$

$$\begin{aligned}
&[f \in \mathcal{F},\ (n_0, \ldots, n_k) \in \mathit{Upaths}(\mathit{start}(f), \llcorner\!\lrcorner)] \\
&\quad \texttt{PreSize}\langle f \rangle \longleftarrow \mathit{frame}(f) + \sum_{0\le i<k} \texttt{ThroughSize}\langle n_i \rangle \\
&\qquad\qquad + \texttt{PreSize}\langle n_k \rangle
\end{aligned}$$

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

5.2 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:

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

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:

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

5.3 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 $\top$:

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

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 $\top$:

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

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

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.

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 $\mathit{Lost} = M' \setminus M$ and $\mathit{Gained} = M \setminus M'$, we have that every node $n \in \mathit{Lost}$ dominates (in $\mathit{CallGraph}$ with respect to paths starting at $\mathcal{S}$) every node $m \in \mathit{Gained}$. 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').[4]

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

Even with this heuristic programs may still have several equally preferred safe solutions. We call such solutions *optimal.* In Section 5.5 we discuss context sensitivity; the context-sensitive version of our analysis has only a single optimal solution.

5.4 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 naïve 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.
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 that this approach can be encoded as a general implication program; this enables a more efficient solver.

5.5 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 6.1), 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 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.

6.1 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.

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:

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

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

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

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:

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

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:

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

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

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).

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

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

6.2 Other rules

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

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

The remaining stack size rules are based on those in Section 5 and are omitted for brevity.

6.3 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 5.3. The merged set M and unguarded merge set U are given by:

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

7 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 $\texttt{SuppPreSize}\langle n\rangle$ equals $\top$ then the value of $\texttt{SuppPostSize}\langle n\rangle$ will not affect the values of $\texttt{PreSize}\langle n\rangle$ or $\texttt{ThroughSize}\langle n\rangle$. 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 $\texttt{ThroughSize}\langle f_{(cr,\mathtt{T},sg,crg)}\rangle \neq \texttt{ThroughSize}\langle f_{(cr,\mathtt{F},sg,crg)}\rangle$ then $\texttt{PostSize}\langle f_{(cr,\mathtt{T},sg,crg)}\rangle = \texttt{PostSize}\langle f_{(cr,\mathtt{F},sg,crg)}\rangle = \top$. Therefore, for any instruction n, the value of $\texttt{SuppPreSize}\langle n\rangle$ will not affect the values of $\texttt{PostSize}\langle n\rangle$. 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.

8 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:

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

Since the number of rules is finite, 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.

9 Evaluation

We implemented the optimisation within our EMCC prototype compiler. We compared it to our compiler without the optimisation, as well as to three other OpenMP implementations: GCC [10], OpenUH [11] and Nanos [12].

Each of these other implementations uses a stack per-task. To prevent excessive memory consumption they restrict parallelism after a certain number of stacks have been created. By decreasing the required number of stacks, our optimisation allows us to restrict parallelism less often.

Our EMCC implementation and OpenUH are more lightweight than GCC and Nanos: OpenUH uses coroutines to implement its tasks, and our implementation divides tasks into continuations.

We compared the implementations using programs from the Barcelona Tasks Suite [13]: *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 results are shown in Fig. 4. Alignment shows no real difference between implementations. Sort shows performance gains for the lightweight implementations, and further gains due to the optimisation. NQueens shows significant gains due to the optimisation.

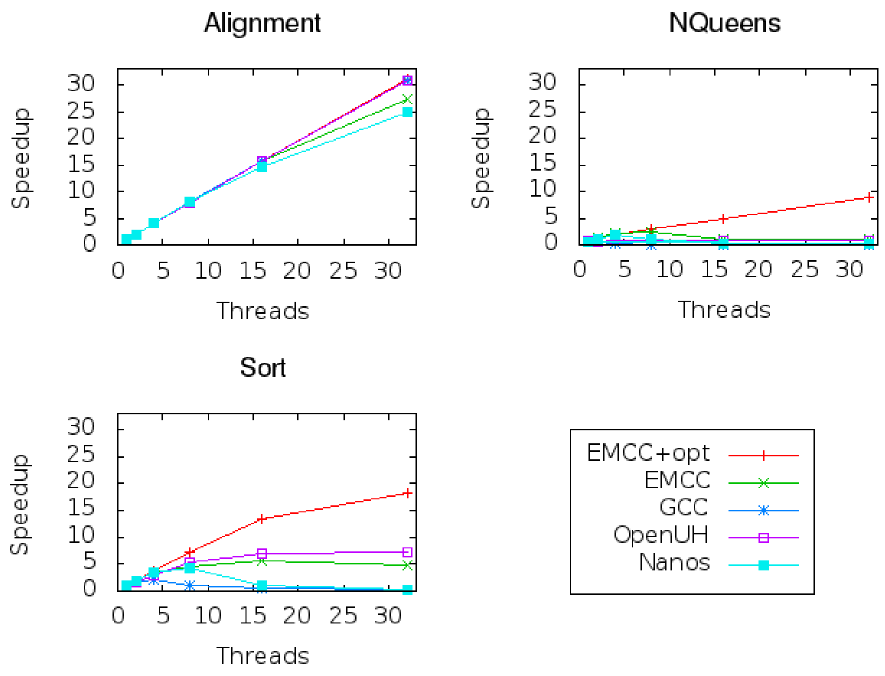

Fig. 4. Results.

10 Conclusion

In this paper 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.

This optimisation has enabled us to implement a very lightweight implementation of OpenMP, and we have shown that this outperforms existing OpenMP implementations that give each task their own stack for task-local memory.

References

1. Supertech Research: Cilk 5.4.6 Reference Manual (1998)
2. Podobas, A., Brorsson, M., Faxén, K.F.: A comparison of some recent task-based parallel programming models. (2010)
3. Olivier, S.L., 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. IWOMP '09, Berlin, Heidelberg, Springer-Verlag (2009) 63–78
4. OpenMP Architecture Review Board: OpenMP Application Program Interface. Technical report (2008)
5. Blumofe, R.D., Leiserson, C.E.: Space-efficient scheduling of multithreaded computations. In: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing. (1993) 362–371
6. Kowalski, R.: Predicate logic as programming language. Edinburgh University (1973)
7. Apt, K.R., Blair, H.A., Walker, A.: Towards a theory of declarative knowledge. IBM TJ Watson Research Center (1986)
8. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings of the 5th International Conference on Logic programming. Volume 161. (1988)
9. Damásio, C., Pereira, L.: Antitonic logic programs. Logic Programming and Nonmotonic Reasoning (2001) 379–393
10. Stallman, R.M.: GNU compiler collection internals. Free Software Foundation (2002)
11. Addison, C., LaGrone, J., Huang, L., Chapman, B.: OpenMP 3.0 tasking implementation in OpenUH. In: Open64 Workshop at CGO. Volume 2009. (2009)
12. Teruel, X., Martorell, X., Duran, A., Ferrer, R., 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) 256–259
13. Duran, A., Teruel, X., Ferrer, R., Martorell, X., 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. ICPP '09, Washington, DC, USA, IEEE Computer Society (2009) 124–131