I Appendices

I.1 mwp-Analysis Improvement and Implementation: Realizing Implicit Computational Complexity

mwp-Analysis Improvement and Implementation: Realizing Implicit Computational Complexity 4

Clément Aubert
School of Computer and Cyber Sciences, Augusta University, GA, USA

Thomas Rubiano
LIPN – UMR 7030 Université Sorbonne Paris Nord, France

Neea Rusch
School of Computer and Cyber Sciences, Augusta University, GA, USA

Thomas Seiller
LIPN – UMR 7030 Université Sorbonne Paris Nord, France
CNRS, Paris, France

Abstract

Implicit Computational Complexity (ICC) drives better understanding of complexity classes, but it also guides the development of resources-aware languages and static source code analyzers. Among the methods developed, the *mwp-flow analysis* [23] certifies polynomial bounds on the size of the values manipulated by an imperative program. This result is obtained by bounding the transitions between states instead of focusing on states in isolation, as most static analyzers do, and is not concerned with termination or tight bounds on values. Those differences, along with its built-in compositionality, make the mwp-flow analysis a good target for determining how ICC-inspired techniques diverge compared with more traditional static analysis methods. This paper's contributions are three-fold: we fine-tune the internal machinery of the original analysis to make it tractable in practice; we extend the analysis to function calls and leverage its machinery to compute the result of the analysis efficiently; and we implement the resulting analysis as a lightweight tool to automatically perform data-size analysis of `C` programs. This documented effort prepares and enables the development of certified complexity analysis, by transforming a costly analysis into a tractable program, that furthermore decorrelates the problem of deciding if a bound exist with the problem of computing it.

2012 ACM Subject Classification Software and its engineering → Automated static analysis; Theory of computation → Complexity theory and logic; Theory of computation → Logic and verification

Keywords and phrases Static Program Analysis, Implicit Computational Complexity, Automatic Complexity Analysis, Program Verification

Digital Object Identifier 10.4230/LIPIcs.FSCD.2022.26

Related Version *Technical report*: `https://hal.archives-ouvertes.fr/hal-03596285` [4]

Supplementary Material *Software (Source Code)*: `https://github.com/statycc/pymwp`
archived at `swh:1:dir:22a4ab0cfad49138981ed25fc2abfe830fb7ccdf`
Software (Documentation and Demo): `https://statycc.github.io/pymwp`

Funding This research is supported by the Th. Jefferson Fund of the Embassy of France in the United States and the FACE Foundation, and has benefited from the research meeting 21453 "Static Analyses of Program Flows: Types and Certificate for Complexity" in Schloss Dagstuhl. Th. Rubiano and Th. Seiller are supported by the Île-de-France region through the DIM RFSI project "CoHOp".

Acknowledgements The authors wish to express their gratitude to Assya Sellak for her contribution to this work, to the reviewers of previous versions for their comments, and to the FSCD community: in particular, the reviews we received were extremely interesting, and generated new directions and questions, for which we are thankful.

7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022).
Editor: Amy P. Felty; Article No. 26; pp. 26:1–26:23
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction: letting ICC drive the development of static analyzers

Certifying program resource usages is possibly as crucial as the specification of program correctness, since a guaranteed correct program whose memory usage exceeds available resources is, in fact, unreliable. The field of Implicit Computational Complexity (ICC) theory [15] pioneers in "embedding" in the program itself a guarantee of its resource usage, using e.g., bounded recursion [8, 27] or type systems [6, 26]. This field initiated numerous distinct and original approaches, primarily to characterize complexity classes in a machine-independent way, with increasing expressivity, but these approaches have rarely materialized into concrete programming languages or program analyzers: even if, as opposed to traditional complexity, its models are generally expressive enough to write down actual algorithms [30, p. 11], they rarely escape the sphere of academia or extend beyond toy languages, with a few exceptions [5, 22]. However, by abstracting away constant factors and insignificant orders of magnitude, it is frequently conjectured that ICC will allow sidestepping some of the difficult issues one usually has to face when inferring the resource usage of a concrete program.

This work reinforces this conjecture by adjusting, improving and implementing an existing ICC technique, the *mwp-bounds analysis* [23], which certifies that the values computed by an imperative program will be bounded by polynomials in the program's input. This flow analysis is elegant but computationally costly, and it missed an opportunity to leverage its built-in compositionality: we address both issues by revisiting and expanding the original flow calculus, and further make our point by implementing it on a subset of the `C` programming language. While the theory has been improved to allow analysis of function definitions and calls – including recursive ones, a feature not widely supported [21, p. 359] – , its integration into the implementation is underway, as we placed primary focus on developing an efficient and implementable technique for program analysis. Implementing a tool along the theory enabled testing improvements in real-life, which in return drove adjustments to the theory.

Our enhanced technique answers positively two questions asked by the authors of the original analysis [23, Section 1.2], namely

1. Can the method be extended to richer languages?
2. Can it lead to powerful and convenient tools?

It also supports the conjecture that ICC can be used to construct concrete tools, but highlights that doing so requires adjusting the theory to make it tractable in practice. This work also provides better insight into the original analysis, by e.g., separating the algorithm to decide the existence of a bound from its evaluation into a concrete bound; and by illustrating its plasticity: while our analysis conservatively extends the original one, it nevertheless greatly alters its internal machinery to ease its implementability. Last but not least, our technique is orthogonal to most static analysis methods, which focus on worst-case resource-usage complexity or termination, while ours establishes that the growth rate of variables values is at most polynomially related to their inputs.

Our paper starts by recalling the "original" *mwp-bounds analysis* [23] – to which we refer for a more gentle introduction – and discuss its limitations (Sect. 2). In Sect. 3, we motivate, introduce and justify two modifications to this original analysis, and state that this calculus can be reduced to the original one. We then extend this analysis along two axis (Sect. 4): we detail how functions calls can be analyzed, and how the structures we implemented allowed to speed up some very costly operations. Finally, Sect. 5 presents and discuss our implementation, and Sect. 6 concludes. The proofs, some additional details on semi-rings and the detail of our benchmarks are in appendix, with the exception of some tedious proofs relative to semi-rings that are only in our technical report [4].

6 2 Background: the original flow analysis

The original analysis [23] computes a polynomial bound – if it exists – on the sizes (of the value itself) of variables in an imperative `while` programming language, extended with a `loop` operator, by computing for each variable a vector that tracks how it depends on other variables – and the program itself gets assigned a matrix collecting those vectors. While this does not ensure termination, it provides a certificate guaranteeing that the program uses throughout its execution at most a polynomial amount of space, and as a consequence that *if* it terminates, it will do so in polynomial time.

2.1 Language analyzed: fragments of imperative language

▶ **Definition 1** (Imperative Language)**.** *Letting natural number variables range over* `X` *and* `Y` *and boolean expressions over* `b`*, we define* expressions `e` *and* commands `C` *as follows:*

$$\begin{aligned} \mathtt{e} &:= \mathtt{X} \parallel \mathtt{X\ -\ Y} \parallel \mathtt{X\ +\ Y} \parallel \mathtt{X\ *\ Y} \\ \mathtt{C} &:= \mathtt{X\ =\ e} \parallel \mathtt{if\ b\ then\ C\ else\ C} \parallel \mathtt{while\ b\ do\ \{C\}} \parallel \mathtt{loop\ X\ \{C\}} \parallel \mathtt{C\ ;\ C} \end{aligned}$$

where `loop X {C}` *means "do* `C` `X` *times" and* `C;C` *is used for sequentiality ("do* `C`*, then* `C`*"). We write "program" for a series of commands composed sequentially.*

This language assumes that the program's inputs are the only variables, and that assigning a value to a variable inside the program is not permitted. Extending flow calculi to those operations has been discussed [23, p. 3] and proven possible [9], but we leave this for future work – in particular, our `C` examples will be of `foo` functions with their variables listed as parameters[1]. However, we disallow w.l.o.g. composed expressions of the form `X + Y * Y`, which can always be dealt with in the style of three-address code.

2.2 A flow calculus of mwp-bounds for complexity analysis

Flows characterize controls from one variable to another, and can be, in increasing growth rate, of type 0 – the absence of any dependency – *m*aximum, *w*eak polynomial and *p*olynomial. The bounds on programs written in the syntax of Sect. 2.1 are represented and calculated thanks to vectors and matrices whose coefficients are elements of the mwp semi-ring.

▶ **Definition 2** (The mwp semi-ring and matrices over it)**.** *Letting* $\textsc{mwp} = \{0, m, w, p\}$ *with* $0 < m < w < p$*, and* α*,* β*,* γ *range over* $\textsc{mwp}$*, the* mwp semi-ring $(\textsc{mwp}, 0, m, +, \times)$ *is defined with* $+ = \max$*,* $\alpha \times \beta = \max(\alpha, \beta)$ *if* $\alpha, \beta \neq 0$*, and* 0 *otherwise.*

We denote $\mathbb{M}(\textsc{mwp})$ *the matrices over* $\textsc{mwp}$*, and, fixing* $n \in \mathbb{N}$*,* M *for* $n \times n$ *matrices over* $\textsc{mwp}$*,* M_{ij} *for the coefficient in the* i*th row and* j*th column of* M*,* $\oplus$ *for the componentwise addition, and* $\otimes$ *for the product of matrices defined in a standard way. The* 0*-element for addition is* $0_{ij} = 0$ *for all* i, j*, and the* 1*-element for product is* $1_{ii} = m$*,* $1_{ij} = 0$ *if* $i \neq j$*, and the resulting structure* $(\mathbb{M}(\textsc{mwp}), 0, 1, \otimes, \oplus)$ *is a semi-ring that we simply write* $\mathbb{M}(\textsc{mwp})$*. The closure operator* $\cdot^*$ *is* $M^* = 1 \oplus M \oplus (M^2) \oplus \ldots$*, for* $M^0 = 1$*,* $M^{m+1} = M \otimes M^m$*.*

[1] Our implementation allows to relax this condition, as exemplified in `inline_variable.c`, without losing any of the results expressed in this paper. Assuming a fixed number of variables, known ahead of time, is mostly a theoretical artifact used to simplify the analysis.

$$\frac{}{\vdash_{\mathrm{JK}} \texttt{Xi} : \{^{m}_{\texttt{i}}\}}\ \mathrm{E1} \qquad\qquad \frac{}{\vdash_{\mathrm{JK}} \texttt{e} : \{^{w}_{\texttt{i}} \mid \texttt{Xi} \in \mathrm{var}(\texttt{e})\}}\ \mathrm{E2}$$

$$\star \in \{+,-\}\ \frac{\vdash_{\mathrm{JK}} \texttt{Xi} : V_1 \quad \vdash_{\mathrm{JK}} \texttt{Xj} : V_2}{\vdash_{\mathrm{JK}} \texttt{Xi}\star\texttt{Xj} : pV_1 \oplus V_2}\ \mathrm{E3} \qquad \star \in \{+,-\}\ \frac{\vdash_{\mathrm{JK}} \texttt{Xi} : V_1 \quad \vdash_{\mathrm{JK}} \texttt{Xj} : V_2}{\vdash_{\mathrm{JK}} \texttt{Xi}\star\texttt{Xj} : V_1 \oplus pV_2}\ \mathrm{E4}$$

(a) Rules for assigning vectors to expressions.

$$\frac{\vdash_{\mathrm{JK}} \texttt{e} : V}{\vdash_{\mathrm{JK}} \texttt{Xj = e} : 1 \overset{\texttt{j}}{\leftarrow} V}\ \mathrm{A} \qquad \frac{\vdash_{\mathrm{JK}} \texttt{C1} : M_1 \quad \vdash_{\mathrm{JK}} \texttt{C2} : M_2}{\vdash_{\mathrm{JK}} \texttt{C1; C2} : M_1 \otimes M_2}\ \mathrm{C}$$

$$\frac{\vdash_{\mathrm{JK}} \texttt{C1} : M_1 \quad \vdash_{\mathrm{JK}} \texttt{C2} : M_2}{\vdash_{\mathrm{JK}} \texttt{if b then C1 else C2} : M_1 \oplus M_2}\ \mathrm{I}$$

$$\forall i, M^*_{ii} = m \quad \frac{\vdash_{\mathrm{JK}} \texttt{C} : M}{\vdash_{\mathrm{JK}} \texttt{loop X1 \{C\}} : M^* \oplus \{^{p}_{\texttt{1}} \rightarrow j \mid \exists i, M^*_{ij} = p\}}\ \mathrm{L}$$

$$\forall i, M^*_{ii} = m \text{ and } \forall i,j, M^*_{ij} \neq p \quad \frac{\vdash_{\mathrm{JK}} \texttt{C} : M}{\vdash_{\mathrm{JK}} \texttt{while b do \{C\}} : M^*}\ \mathrm{W}$$

(b) Rules for assigning matrices to commands.

Figure 1 Original non-deterministic ("Jones-Kristiansen") flow analysis rules.

Although not crucial to understand our development, details about (strong) semi-rings and the mwp semi-ring, and the construction of a semi-ring whose elements are matrices with coefficients in a semi-ring – so, in particular, $\mathbb{M}(\text{MWP})$ – are given in our technical report [4, A.1 and A.2] and sketched in appendix Appendix A.

Below, we let V_1, V_2 be column vectors with values in MWP, αV_1 be the usual scalar product, and $V_1 \oplus V_2$ be defined componentwise. We write $\{^{\alpha}_{i}\}$ for the vector with 0 everywhere except for α in its ith row, and $\{^{\alpha}_{i},^{\beta}_{j}\}$ for $\{^{\alpha}_{i}\} \oplus \{^{\beta}_{j}\}$.

Replacing in a matrix M the jth column vector by V is denoted $M \overset{j}{\leftarrow} V$. The matrix M with $M_{ij} = \alpha$ and 0 everywhere else is written $\{^{\alpha}_{i} \rightarrow j\}$, and the set of variables in the expression $\texttt{e}$ is written $\mathrm{var}(\texttt{e})$. The assumption is made that exactly n different variables are manipulated throughout the analyzed program, so that n-vectors are assigned to expressions – in a non-deterministic way, to capture larger classes of programs [23, Section 8] – and $n \times n$ matrices are assigned to commands using the rules presented Fig. 1 [23, Section 5].

The intuition is that if $\vdash_{\mathrm{JK}} \texttt{C} : M$ can be derived, then all the values computed by $\texttt{C}$ will grow at most polynomially w.r.t. its inputs [23, Theorem 5.3], e.g., will be bounded by $\max(\vec{x}, p_1(\vec{y})) + p_2(\vec{z})$, where p_1 and p_2 are polynomials and $\vec{x}$ (resp. $\vec{y}$, $\vec{z}$) are m-(resp. w-, p-)annotated variables in the vector for the considered output. Since the derivation system is non-deterministic, multiple matrices and polynomial bounds – that sometimes coincide – may be assigned to the same program. Furthermore, the coefficient at $M_{\texttt{ij}}$ carries quantitative information about the way $\texttt{Xi}$ depends on $\texttt{Xj}$, knowing that 0- and m-flows are harmless and without constraints, but that w- and p- flows are more harmful w.r.t. polynomial bounds and need to be handled with care, particularly in loops – hence the condition on the L and W rules. The derivation may fail – some programs may not be assigned a matrix – if at least one of the variables used in the body of a loop depends "too strongly" upon another, making it impossible to ensure polynomial bounds on the loop itself. We will use the following example as a common basis to discuss possible failure, non-determinism, and our improvements.

8 ▶ **Example 3.** Consider `loop X3 {X2 = X1 + X2}`. The body of the `loop` command admits three different derivations, obtained by applying A to one of the three derivation of the expression `X1 + X2`, that we name π_0, π_1 and π_2:

$$\dfrac{\dfrac{}{\vdash_{\text{JK}} \texttt{X1} : \begin{pmatrix} m \\ 0 \\ 0 \end{pmatrix}}\text{E1} \qquad \dfrac{}{\vdash_{\text{JK}} \texttt{X2} : \begin{pmatrix} 0 \\ m \\ 0 \end{pmatrix}}\text{E1}}{\vdash_{\text{JK}} \texttt{X1 + X2} : \begin{pmatrix} p \\ m \\ 0 \end{pmatrix}}\text{E3} \qquad \dfrac{\dfrac{}{\vdash_{\text{JK}} \texttt{X1} : \begin{pmatrix} m \\ 0 \\ 0 \end{pmatrix}}\text{E1} \qquad \dfrac{}{\vdash_{\text{JK}} \texttt{X2} : \begin{pmatrix} 0 \\ m \\ 0 \end{pmatrix}}\text{E1}}{\vdash_{\text{JK}} \texttt{X1 + X2} : \begin{pmatrix} m \\ p \\ 0 \end{pmatrix}}\text{E4} \qquad \dfrac{}{\vdash_{\text{JK}} \texttt{X1 + X2} : \begin{pmatrix} w \\ w \\ 0 \end{pmatrix}}\text{E2}$$

From π_0, the derivation of `loop X3 {X2 = X1 + X2}` can be completed using A and L, but since L requires having only m coefficients on the diagonal, π_1 cannot be used to complete the derivation, because of the p coefficient in a box below:

$$\dfrac{\dfrac{\dfrac{\vdots\, \pi_0}{\vdash_{\text{JK}} \texttt{X1 + X2} : \begin{pmatrix} p \\ m \\ 0 \end{pmatrix}}}{\vdash_{\text{JK}} \texttt{X2 = X1 + X2} : \begin{pmatrix} m & p & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{pmatrix}}\text{A}}{\vdash_{\text{JK}} \texttt{loop X3 \{X2 = X1 + X2\}} : \begin{pmatrix} m & p & 0 \\ 0 & m & 0 \\ 0 & p & m \end{pmatrix}}\text{L} \qquad \dfrac{\dfrac{\vdots\, \pi_1}{\vdash_{\text{JK}} \texttt{X1 + X2} : \begin{pmatrix} m \\ p \\ 0 \end{pmatrix}}}{\vdash_{\text{JK}} \texttt{X2 = X1 + X2} : \begin{pmatrix} m & m & 0 \\ 0 & \boxed{p} & 0 \\ 0 & 0 & m \end{pmatrix}}\text{A}$$

Similarly, using A after π_2 gives a w coefficient on the diagonal and makes it impossible to use L, hence only one derivation for this program exists.

2.3 Limitations and inefficiencies of the mwp analysis

Even if the proof techniques are far from trivial, with only 9 rules and skipping over boolean expressions (observe that the condition `b` has no impact in the rules I or W), the analysis is flexible and easy to carry out – at least mathematically. It also has inherent limitations: while the technique is sound, it is not complete and programs such as greatest common divisor fail to be assigned a matrix. We will discuss in Sect. 5.2, the benefits and originality of this analysis, but we would now like to stress how it is computationally inefficient, since the non-determinacy makes the analysis costly to carry out and can lead to memory explosions.

Abstracting Example 3, one can see that the base case of non-determinism – e.g., to assign a vector to `X1` $\star$ `X2`–yields vectors $\left(\begin{smallmatrix} p \\ m \end{smallmatrix}\right)$ (using E1 then E3), $\left(\begin{smallmatrix} m \\ p \end{smallmatrix}\right)$ (using E1 then E4) and $\left(\begin{smallmatrix} w \\ w \end{smallmatrix}\right)$ (using E2). Since none of those vectors is less than the others, only two strategies are available to analyze a larger program containing `X1` $\star$ `X2`: either the derivations for this base case are considered one after the other, or they are all stored in memory at the same time. Considering the derivations for the base case one after the other can lead to a time explosion, as a program of n lines can have 3^n different derivations – as exemplified by `explosion.c`, a simple series of applications – and it is possible that only one of them can be completed, so all must be explored. On the other hand, storing those three vectors and constructing all the matrices in parallel leads to a memory explosion: the analysis for two commands involving 6 variables, with 3 choices – which cannot be simplified as explained previously – would result in 9 matrices of size 6×6, i.e., 324 coefficients. All in all, a program of n lines with x different variables can require c_1^n different derivations, which can produce up to $(c_2 \times x)^2$ coefficients to store for some constants c_1, c_2.

Beyond inefficiency, there are additional limitations: while the analysis is naturally compositional, this feature is not leveraged in the original system; furthermore, an occurrence of non-polynomial flows in the matrix causes the analysis to simply stop, thus not capturing failure in a meaningful way. We will discuss our solutions to these deficiencies next.

$$\star \in \{+,-\}\ \frac{}{\vdash \mathtt{Xi{\star}Xj} : (0 \mapsto \{{}^{m}_{\mathtt{i}}, {}^{p}_{\mathtt{j}}\}) \oplus (1 \mapsto \{{}^{p}_{\mathtt{i}}, {}^{m}_{\mathtt{j}}\}) \oplus (2 \mapsto \{{}^{w}_{\mathtt{i}}, {}^{w}_{\mathtt{j}}\})}\ \mathrm{E}^{\mathrm{A}}$$

$$\frac{}{\vdash \mathtt{Xi\ *\ Xj} : \{{}^{w}_{\mathtt{i}}, {}^{w}_{\mathtt{j}}\}}\ \mathrm{E}^{\mathrm{M}} \qquad \frac{}{\vdash \mathtt{Xi} : \{{}^{m}_{\mathtt{i}}\}}\ \mathrm{E}^{\mathrm{S}}$$

(a) Rules for assigning vectors to expressions.

$$\frac{\vdash \mathtt{e} : V}{\vdash \mathtt{Xj\ =\ e} : 1 \overset{\mathtt{j}}{\leftarrow} V}\ \mathrm{A} \qquad \frac{\vdash \mathtt{C1} : M_1 \quad \vdash \mathtt{C2} : M_2}{\vdash \mathtt{C1;\ C2} : M_1 \otimes M_2}\ \mathrm{C} \qquad \frac{\vdash \mathtt{C1} : M_1 \quad \vdash \mathtt{C2} : M_2}{\vdash \mathtt{\textbf{if}\ b\ \textbf{then}\ C1\ \textbf{else}\ C2} : M_1 \oplus M_2}\ \mathrm{I}$$

$$\frac{\vdash \mathtt{C} : M}{\vdash \mathtt{\textbf{loop}\ Xl\ \{C\}} : M^* \oplus \{{}^{\infty}_{j} \to j \mid M^*_{jj} \neq m\} \oplus \{{}^{p}_{\mathtt{l}} \to j \mid \exists i, M^*_{ij} = p\}}\ \mathrm{L}^{\infty}$$

$$\frac{\vdash \mathtt{C} : M}{\vdash \mathtt{\textbf{while}\ b\ \textbf{do}\ \{C\}} : M^* \oplus \{{}^{\infty}_{j} \to j \mid M^*_{jj} \neq m\} \oplus \{{}^{\infty}_{i} \to j \mid M^*_{ij} = p\}}\ \mathrm{W}^{\infty}$$

(b) Rules for assigning matrices to commands.

Figure 2 Deterministic improved flow analysis rules.

3 A deterministic, always-terminating, declension of the mwp analysis

The problem of finding a derivation in the original calculus is in NP [23, Theorem 8.1]. But since all the non-determinism is in the rules to assigning a vector, the potentially exponential number of derivations are actually extremely similar. Hence, instead of having the analysis stop when failing to establish a derivation and re-starting from scratch, storing the different vectors and constructing the derivation while keeping all the options open seems to be a better strategy, but, as we have seen, this causes a memory blow-up. We address it by fine-tuning the internal machinery: to represent non-determinism, we let the matrices take as values either functions from choices to coefficients in MWP or coefficients in MWP, so that instead of mapping choices to derivations, all the derivations are represented by the same matrix that internalizes the different choices. Sect. 3.1 discusses this improvement, which results in a notable gain: getting back to the example of Sect. 2.3, a program involving 6 variables, with 3 choices, would now be assigned a (unique) 6×6 matrix that requires 66 coefficients instead of the 324 we previously had – this is because 30 coefficients are "simple" values in MWP, and 6 are functions from a set of choices $\{0, 1, 2\}$ to values in MWP, each represented with 6 coefficients.

For the choices that give coefficients fulfilling the side condition of L or W, the derivation can proceed as usual, but when a particular choice gives a coefficient that violates it, we decided against simply removing it. Instead, to guarantee that all derivations always terminate, we mark that choice by indicating that it would not provide a polynomial bound. This requires extending the MWP semi-ring with a special value ∞ that represents failure in a local way, marking non-polynomial flows, and is detailed in Sect. 3.2. As a by-product, this enables fine-grained information on programs that *do not* have polynomially bounded growth, since the precise dependencies that break this growth rate can be localized.

Taken together (Sect. 3.3), our improvements ensure that exactly one matrix will always be assigned to a program while carrying over the correctness of the original analysis. We give in Fig. 2 the deterministic system we are introducing in full, but will gently introduce it though the remaining parts of this section: note that the rules A, C and I are unchanged, up to the fact that the matrices, sum and product are in a different semi-ring.

3.1 Internalizing non-determinism: the choice data flow semi-rings

Internalizing the choice requires altering the semi-ring used in the analysis: we want to replace the three vectors over MWP that can be assigned to an expression by a single vector over $\{0,1,2\} \to \text{MWP}$ that captures the same three choices. For a program needing to decide p times between the 3 available choices, this means replacing the $3 \times p$ different matrices in $\mathbb{M}(\text{MWP})$ by a single matrix in $\mathbb{M}(\{0,1,2\}^p \to \text{MWP})$. For any strong semi-ring $\mathbb{S}$ and family of sets $(A_i)_{i=1,\dots,p}$, both $A_i \to \mathbb{S}$ and $\mathbb{M}(\prod_{i=1}^p A_i \to \mathbb{S})$ are semi-rings, using the usual cartesian product of sets, and there exists an isomorphism $\mathbb{M}(\prod_{i=1}^p A_i \to \mathbb{S}) \cong \prod_{i=1}^p A_i \to \mathbb{M}(\mathbb{S})$ [4, A.3]. This dual nature of the semi-ring considered is useful:

- the analysis will now assign an element M of $\mathbb{M}(\prod_{i=1}^p A_i \to \text{MWP})$ to a program;
- representing M as an element of $\prod_{i=1}^p A_i \to \mathbb{M}(\text{MWP})$ allows one to use an *assignment* $\vec{a} = (a_1, \dots, a_p) \in \prod_{i=1}^p A_i$ to produce a matrix $M[\vec{a}] \in \mathbb{M}(\text{MWP})$, recovering the mwp-flow that would have been computed by making the choices $a_1, \dots, a_p$ in the derivation.

▶ Remark 4. As the unique degree of non-determinism to assign a matrix to commands is 3, our modification of the analysis flow consists simply of recording the different choices by letting $A_i = \{0,1,2\}$ for all $i = 1, \dots, p$ where p is the number of times a choice had to be taken. Starting with Sect. 4, function calls will require potentially different sets A_i.

▶ Notation 5. In the following and in the implementation alike, we will denote a function $(a_1^0 \times \cdots \times a_p^0 \mapsto \alpha_0) + \cdots + (a_1^k \times \cdots \times a_p^k \mapsto \alpha_k)$ in $A^p \to \text{MWP}$ with $\text{Card}(A) = k$ by, omitting the product, $(\alpha_0 \delta(a_1^0, 0) \cdots \delta(a_p^0, p)) + \cdots + (\alpha_k \delta(a_1^k, 0) \cdots \delta(a_p^k, p))$, with $\delta(i,j) = m$ if the jth choice is i, 0 otherwise. Example 8 will justify and explain this choice.

Our derivation system replaces the E3 and E4 rules with a single rule E^{A} ("additive"), and splits E2 in two exclusive rules, E^{M} for "multiplicative" and E^{S} for "simple" (atomic) expressions – Theorem 11 will prove how they are equivalent.

▶ **Example 6.** We represent the vectors $\begin{pmatrix} p \\ m \\ 0 \end{pmatrix}$, $\begin{pmatrix} m \\ p \\ 0 \end{pmatrix}$ and $\begin{pmatrix} w \\ w \\ 0 \end{pmatrix}$ from Example 3 with a single vector $\begin{pmatrix} p\delta(0,0)+m\delta(1,0)+w\delta(2,0) \\ m\delta(0,0)+p\delta(1,0)+w\delta(2,0) \\ 0 \end{pmatrix}$, that can be read as $\begin{pmatrix} \{0 \mapsto p, 1 \mapsto m, 2 \mapsto w\} \\ \{0 \mapsto m, 1 \mapsto p, 2 \mapsto w\} \\ 0 \end{pmatrix}$, where we write 0 for $\{0 \mapsto 0, 1 \mapsto 0, 2 \mapsto 0\}$[2]. Since in particular[3], $\mathbb{M}(\{0,1,2\} \to \text{MWP}) \cong \{0,1,2\} \to \mathbb{M}(\text{MWP})$, the obtained vector can be rewritten as $0 \mapsto \begin{pmatrix} p \\ m \\ 0 \end{pmatrix}, 1 \mapsto \begin{pmatrix} m \\ p \\ 0 \end{pmatrix}, 2 \mapsto \begin{pmatrix} w \\ w \\ 0 \end{pmatrix}$.

3.2 Internalizing failure: de-correlating derivations and bounds

The original analysis stops when detecting a non-polynomial flow, puts an end to the chosen strategy (i.e., set of choices) and restarts from scratch with another one. We adapt the rules so that every derivation can be completed even in the presence of non-polynomial flows, thanks to a new top element, ∞, representing failure in a local way.

Ignoring our previous modification in this subsection, the semi-ring MWP^∞ we need to consider is $(\text{MWP} \cup \{\infty\}, 0, m, +^\infty, \times^\infty)$, with $\infty > \alpha$ for all $\alpha \in \text{MWP}$, $+^\infty = \max$ as before, and $\alpha \times^\infty \beta = 0$ if $\alpha, \beta \neq \infty$ and α or β is 0, $\max(\alpha, \beta)$ otherwise. This different condition in the definition of $\times^\infty$ ensures that once non-polynomial flows have been detected, they cannot be erased (as $\infty \times^\infty 0 = \infty$).

[2] The implementation supports both coefficients from MWP *and* coefficients from $\{0,1,2\}^p \to \text{MWP}$, cf. e.g., a simple assignment example `assign_expression.c`.

[3] This is a variant of Lemma 21 [4, A.3]. While the latter lemma applies to algebras of square matrices, a similar result holds for rectangular matrices of a fixed size; the algebraic structure is no longer that of a semi-ring as rectangular matrices do not possess a proper multiplication, but the proof can be adapted to show the existence of an isomorphism of modules between the considered spaces.

11 The only cases where the original analysis may fail is if the side conditions of L or W (Fig. 1) are not met. We replace those by L^∞ and W^∞ (Fig. 2), which replace the problematic coefficients with ∞, marking non-polynomial dependencies, and carry on the analysis.

▶ **Example 7.** The program from Example 3 would now receive three derivations (omitting the one obtained from π_0, as the resulting matrix is identical):

$$\dfrac{\dfrac{\dfrac{\vdots\ \pi_1}{\vdash \texttt{X1 + X2} : \begin{pmatrix} m \\ p \\ 0 \end{pmatrix}}}{\vdash \texttt{X2 = X1 + X2} : \begin{pmatrix} m & m & 0 \\ 0 & p & 0 \\ 0 & 0 & m \end{pmatrix}}\ \mathrm{A}}{\vdash \texttt{loop X3 \{X2 = X1 + X2\}} : \begin{pmatrix} m & p & 0 \\ 0 & \infty & 0 \\ 0 & p & m \end{pmatrix}}\ \mathrm{L}^\infty \qquad \dfrac{\dfrac{\dfrac{}{\vdash \texttt{X1 + X2} : \begin{pmatrix} w \\ w \\ 0 \end{pmatrix}}\ \mathrm{E2}}{\vdash \texttt{X2 = X1 + X2} : \begin{pmatrix} m & w & 0 \\ 0 & w & 0 \\ 0 & 0 & m \end{pmatrix}}\ \mathrm{A}}{\vdash \texttt{loop X3 \{X2 = X1 + X2\}} : \begin{pmatrix} m & w & 0 \\ 0 & \infty & 0 \\ 0 & 0 & m \end{pmatrix}}\ \mathrm{L}^\infty$$

Of course, neither of those two derivations would yield polynomial bound – since they contain ∞ coefficients – but it becomes possible to determine that the last one is “better” – since $\begin{pmatrix} p \\ \infty \\ p \end{pmatrix} > \begin{pmatrix} w \\ \infty \\ 0 \end{pmatrix}$ – and to observe how their “failure” would propagate in larger programs, possibly establishing that one fares better than the other in terms of non-polynomial growths. This could imply, for instance, that particular programs without polynomial bounds could still be considered “reasonnable” if they are exponential only in some variables that are known to have smaller values in input.

3.3 Merging the improvements: illustrations and proofs

We prove that our system captures the original system in the sense that set aside ∞ coefficients, both systems agree (Theorem 11), but also that exactly one matrix is produced per program (Theorem 10) – i.e., that we can analyze as many programs as originally, and still be correct regarding the bounds. Before doing so, we would like to give more specifics on our system, by combining the semi-rings and intuitions from the previous two subsections. We have discussed our “axiomatic” (E^A, E^M, E^S) and “loop” rules (L^∞ and W^∞), but remain to discuss the rules for assignment (A), **`if`** (I) and composition (C) – which is where both improvements meet. Mathematically speaking, adopting the semi-ring defined over matrices with coefficients in $\{0, 1, 2\}^p \to \textsc{mwp} \cup \{\infty\}$ is straightforward, and we simply write $\oplus$ and $\otimes$ the operations resulting from merging the two transformations. We discuss in Sect. 4.3 how, however, those operations are computationally costly and how we address this challenge.

▶ **Example 8.** Using our deterministic system presented in Fig. 2, consider the following:

$$\dfrac{\dfrac{\dfrac{}{\vdash \texttt{X1 + X2} : V}\ \mathrm{E}^\mathrm{A}}{\vdash \texttt{X1 = X1 + X2} : 1 \overset{1}{\leftarrow} V}\ \mathrm{A} \qquad \dfrac{\dfrac{}{\vdash \texttt{X1 - X3} : V'}\ \mathrm{E}^\mathrm{A}}{\vdash \texttt{X1 = X1 - X3} : 1 \overset{1}{\leftarrow} V'}\ \mathrm{A}}{\vdash \texttt{if b then \{X1 = X1 + X2\} else \{X1 = X1 - X3\}} : (1 \overset{1}{\leftarrow} V) \oplus (1 \overset{1}{\leftarrow} V')}\ \mathrm{I}$$

with

$$\begin{aligned} V &= 0 \mapsto \left\{\genfrac{}{}{0pt}{}{m}{\mathsf{1}},\genfrac{}{}{0pt}{}{p}{\mathsf{2}}\right\} \oplus 1 \mapsto \left\{\genfrac{}{}{0pt}{}{p}{\mathsf{1}},\genfrac{}{}{0pt}{}{m}{\mathsf{2}}\right\} \oplus 2 \mapsto \left\{\genfrac{}{}{0pt}{}{w}{\mathsf{1}},\genfrac{}{}{0pt}{}{w}{\mathsf{2}}\right\} \\ V' &= 0 \mapsto \left\{\genfrac{}{}{0pt}{}{m}{\mathsf{1}},\genfrac{}{}{0pt}{}{p}{\mathsf{3}}\right\} \oplus 1 \mapsto \left\{\genfrac{}{}{0pt}{}{p}{\mathsf{1}},\genfrac{}{}{0pt}{}{m}{\mathsf{3}}\right\} \oplus 2 \mapsto \left\{\genfrac{}{}{0pt}{}{w}{\mathsf{1}},\genfrac{}{}{0pt}{}{w}{\mathsf{3}}\right\} \\ 1 \overset{1}{\leftarrow} V &\cong \begin{pmatrix} (0\mapsto m)\oplus(1\mapsto p)\oplus(2\mapsto w) & 0 & 0 \\ (0\mapsto p)\oplus(1\mapsto m)\oplus(2\mapsto w) & m & 0 \\ 0 & 0 & m \end{pmatrix} = \begin{pmatrix} m\delta(0,0)\oplus p\delta(1,0)\oplus w\delta(2,0) & 0 & 0 \\ p\delta(0,0)\oplus m\delta(1,0)\oplus w\delta(2,0) & m & 0 \\ 0 & 0 & m \end{pmatrix} \\ 1 \overset{1}{\leftarrow} V' &\cong \begin{pmatrix} (0\mapsto m)\oplus(1\mapsto p)\oplus(2\mapsto w) & 0 & 0 \\ 0 & m & 0 \\ (0\mapsto p)\oplus(1\mapsto m)\oplus(2\mapsto w) & 0 & m \end{pmatrix} = \begin{pmatrix} m\delta(0,1)\oplus p\delta(1,1)\oplus w\delta(2,1) & 0 & 0 \\ 0 & m & 0 \\ p\delta(0,1)\oplus m\delta(1,1)\oplus w\delta(2,1) & 0 & m \end{pmatrix} \end{aligned}$$

Some care is needed to perform the addition for the I rule: the choices in the left and right branches are independent, so we must use coefficients in $\{0,1,2\}^2 \to \textsc{mwp}$ for the 2^3 choices. While the mapping notation would require to use positions to describe which choice is being refereed to, the δ notation makes it immediate, as it encodes in the second value of δ that two choices are considered, numbering the choice in the left branch 0. Hence we can sum the coefficients and obtain the matrix that can be observed in our implementation by analyzing `example7.c`.

▶ **Example 9.** Our deterministic system now assigns to `loop X3 {X2 = X1 + X2}` from Example 3 the unique matrix

$$\begin{pmatrix} m & (0{\mapsto}p)\oplus(1\ {\mapsto}m)\oplus(2{\mapsto}w) & 0 \\ 0 & (0{\mapsto}m)\oplus(1\ {\mapsto}\infty)\oplus(2{\mapsto}\infty) & 0 \\ 0 & (0{\mapsto}p)\oplus(1\ {\mapsto}0)\oplus(2{\mapsto}0) & m \end{pmatrix} = \begin{pmatrix} m & p\delta(0,0)\oplus m\delta(1,0)\ \oplus w\delta(2,0) & 0 \\ 0 & m\delta(0,0)\oplus\infty\delta(1,0)\oplus\infty\delta(2,0) & 0 \\ 0 & p\delta(0,0)\oplus 0\delta(1,0)\oplus 0\delta(2,0) & m \end{pmatrix}$$

where we observe that

1. only one choice, one assignment, 0, gives a matrix without ∞ coefficient, corresponding to the fact that, in the original system, only π_0 could be used to complete the proof,
2. the choice impacts the matrix locally, the coefficients being mostly the same, independently from the choice,
3. the influence of `X2` on itself is where possible non-polynomial growth rates lies, as the ∞ coefficient are in the second column, second row.

We are now in possession of all the material and intuitions needed to state the correspondence between our system and the original one of Jones and Kristiansen.

▶ **Theorem 10** (Determinancy and termination). *Given a program P, there exists unique $p \in \mathbb{N}$ and $M \in \mathbb{M}(\{0,1,2\}^p \to \textsc{mwp}^\infty)$ such that $\vdash P : M$.*

Proof. The existence of the matrix is guaranteed by the completeness of the rules, as any program written in the syntax presented in Sect. 2.1 can be typed with the rules of Fig. 2. The uniqueness of the matrix is given by the fact that no two rules can be applied to the same command. Details are provided in Appendix B. ◀

▶ **Theorem 11** (Adequacy). *If $\vdash P : M$, then for all $\vec{a} \in A^p$, $\vdash_{\text{JK}} P : M[\vec{a}]$ iff $\infty \notin M[\vec{a}]$.*

Proof. The proof uses that P cannot be assigned a matrix in the original calculus iff the deterministic calculus introduce a ∞ coefficient, and from the fact that both calculus coincide in all the other cases. Details are provided in Appendix B. ◀

▶ **Corollary 12** (Soundness). *If $\vdash P : M$ and there exists $\vec{a} \in A^p$ such that $\infty \notin M[\vec{a}]$, then every value computed by P is bounded by a polynomial in the inputs.*

Proof. This is an immediate corollary of the original soundness theorem [23, Theorem 5.3] and of Theorem 11. ◀

This proves that the two analyses coincide, when excluding ∞, and that we can re-use the original proofs. However, our alternative definition should be understood as an important improvement, as it enables a better proof-search strategy while optimizing the memory usage, and hence enables the implementation (Sect. 5). It also lets the programmer gain more fine-grained feedback, and illustrates the flexibility of the analysis: the latter will also be demonstrated by the improvements we discuss in the next section.

13

4 Extending and improving the analysis: functions and efficiency

To improve this analysis, one could try to extract a tight bound, to certify it, or to port it to a compiler's intermediate representation. Adding constant values is arguably immediate [23, p. 3] but handling pointers, even if technically possible, would probably require significant work. This illustrates at the same time the flexibility of the analysis, and the distance separating ICC-inspired techniques from their usage on actual programs. We decided to narrow this gap along two axes: the first one consists of allowing function definitions and calls in our syntax. It is arguably a small improvement, but illustrates nicely the compositionality of the analysis, and includes recursively defined functions. The second extension intersects the theory and the implementation: it details how our semi-ring structure can be leveraged to maintain a tractable algorithm to compute costly operations on our matrices, and to separate the problem of deciding if a bound exists from computing its form.

4.1 Leveraging compositionality to analyze function calls

Thanks to its compositionality, this analysis can easily integrate functions and procedures, by re-using the matrix and choices of a program implementing the function called. We begin by adding to the syntax the possibility of defining multiple functions and calling them:

▶ **Definition 13** (Functions). *Letting* `R` *(resp.* f*) range over variables (resp. function names), we add* function calls[4] *to the commands (Def. 1) and allow* function declarations*:*

C := *`Xi = f(X1, ..., Xn)`* F := *`f(X1, ..., Xn){C; return R}`*

In a function declaration, `f(X1, ..., Xn)` *is called the* header*, and the* body *is simply* `C` *(i.e.,* `return R` *is not part of the body). A* program *is now a series of function declarations such that all the function calls refer to previously declared functions – we deal with recursive calls in Sect. 4.2 – and a* chunk *is a series of commands.*

Now, given a function declaration computing f, we can obtain the matrix M_f by analyzing the body of f as previously done. It is then possible to store the assignments $\vec{a}_0, \dots, \vec{a}_k$, for which no ∞ coefficients appear[5], and to project the resulting matrices to only keep the vector at `R` that provides quantitative information about all the possible dependencies of the output variable `R` w.r.t. input values, possibly merging choices leading to the same result. After this, we are left with a family $(M_f[\vec{a}_0])|_{\texttt{R}}, \dots, (M_f[\vec{a}_k])|_{\texttt{R}}$ of vectors – as the syntax here is restricted to functions with a single output value, even if accommodating multiple return values would be dealt with the same way – that we can re-use when calling the function.

The analysis of the command calling f is then dealt with the F rule below:

$$\frac{}{\vdash \texttt{Xi = F(X1,..., Xn)} : 1 \overset{\texttt{i}}{\leftarrow} (((M_f[\vec{a}_0])|_{\texttt{R}})\delta(0,c) \oplus \cdots \oplus ((M_f[\vec{a}_k])|_{\texttt{R}})\delta(k,c))}\ \text{F}$$

This rule introduces a choice c over k possible matrices, and it is possible that $k \neq 3$, but this is not an issue, since our semi-ring construction can accommodate any set of choice A.

[4] Function calls that discard the output – procedures – could also be dealt with easily, but are vacuous in our effect-free, in particular pointer-free, language

[5] Allowing ∞ coefficients would not change the method described nor its results, but it does not seem relevant to allow calling functions that are not polynomially bounded.

14 ▶ **Example 14.** Consider the following two programs `Q` and `P`:

`Q` =
```
int f(X1, X2){
  while b do {X2=X1+X1};
  return X2;
}
```

`P` =
```
int foo(X1, X2){
  X2=X1+X1;
  X1=f(X2, X2);
}
```

We first have $\vdash$ `X2 = X1 + X1` : V for $V = \left(\begin{smallmatrix} m & p\delta(0,0)\oplus p\delta(1,0)\oplus w\delta(2,0) \\ 0 & 0 \end{smallmatrix}\right)$, and since $V^* = \left(\begin{smallmatrix} m & p\delta(0,0)\oplus p\delta(1,0)\oplus w\delta(2,0) \\ 0 & m \end{smallmatrix}\right)$, applying W$^\infty$ gives $\vdash$ `Q` : $\left(\begin{smallmatrix} m & \infty\delta(0,0)\oplus\infty\delta(1,0)\oplus w\delta(2,0) \\ 0 & m \end{smallmatrix}\right)$. Noting that only one choice gives an ∞-free matrix, we can now carry on the analysis of `P`:

$$\dfrac{\dfrac{\vdots}{\vdash \texttt{X2 = X1 + X1} : V} \qquad \dfrac{}{\vdash \texttt{X1 = f(X2, X2)} : 1 \overset{1}{\leftarrow} \left(\left(\begin{smallmatrix} w \\ m \end{smallmatrix}\right)\delta(0,c)\right)}\ \mathrm{F}}{\vdash \texttt{P} : V \otimes 1 \overset{1}{\leftarrow} \left(\left(\begin{smallmatrix} w \\ m \end{smallmatrix}\right)\delta(0,c)\right)}\ \mathrm{C}$$

In this particular case, the c choice can be discarded, since only one option is available.

Now, to prove that the F rule faithfully extends the analysis (Theorem 17), i.e., preserves Corollary 12, we prove that the analysis of the program "inlining" the function call – as defined below – is, up to some bureaucratic variable manipulation and ignoring some ∞ coefficients, the same as the analysis resulting from using our rule. Intuitively, this mechanism provides the expected result because the choices in the function *do not* affect the program calling it, and because their sets of variables are disjoint – except for the return variable.

▶ **Definition 15** (In-lining function calls)**.** *Let P be a chunk containing a call to the function f, and F be the function declaration computing the function f. The* context *$P[\cdot]$, a chunk containing a slot $[\cdot]$, is obtained by replacing in P the function call `Xi=f(X1, ..., Xn)`, with `X'1=X1; ...; X'n=Xn;` $[\cdot]$ `Xi=R`, for `R`, `X'1`, ..., `X'n` fresh variables added to the header containing the chunk.*

The chunk $\tilde{F}$ is obtained from the body of F by renaming the input variables to `X'1`, ..., `X'n`, and the variable returned by F to `R`. The code $P[F]$ is finally obtained by computing the chunk $\tilde{F}$, and inserting it in place of the symbol $[\cdot]$ in $P[\cdot]$.

That P and $P[F]$ have, at the end of their executions, the same values stored in the variables of P is straightforward in our imperative programming language.

▶ **Example 16.** The in-lining of `Q` in `P` from Example 14 would give the following chunk $\tilde{\texttt{Q}}$ and context `P`$[\cdot]$, `P[Q]` being obtained by replacing in the latter $[\cdot]$ with the former:

$\tilde{Q}$ = `while b do {R=X'1+X'1};`

$P[\cdot]$ =
```
int foo(X1, X2, X'1, R){
    X2=X1+X1;
    X'1=X2;
    [·]
    X1=R;
}
```

The analysis of `P` (excluding the function call) and `Q` is implemented at `example15a.c`, and of `P[Q]` at `example15b.c`: this latter diverges with Example 14 only up to projection and ∞-coefficients that are removed by F but not when in-lining the function call.

Now, we need to prove that the matrices $M(P)$ – obtained by analyzing P and using the F rule for `Xi=f(X1, ..., Xn);` – and $M(P[F])$ – obtained by analyzing the inlined $P[F]$ – are the same. However, to avoid conflict with the variables and to project the matrices on the relevant values, some bureaucracy is needed: we write $\Pi_P(M(P[F]))$ (resp. $(1-\Pi_P)(M(P[F]))$) the projection of $M(P[F])$ onto the variables in (resp. *not* in) P. Some non-deterministic choices may appear within the (modified) chunk $\tilde{F}$ inside $P[F]$, i.e.,

- the coefficients of $M(P)$ are elements of the semi-ring $\prod_{i=1}^{p+1} A_i \to \mathbb{M}(\textsc{mwp})$, with one particular choice corresponding to the F rule – we write the corresponding index i_0;
- the coefficients of $M(P[F])$ are elements of the semi-ring $\prod_{i=1}^{p+k} B_i \to \mathbb{M}(\textsc{mwp})$, where k choices are made within the chunk $\tilde{F}$ – we write the corresponding indexes $j_1, j_2, \dots, j_k$ (note these are in fact consecutive indexes).

We note $\pi : \{1, \dots, p+k\} \to \{1, \dots, p+1\}$ the projection of the choices in $P[F]$ onto the corresponding choices in P, i.e., $\pi(j) = \begin{cases} j & \text{if } j < j_1 \\ i_0 & \text{if } j_1 \leqslant j < j_k \\ j-k+1 & \text{if } j_k < j \end{cases}$. We note that each matrix used as axiom in the function call corresponds to a specific assignment on indexes $j_1, \dots, j_k$. We write $\Psi : A_{i_0} \to \prod_{i=j_1}^{j_k} B_i$ the corresponding injection, extended to $\bar{\Psi} : \prod_{i=1}^{p+1} A_i \to \prod_{i=0}^{p+k} B_i$ straightforwardly.

▶ **Theorem 17.** *For all $\vec{a}$ in $\prod_{i=1}^{p+1} A_i$, $(M(P))[\vec{a}] = (1 - \Pi_P)(M(P[F]))[\bar{\Psi}(\vec{a})]$, and for all β in $\prod_{i=0}^{p+k} B_i$ not in the image of $\bar{\Psi}$, $(1 - \Pi_P)(M(P[F])[\beta])$ contains ∞.*

Proof. It is sufficient to prove it for the simplest chunk P containing only one command `Xi = f(X1, ..., Xn)`. This comes from the compositional nature of the analysis, as a sequence of commands is assigned the product of the matrices of each individual command. Then, checking the theorem in this case is a straightforward, though tedious (due to keeping track of all indices), computation. ◀

4.2 Integrating recursive calls, the easy way

The question of dealing with self-referential, or recursive, calls, naturally arises when extending to function calls. It turns out that our approach makes such cases easy to handle.

A program implementing a function `rec` calling itself cannot use the F rule presented above as is, since the result of the analysis of `rec` is precisely what we are trying to establish. However, if `rec` takes two input variables `X1` and `X2` and its return value is assigned to a third variable `X3`, then we already know that the vector at 3 will need to be replaced by the vector capturing the dependency between `X1`, `X2`, and the return variable of `rec` (which we will take to be `X3` in our example). The solution consists in replacing the actual values in this vector by variables α, β ranging over values in $\textsc{mwp}^{\infty}$, terminating the analysis with those variables, and then to resolve the equation – which is easy given the small size of the $\textsc{mwp}^{\infty}$ semiring.

As an example[6], consider the following program and compute the corresponding matrix:

```
int rec(X1, X2){
  X1 = X1 + X2;
  X3 = rec(X1, X2);
  return X3;
}
```

$$\begin{pmatrix} m\delta(0,0)\oplus p\delta(1,0)\oplus w\delta(2,0) & 0 & 0 \\ p\delta(0,0)\oplus m\delta(1,0)\oplus w\delta(2,0) & m & 0 \\ 0 & 0 & m \end{pmatrix} \otimes 1 \xleftarrow{3} \begin{pmatrix} \alpha \\ \beta \\ 0 \end{pmatrix}$$
$$= \begin{pmatrix} m\delta(0,0)\oplus p\delta(1,0)\oplus w\delta(2,0) & 0 & \alpha m\delta(0,0)\oplus\alpha p\delta(1,0)\oplus\alpha w\delta(2,0) \\ p\delta(0,0)\oplus m\delta(1,0)\oplus w\delta(2,0) & m & \alpha p\delta(0,0)\oplus\alpha m\delta(1,0)\oplus\alpha w\delta(2,0)\oplus\beta \\ 0 & 0 & 0 \end{pmatrix}$$

Using the assignments 0, 1 and 2 gives $\begin{pmatrix} m & 0 & \alpha m \\ p & m & \alpha p\oplus\beta \\ 0 & 0 & 0 \end{pmatrix}$, $\begin{pmatrix} p & 0 & \alpha p \\ m & m & \alpha m\oplus\beta \\ 0 & 0 & 0 \end{pmatrix}$ and $\begin{pmatrix} w & 0 & \alpha w \\ w & m & \alpha w\oplus\beta \\ 0 & 0 & 0 \end{pmatrix}$, and since the third vector should be equal to $\begin{pmatrix} \alpha \\ \beta \\ 0 \end{pmatrix}$, this gives three systems of equations:

[6] Where we use variables that are not parameters, following footnote 1, and where our recursive call does not terminate: we are focusing on growth rates and not on termination, and keep the example compact.

$$\begin{cases} \alpha m &= \alpha \\ \alpha p \oplus \beta &= \beta \end{cases} \qquad \begin{cases} \alpha p &= \alpha \\ \alpha m \oplus \beta &= \beta \end{cases} \qquad \begin{cases} \alpha w &= \alpha \\ \alpha w \oplus \beta &= \beta \end{cases}$$

The smaller solution to the first (resp. second, third) equational system is $\{\alpha = m; \beta = p\}$ (resp. $\{\alpha = p; \beta = p\}$, $\{\alpha = w; \beta = w\}$), and as a consequence, we find two meaningful solutions (all others being larger than those): $\begin{pmatrix} m \\ p \\ 0 \end{pmatrix}$ and $\begin{pmatrix} w \\ w \\ 0 \end{pmatrix}$.

4.3 Taking advantage of polynomial structure to compute efficiently

Ensuring that the analysis is tractable is an important part of our contribution. For a program accepting n different derivations and having k different derivations that cannot be completed, the original flow calculus must run at most $k + 1$ times to find *one* derivation, while our analysis outputs the $k + n$ different derivations in one run, and then sorts them – as discussed next – by listing all the evaluations and looking for ∞ values. In this task, the C rule, that lets building programs from commands, is obviously crucial and consists simply in multiplying two matrices: however, since we are internalizing the choices, those matrices contain a mixture of functions from choices to coefficients in $\textsc{mwp}^\infty$ and of coefficients in $\textsc{mwp}$. Multiplying such matrices is more costly, but also essential: an 8-line program such as `explosion.c` requires to multiply elements of its matrix 34,992 times[7]. This forces to represent and manipulate the elements of $\prod_{i=1}^{p} A_i \to \mathbb{M}(\textsc{mwp})$ – setting aside ∞ coefficients for a moment – cleverly: simple comparison showed that the improved algorithm presented below made the analysis roughly *five times* faster (Sect. C.3).

As discussed in Notation 5, elements of this semi-ring are represented as *polynomials* w.r.t. the generating set given by the functions $\delta(i,j) : \prod_{i=1}^{p} A_i \to \textsc{mwp}$ defined by $\delta(i,j)(a_1, \dots, a_p) = m$ if $a_j = i$ and $\delta(i,j)(a_1, \dots, a_p) = 0$ otherwise, i.e., an element of $\prod_{i=1}^{p} A_i \to \textsc{mwp}$ is represented as a polynomial $\sum_{i=1}^{n} \alpha_i \prod_{j=1}^{k_i} \delta(a_{i,j}, b_{i,j})$ with $\alpha_i \in \textsc{mwp}$.

This basis has an important property: the *monomials* $\alpha_i \prod_{j=1}^{k_i} \delta(a_{i,j}, b_{i,j})$ in a polynomial can be ordered so that the product with another monomial is ordered, i.e., if $\alpha \leqslant \beta$ and both $\alpha \times \gamma$ and $\beta \times \gamma$ are non-zero, then $\alpha \times \gamma \leqslant \beta \times \gamma$. This order is leveraged to obtain efficient algorithms, similar to what is done using Gröbner bases for computation of standard polynomials [35]. For instance, the algorithm for multiplication of polynomials uses this property to compute the product of an ordered polynomial P with $\sum_{i=1}^{n} \alpha_i \prod_{j=1}^{k_i} \delta(a_{i,j}, b_{i,j})$:

1. compute the products $P_i = P \times \alpha_i \prod_{j=1}^{k_i} \delta(a_{i,j}, b_{i,j})$ for all i;
2. compare and order a list L of all the first elements of those polynomials;
3. append the smallest element to the result and remove it from the corresponding P_i;
4. insert the (new) first element of P_i to the list L if it exists;
5. if L is non-empty, go back to step 3.

When adding or multiplying polynomials, which consist of monomials, we check if a monomial is contained or included by another, and exclude all redundant cases (cf. `contains` or `includes`). This is also done when inserting monomials. Thus we keep polynomials free of implementation choices that we would otherwise have to handle during evaluation.

[7] The need to optimize functions is made even more obvious when we discuss benchmarking in Sect. 5.1.

4.4 Deciding the existence of a bound faster thanks to delta graphs 17

Adopting the $\prod_{i=1}^{p} A_i \to \textsc{mwp}^{\infty}$ semi-ring permits to complete all derivations simultaneously, but remains to determine if there exists an assignment $\vec{a} \in \prod_{i=1}^{p} A_i$ s.t. the resulting matrix is ∞-free, to decide whenever a program accepts a polynomial bound: this is the *evaluation* step. Despite the optimizations detailed above that simplifies the task, this phase remains particularly costly, since the number of assignment grows exponentially w.r.t. the number of choice, which is linear in the number of variables. While this step is necessary (in one form or another) if one wishes to produce the actual mwp matrices certifying polynomial bounds, we implemented a specific data structure to keep track of assignments resulting in ∞ coefficients on the fly, thus allowing the analysis to provide a qualitative answer quickly. This section details how those *delta graphs* allow to immediately determines whenever a polynomial bound exists without having to compute the corresponding matrix, something that was not possible in the original, non-deterministic, calculus.

A delta graph is a graph whose vertices are monomials. The graph is populated during the analysis by adding those monomials that appear with an infinite coefficient – i.e., possible choices leading to ∞ in the resulting matrix. This graph is structured in layers: each layer corresponds to the size of the monomials (the number of deltas) it contains. The intuition is that a monomial – or rather a list of deltas $\delta(\_,\_)$ – defines a subset of the space $\prod_{i=1}^{p} A_i$; the less deltas in the monomial, the greater the subspace represented[8]. As we populate the delta graph, we create edges within a given layer to keep track of differences between monomials: we add an edge labeled i between two monomials if and only if they differ only on one delta $\delta(\_, i)$ (i.e., one is obtained from the other by replacing the first index of $\delta(\_, i)$). This is used to implement a `fusion` method on delta graphs, which simplifies the structure: as soon as a monomial m in layer n has $\mathrm{Card}(A_i) - 1$ outgoing edges labelled i, we can remove all these monomials and insert a shorter monomial in layer $n-1$, obtained from m by simply removing $\delta(\_, i)$. This implements the fact that $\sum_{k=0}^{\mathrm{Card}(A_i)-1} m\delta(k, j) = m$.

Now, remember the delta graph represents the subspace of assignments for which an ∞ appears. If at some point the delta graph is completely simplified (i.e., "fusions" to the graph with a unique monomial consisting in an empty list of $\delta(\_,\_)$), it means the whole space of assignments is represented and no mwp-bounds can be found. On the contrary, if the analysis ends with a delta graph different from the completely simplified one, at least one assignment exists for which no infinite coefficients appear, and therefore at least one mwp-bound exists. This allows one to answer the question "Is there at least one mwp-bound?" *without actually computing said bounds*. Based on the information collected in the delta graph and the matrix with polynomial coefficients, one can however recover all possible matrix assignments by going through all possible valuations.

This last part is implemented with a specific iterator that leverages the information collected in the delta graph to skip large sets of valuations in a single step. For instance, suppose the monomial $\delta(1, 1)$ lies in the delta graph – i.e., that an infinite coefficient will be reached if the second index is equal to 1. When asked the valuation after $(0, 0, 2, 2)$ (and supposing that $\mathrm{Card}(A_i) = 3$ for all i), our `delta_iterator` will jump directly to $(0, 2, 0, 0)$, skipping all intermediate valuation of the form $(0, 1, a, b)$ in a single step. Similarly, it will jump from $(1, 0, 2, 2)$ to $(1, 2, 0, 0)$, again skipping several valuations at a time, providing a

[8] Our intuitions here come from the standard topological structure of spaces of infinite sequences, where such a monomial represents a "cylinder set", i.e., an element of the standard basis for open sets.

18 faster analysis. Note that the implementation required care, to correctly jump when given additional informations from the delta graph, e.g., to produce $(2, 0, 1, 0)$ as the successor of $(0, 0, 2, 2)$ if $\delta(0, 0)$, $\delta(1, 1)$ and $\delta(0, 2)$ all belong to the delta graph.

5 Implementing, testing and comparing the analysis

Demonstrating the implementability of the improved and extended mwp-bounds analysis requires an implementation. Our open-source solution, packaged through Python Package Index (PyPI) as `pymwp`, is a standalone command line tool, written in `Python`, that automatically performs growth-rate analysis on programs written in a subset of the `C` programming language. For programs that pass the analysis, it produces a matrix corresponding to the input program and a list of valid derivation choices; and for programs that do not have polynomial bounds, it reports infinity. Our motivation for choosing `C` as the language of analysis resulted from its central role and similarity with the original `while` language. `Python` was an ideal choice for the implementation because of its plasticity, collection of libraries, and because it allowed partial reuse of a previous flow analysis tool [3, 31, 32]. The source code is available on Github, along with an online demo, and detailed documentation [33] describing its current supported features and functionality. We now discuss how we tested and assesed it, and how it compares (or, rather *does not* compare) to other similar approaches.

5.1 Experimental evaluation

We allocated extensive focus and effort on testing and profiling our implementation, to ensure the correctness and efficiency of the analysis, and with the terminal objective of obtaining a usable tool. The test suite includes 42 `C` programs, carefully designed to exercise different aspects of the analysis, ranging from basic derivations, to ones producing worst-case behavior (by yielding e.g., dense matrices or exponential number of derivations), and classical examples such as computing the greatest common divisor or exponentiation.

We refer to our benchmarks (presented in Appendix C) for measured analysis results for each program. The most salient aspect is that our analysis is extremely fast (the time is measured in *milli*seconds) despite important numbers of function calls (in the 10k range, excluding builtin Python language calls, for 10-lines programs). Even examples tailored to stress our implementation cannot make the analysis go over *4 seconds*. We cannot compare our implementation with implementations of the original analysis, since it has never been implemented, and (according to our attempts) cannot be implemented in any realistic manner.

5.2 Related tools and incompatible metrics

This work was inspired by the series of works of the flow analysis from the "Copenhagen school" [11, 24]. The overall flow analysis approach is related in spirit to abstract interpretation [13, 14]; that bounds *transitions* between states (e.g., commands) instead of states [24]. This approach shaped the implementation of tools detecting loop quasi-invariants [31, 32].

Other communities share a similar goal of inferring resource-usage. Complexity analyzers such as SPEED [19] for `C++`, COSTA [1] for `Java` bytecode, ComplexityParser [21] for `Java`, Resource Aware ML for `OCaml` [29] or Cerco [2] and Verasco [25] for `C` generate (certified) cost or runtime analysis on (subsets of) imperative programming languages. Embracing such a large diversity is difficult, but our technique is different from existing implementations and tools: most of them focus on worst-case resource-usage complexity or termination, while we

19 are interested in upper-bounds on the final values of program variables, i.e., we focus on *growth* instead of actual values. This makes the comparison with our approach difficult, but highlights at the same time its uniqueness in today's landscape of static analyzers.

Further, our approach provides other desirable properties:

1. it is compositional, which allows one to "hot-plug" bounds of previously analyzed functions without additional work,
2. it is modular, as the internal machinery can be altered – as in this paper – without having to re-develop the theory,
3. it is language-independent, as it reasons abstractly on imperative languages, but can be applied to real programs, as our implementation illustrates, and should extend to more complex languages,
4. it is lightweight and programmer-friendly, as it is fast, does not require annotations or to record value ranges,
5. it studies growth independently from e.g., iteration bounds, thus sidestepping difficult cases that worst-case analysis has to tackle, and
6. it may enable tight bounds on programs, as it has been done recently [10] for a similar analysis [11].

In particular compositionality is a highly desirable property – because otherwise the analysis needs to be re-run on programs or API whenever embedded into different pieces of software – yet difficult to achieve by most other approaches, as discussed and partially remedied recently [12]. While we suppose one approach could be used to derive the result obtained by the other, we do believe the originality of our pioneering ICC-based approach may inspire new and original directions in static program analysis.

6 Conclusion: limitations, strengths and future work

This work attempts to illustrate the usefulness and applicability of ICC results, but also the need to refine and adapt them. We showed that the mwp-flow analysis as originally described cannot scale to programs in a real programming language: while the considered analysis is definitely powerful and elegant, its mathematical nature let some costly operations go unchecked. However we have shown that, extended and coupled to optimizations techniques, its result enable the development of a novel and original static analysis technique on imperative programs, focused on *growth* rather than on termination or worst-case bounds.

This work is a proof of concept and it has limitations, both theoretical and practical: the theory is missing memory uses, pointers, and arrays and the supported feature set of the implementation could be extended. But instead of focusing on what this analysis *cannot* perform, we would like to stress that all the tools are in place to perform similar analysis on intermediate representations of code in compilers, which will naturally simplify the task of fitting richer program syntax to our analysis, and brings this technique yet another step closer to practical use cases.

One of our next steps include certifying the analysis using the Coq proof assistant [34], and implementing the analysis in certified tools such as the Compcert compiler [28] (or, more precisely, its static single assignment version [7]) or certified-llvm [36]. The plasticity of both compilers and of the implemented analysis should facilitate porting our results and approaches to support further programming languages in addition to `C`. As complexity analysis is notably difficult in Coq [20], we believe a push in this direction would be welcome, and that ICC provides all the needed tools for it.

Another direction is to explore the possibility for our analysis to focus on the *final* values of variables instead of tracking them throughout the whole program. Indeed, recall from Sect. 3.2 that our semi-ring is such that $\infty \times^{\infty} 0 = \infty$, but another valid choice would have been to pick $\infty \times^{\infty} 0 = 0$ [4, A.4]. In this case, it seems that if some non-polynomial growth is caused by a variable that is then “thrown away” (overridden), then a program could still pass the analysis: whether this lead gives relevant results is yet to determine, but it would be another nice illustration of the plasticity of this analysis.

Last but not least, working on finer comparison with other static analyzers [18] could be useful. We have stressed in Sect. 5.2 how such comparison was uneasy, as the finality of our tool is not directly comparable with any other analyzer we know of. However, some tools such as AProVE [17] or CoFloCo [16] provide polynomial upper bounds for `C` programs, and we could assess e.g., on the Termination Problems Data Base whether `pymwp` can analyze as many problems and how often all three analyzers agree. The motivation for the original mwp-analysis was to develop resource analysis for distinguishing feasible problems and to work at the boundary of undecidability, but this could actually be one of `pymwp`’s strength as a *pre-processor* to other static analyzers, to save them from running costly analysis on programs known to be unfeasible. This fast analysis and the compositionality of our tool could also, on longer term, be useful to construct IDE plug-ins that provide low-latency feedback to the programmer.

References

1 Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. COSTA: design and implementation of a cost and termination analyzer for java bytecode. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures*, volume 5382 of *LNCS*, pages 113–132. Springer, 2007. `doi:10.1007/978-3-540-92188-2_5`.

2 Roberto M. Amadio, Nicholas Ayache, François Bobot, Jaap Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, Dominic P. Mulligan, Mauro Piccolo, Randy Pollack, Yann Régis-Gianas, Claudio Sacerdoti Coen, Ian Stark, and Paolo Tranquilli. Certified complexity (cerco). In Ugo Dal Lago and Ricardo Peña, editors, *Foundational and Practical Aspects of Resource Analysis - Third International Workshop, FOPARA 2013, Bertinoro, Italy, August 29-31, 2013, Revised Selected Papers*, volume 8552 of *LNCS*, pages 1–18. Springer, 2013. `doi:10.1007/978-3-319-12466-7_1`.

3 Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. Lqicm on c toy parser. URL: `https://github.com/statycc/LQICM_On_C_Toy_Parser`.

4 Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. mwp-analysis improvement and implementation: Realizing implicit computational complexity. Preliminary technical report, March 2022. URL: `https://hal.archives-ouvertes.fr/hal-03596285`.

5 Martin Avanzini and Ugo Dal Lago. Automating sized-type inference for complexity analysis. *Proc. ACM Program. Lang.*, 1(ICFP):43:1–43:29, 2017. `doi:10.1145/3110287`.

6 Patrick Baillot and Kazushige Terui. Light types for polynomial time computation in lambda-calculus. In *LICS*, pages 266–275. IEEE Computer Society, 2004. `doi:10.1109/LICS.2004.1319621`.

7 Gilles Barthe, Delphine Demange, and David Pichardie. Formal verification of an SSA-based middle-end for compcert. *ACM Trans. Program. Lang. Syst.*, 36(1):4:1–4:35, 2014. `doi:10.1145/2579080`.

8 Stephen J. Bellantoni and Stephen Arthur Cook. A new recursion-theoretic characterization of the polytime functions (extended abstract). In S. Rao Kosaraju, Mike Fellows, Avi Wigderson, and John A. Ellis, editors, *STOC*, pages 283–93. ACM, 1992. `doi:10.1145/129712.129740`.

21 **9** Amir M. Ben-Amram. On decidable growth-rate properties of imperative programs. In Patrick Baillot, editor, *Proceedings International Workshop on Developments in Implicit Computational complExity, DICE 2010, Paphos, Cyprus, 27-28th March 2010*, volume 23 of *EPTCS*, pages 1–14, 2010. `doi:10.4204/EPTCS.23.1`.

10 Amir M. Ben-Amram and Geoff W. Hamilton. Tight polynomial worst-case bounds for loop programs. *Log. Meth. Comput. Sci.*, 16(2), 2020. `doi:10.23638/LMCS-16(2:4)2020`.

11 Amir M. Ben-Amram, Neil D. Jones, and Lars Kristiansen. Linear, polynomial or exponential? complexity inference in polynomial time. In Arnold Beckmann and Costas Dimitracopoulos andBenedikt Löwe, editors, *Logic and Theory of Algorithms, 4th Conference on Computability inEurope, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings*, volume 5028 of *LNCS*, pages 67–76. Springer, 2008. `doi:10.1007/978-3-540-69407-6_7`.

12 Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 467–478. ACM, 2015. `doi:10.1145/2737924.2737955`.

13 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977. `doi:10.1145/512950.512973`.

14 Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of recursive procedures. In Erich J. Neuhold, editor, *Formal Description of Programming Concepts: Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts, St. Andrews, NB, Canada, August 1-5, 1977*, pages 237–278. North-Holland, 1977.

15 Ugo Dal Lago. A short introduction to implicit computational complexity. In Nick Bezhanishvili and Valentin Goranko, editors, *ESSLLI*, volume 7388 of *LNCS*, pages 89–109. Springer, 2011. `doi:10.1007/978-3-642-31485-8_3`.

16 Antonio Flores-Montoya. Upper and lower amortized cost bounds of programs expressed as cost relations. In John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, volume 9995 of *LNCS*, pages 254–273, 2016. `doi:10.1007/978-3-319-48989-6_16`.

17 Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Jera Fuhs, Carstenand Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, and René Swiderski, Stephanie andThiemann. Analyzing program termination and complexity automatically with aprove. *J. Autom. Reasoning*, 58(1):3–31, 2017. `doi:10.1007/s10817-016-9388-y`.

18 Jürgen Giesl, Albert Rubio, Christian Sternagel, Johannes Waldmann, and Akihisa Yamada. The termination and complexity competition. In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*, volume 11429 of *LNCS*, pages 156–166. Springer, 2019. `doi:10.1007/978-3-030-17502-3_10`.

19 Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. Speed: Precise and efficient static estimation of program computational complexity. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 127–139, New York, NY, USA, 2009. Association for Computing Machinery. `doi:10.1145/1480881.1480898`.

20 Armaël Guéneau. *Mechanized Verification of the Correctness and Asymptotic Complexity of Programs. (Vérification mécanisée de la correction et complexité asymptotique de programmes)*. PhD thesis, Inria, Paris, France, 2019. URL: `https://tel.archives-ouvertes.fr/tel-02437532`.

21 Emmanuel Hainry, Emmanuel Jeandel, Romain Péchoux, and Olivier Zeyen. Complexityparser: An automatic tool for certifying poly-time complexity of Java programs. In Antonio Cerone and

22 Peter Csaba Ölveczky, editors, *Theoretical Aspects of Computing - ICTAC 2021 - 18th International Colloquium, Virtual Event, Nur-Sultan, Kazakhstan, September 8-10, 2021, Proceedings*, volume 12819 of *LNCS*, pages 357–365. Springer, 2021. doi:10.1007/978-3-030-85315-0_20.

22 Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ML. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *LNCS*, pages 781–786. Springer, 2012. doi:10.1007/978-3-642-31424-7_64.

23 Neil D. Jones and Lars Kristiansen. A flow calculus of *mwp*-bounds for complexity analysis. *ACM Trans. Comput. Log.*, 10(4):28:1–28:41, 2009. doi:10.1145/1555746.1555752.

24 Neil D. Jones and Flemming Nielson. Abstract interpretation: A semantics-based tool for program analysis. In Samson Abramsky, Dov M. Gabbay, and Thomas Stephen Edward Maibaum, editors, *Semantic Modelling*, volume 4 of *Handbook of Logic in Computer Science*, pages 527–636. Oxford University Press, 1995.

25 Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium onPrinciples of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 247–259. ACM, 2015. doi:10.1145/2676726.2676966.

26 Yves Lafont. Soft linear logic and polynomial time. *Theor. Comput. Sci.*, 318(1):163–180, 2004. doi:10.1016/j.tcs.2003.10.018.

27 Daniel Leivant. Stratified functional programs and computational complexity. In Mary S. Van Deusen and Bernard Lang, editors, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 325–333. ACM Press, 1993. doi:10.1145/158511.158659.

28 Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. doi:10.1145/1538788.1538814.

29 Benjamin Lichtman and Jan Hoffmann. Arrays and references in resource aware ML. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK*, volume 84 of *LIPIcs*, pages 26:1–26:20. Schloss Dagstuhl, 2017. doi:10.4230/LIPIcs.FSCD.2017.26.

30 Jean-Yves Moyen. *Implicit Complexity in Theory and Practice.* Habilitation thesis, University of Copenhagen, 2017. URL: https://lipn.univ-paris13.fr/~moyen/papiers/Habilitation_JY_Moyen.pdf.

31 Jean-Yves Moyen, Thomas Rubiano, and Thomas Seiller. Loop quasi-invariant chunk detection. In Deepak D'Souza and K. Narayan Kumar, editors, *ATVA*, volume 10482 of *LNCS*. Springer, 2017. doi:10.1007/978-3-319-68167-2_7.

32 Jean-Yves Moyen, Thomas Rubiano, and Thomas Seiller. Loop quasi-invariant chunk motion by peeling with statement composition. In Guillaume Bonfante and Georg Moser, editors, *Proceedings 8th Workshop on Developments in Implicit Computational Complexity and 5th Workshop on Foundational and Practical Aspects of Resource Analysis, DICE-FOPARA@ETAPS 2017, Uppsala, Sweden, April 22-23, 2017*, volume 248 of *EPTCS*, pages 47–59, 2017. doi:10.4204/EPTCS.248.9.

33 pymwp's documentation, 2021. URL: https://statycc.github.io/pymwp/.

34 Coq Team. Coq documentation, 2022. URL: https://coq.github.io/doc/.

35 Joris van der Hoeven and Robin Larrieu. Fast Gröbner basis computation and polynomial reduction for generic bivariate ideals. *Applicable Algebra in Engineering, Communication and Computing*, 30(6):509–539, December 2019. doi:10.1007/s00200-019-00389-9.

36 Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formal verification of SSA-based optimizations for LLVM. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 175–186. ACM, 2013. doi:10.1145/2491956.2462164.

A Technical appendix on semi-rings (abridged)

This is an abridged version of the technical development on semi-ring that is exposed in full details in our technical report [4, A.1 and A.2].

▶ **Lemma 18** (mwp semi-ring). *The tuple* $(\{0, m, w, p\}, 0, m, +, \times)$*, with*

- $0 < m < w < p$,
- $\alpha + \beta = \begin{cases} \alpha & \text{if } \alpha \geqslant \beta \\ \beta & \text{otherwise} \end{cases}$
- $\alpha \times \beta = \begin{cases} \alpha + \beta & \text{if } \alpha \neq 0 \text{ and } \beta \neq 0 \\ 0 & \text{otherwise} \end{cases}$

is a strong *semi-ring.*

▶ **Lemma 19** (Matrix semi-ring). *Given a strong semi-ring* $\mathbb{S} = (S, 0, 1, +, \times)$*, the tuple* $\mathbb{M} = (M, 0, 1, \oplus, \otimes)$*, with*

- M *the set of all* $n \times n$ *matrices over* S*, for all* $n \in \mathbb{N}$*,*
- 0 *defined by* $M = 0$ *iff* $M_{ij} = 0$ *for all* i *and* j*,*
- 1 *defined by* $M = 1$ *iff* $M_{ij} = 1$ *for* $i = j$*,* $M_{ij} = 0$ *otherwise,*
- $\oplus$ *defined by* $C = A \oplus B$ *iff* $C_{ij} = A_{ij} + B_{ij}$*,*
- $\otimes$ *defined by* $C = A \otimes B$ *iff* $C_{ij} = \sum_{k=1}^{n} A_{ik} \times B_{kj}$*,*

is a strong semi-ring.

For simplicity, we will write $\mathbb{M}$ as $\mathbb{M}(\mathbb{S}) = (M(S), 0, 1, \oplus, \otimes)$.

▶ **Lemma 20** (Choices semi-ring). *Given a strong semi-ring* $\mathbb{S} = (S, 0, 1, +, \times)$ *and a set* A*, the tuple* $\mathbb{F} = (F, 0, 1, \boxplus, \boxtimes)$*, with*

- F *the set of functions from* A *to* S*,*
- 0 *the constant function* $0(a) = 0$ *for all* $a \in A$*,*
- 1 *the constant function* $1(a) = 1$ *for all* $a \in A$*,*
- $\boxplus$ *defined componentwise:* $(f \boxplus g)(a) = (f(a)) + (g(a))$*, for all* f*,* g *in* F *and* $a \in A$*,*
- $\boxtimes$ *defined componentwise:* $(f \boxtimes g)(a) = (f(a)) \times (g(a))$*, for all* f*,* g *in* F *and* $a \in A$*,*

is a strong semi-ring.

For simplicity, we will write $\mathbb{F}$ as $A \to \mathbb{S} = (A \to S, 0, 1, +, \times)$.

▶ **Lemma 21.** *For all set* A *and strong semi-ring* $\mathbb{S}$*,* $\mathbb{M}(A \to \mathbb{S}) \cong A \to \mathbb{M}(\mathbb{S})$*.*

▶ **Lemma 22.** *Given a strong semi-ring* $\mathbb{S} = (S, 0, 1, +, \times)$ *and an element* $\bot \notin S$*,* $\mathbb{S}^{\bot} = (S \cup \{\bot\}, 0, 1, +^{\bot}, \times^{\bot})$ *with, for all* a*,* $b \in S \cup \{\bot\}$*,*

$$a +^{\bot} b = \begin{cases} a + b & \text{if } a, b \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

$$a \times^{\bot} b = \begin{cases} a \times b & \text{if } a, b \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

is a semi-ring.

Proof. The proof is immediate, but note that $\mathbb{S}^{\bot}$ is not strong, as $\bot \times 0 = \bot$. ◀

B Omitted Proofs

▶ **Theorem 10** (Determinancy and termination). *Given a program P, there exists unique $p \in \mathbb{N}$ and $M \in \mathbb{M}(\{0,1,2\}^p \to \text{MWP}^\infty)$ such that $\vdash P : M$.*

Proof. The proof proceeds by induction on the length of the program P, expressed in number of commands. We let p be the number of variables in P, but observe that any program P can be treated as manipulating $p' > p$ different variables, by simply adding $p' - p$ additional rows and columns to the matrix, and leaving them unchanged by the derivation of P. While a complete proof would need to constantly account for the number of actual and potential variables used by P, we will simply assume that the reader understands that accounting for this technicality obfuscate more than it clarifies the proof, and we will freely resize the matrices to account for additional variables when needed.

If P is of length 1 Then we know P is of the form `X = e`, and only the rule A can be applied. But then we need to prove that all expression `e` can be typed with exactly one vector. An expression `e` is either a variable `X`, or a composed expression `X * Y`, `X - Y`, or `X + Y`. But then, respectively, only E^{S}, E^{M} or E^{A} (for addition and substraction) can be applied, and this case is proven.

If P is of length $n > 1$ Then we proceed by case on the structure of the command:

- If P is of the form `if b then P1  else P2`, then by induction we know for $i \in \{1,2\}$ there exists p_i and M_i of size $p_i \times p_i$ such that $\vdash$ `Pi` $: M_i$. If $p_1 \neq p_2$, then letting `Mj` being the smaller matrix, it is easy to rewrite `Pj`'s derivation to account for $|p_1 - p_2|$ additional variables, and as $\oplus$ is uniquely defined, we know that $M_1 \oplus M_2$ results in a unique matrix of size $\max(p_1, p_2)$.
- If P is of the form `while b do P'`, this is immediate by induction hypothesis on `P'`, considering that only W^∞ can be applied, and that this rule produces a unique matrix.
- If P is of the form `loop X {P'}`, this case is similar to the previous one, using L^∞ instead of W^∞.
- If P is of the form `P1;P2`, this case is similar to the `if` case, with the possible need to resize one of the matrix obtained by induction, and using that $\otimes$ is uniquely defined. ◀

▶ **Theorem 11** (Adequacy). *If $\vdash P : M$, then for all $\vec{a} \in A^p$, $\vdash_{\text{JK}} P : M[\vec{a}]$ iff $\infty \notin M[\vec{a}]$.*

Proof. The proof proceeds by induction on the length of the program P, expressed in number of commands.

If P is of length 1 Then we know P is of the form `X = e`, and only the rule A can be applied, in both systems. Hence, we need to prove that all expression `e` can be typed the same way in both systems. A careful comparison of Figures 1 and 2 shows that if `e` is of the form `Xi`, then there is a small mismatch. In the original system, we can use either E2, and obtain $\vdash_{\text{JK}}$ `Xi` $: \{{}^{w}_{\mathtt{i}}\}$, or E1, and obtain $\vdash_{\text{JK}}$ `Xi` $: \{{}^{m}_{\mathtt{i}}\}$, while the only derivation in the deterministic system is using E^{S} to get $\vdash_{\text{JK}}$ `Xi` $: \{{}^{m}_{\mathtt{i}}\}$. As $m < w$, we argue that the deterministic system cannot obtain a derivation that is not useful anyway, and hence that it can be ignored.

As for the other cases, if `e` is a composed expression `X * Y`, `X - Y`, or `X + Y`, it is easy to observe that E^{A} and E^{M} encapsulates all the possible combinations of E2 and of E1 followed by E3 or E4 that can be used.

If P is of length $n > 1$ Then the result holds by induction, once we observed that L^∞ and W^∞ are introducing ∞ coefficients *only if* L and W cannot be applied. ◀

C Benchmarks 25

C.1 Descriptions of program groups

- *Basics* – C programs performing operations corresponding to simple derivation trees.
- *Implementation paper* – example programs presented in this paper.
- *Original paper* – examples taken from or inspired by the original analysis [23].
- *Infinite* – programs whose matrices always contain infinite coefficients.
- *Polynomial* – programs whose matrices do not always contain infinite coefficients.
- *Other* – other C programs of interest.

C.2 Results

The benchmarks are categorized and grouped to distinguish the type of system behavior they exercise. For each program we capture in Table 1

1. program variable count
2. the lines of code in the source program (LOC column)
3. clock time taken by the full analysis (excluding saving result to file, which is otherwise default behavior),
4. number of function calls excluding builtin Python language calls, and
5. the result of the analysis.

Collectively the LOC, time, and function calls columns provide insight into the behavior of the analysis as different aspects of the system are being stress-tested. From the results column we report expected results on each benchmarked program. In the benchmarks table a passing result is represented with ✓and ∞ otherwise. We do not report manually computed bounds as comparison, because the analysis is carried out on individual variables, thus calculating them on multivariate programs is tedious and futile. However, for simple programs such as while_2.c, it is straightforward through visual inspection to verify the obtained 2×2-matrix is indeed the correct result.

These benchmarks were obtained using Python's built-in cProfile utility, extended in `pymwp` implementation to enable batch profiling. The clock times are slight overestimates because the utility adds minor runtime overhead. The number of function calls includes primitive calls, but exclude built-in Python language calls. Full detailed results are viewable in the source code repository: https://github.com/statycc/pymwp/releases/tag/profile-latest

C.3 Comparison

It is not really meaningful or possible to compare those results with any other static analyzer, and impossible to compare it with any other implementation of this type of flow analysis. While we could, in theory, analyze our examples with other static analyzers, their results would be incomparable, as they would produce guarantees on termination or worst case resource usage, which are both orthogonal to our polynomial bounds on value growth. To our knowledge, the only static analyzer using similar metrics [5] was developed only for functional languages, thus preventing comparison. As for implementations of the original analysis, our first attempts showed that a naive implementation would likely fail to handle the memory or time explosions. We did, however, compare the gains resulting from the optimizations described in Sect. 4.3. In a nutshell, our improved algorithm for adding and multiplying polynomials resulted in the analysis being roughly *five times faster* for two programs that we estimate to be representative.

Table 1 Benchmark results produced by `pymwp` on `C` programs.

Program name	Variables	LOC	Time (ms)	Function calls	Bound
Basics					
assign_expression	2	8	133	81614	✓
assign_variable	2	9	115	81238	✓
if	2	9	118	82046	✓
if_else	2	7	118	82928	✓
inline_variable	2	9	118	81979	✓
while_1	2	7	117	82934	✓
while_2	2	7	117	83964	✓
while_if	3	9	122	91572	✓
Implementation paper					
example7	3	10	122	86898	✓
example15_a	2+2	25	122	88763	✓
example15_b	4	16	137	122016	✓
Original paper					
example3_1_a	3	10	110	85286	✓
example3_1_b	3	10	120	87637	✓
example3_1_c	3	11	121	89173	✓
example3_1_d	2	12	116	80002	∞
example3_2	3	12	118	83182	∞
example3_4	5	18	134	108890	∞
example5_1	2	10	116	81185	✓
example7_10	3	10	119	86053	✓
example7_11	4	11	139	119379	✓
Infinite					
exponent_1	4	16	127	99893	∞
exponent_2	4	13	123	92846	∞
infinite_2	2	6	143	128275	∞
infinite_3	3	9	120	89880	∞
infinite_4	5	9	3274	5924420	∞
infinite_5	5	11	369	529231	∞
infinite_6	4	14	1624	2836726	∞
infinite_7	5	15	631	964189	∞
infinite_8	6	23	880	1444782	∞
Polynomial					
notinfinite_2	2	4	119	86174	✓
notinfinite_3	4	9	131	104826	✓
notinfinite_4	5	11	169	168242	✓
notinfinite_5	4	11	174	176179	✓
notinfinite_6	4	16	195	215765	✓
notinfinite_7	5	15	1161	1961806	✓
notinfinite_8	6	22	1893	3172293	✓
Other					
dense	3	16	157	151428	✓
dense_loop	3	17	269	353068	✓
explosion	18	23	1296	2327071	✓
gcd	2	12	114	84914	∞
simplified_dense	2	9	118	85098	✓

I.2 Distributing and Parallelizing Non-canonical Loops

Distributing and Parallelizing Non-canonical Loops[⋆]

Clément Aubert[1][0000−0001−6346−3043], Thomas Rubiano[2], Neea Rusch[1][0000−0002−7354−5330], and Thomas Seiller[2,3][0000−0001−6313−0898]

[1] School of Computer and Cyber Sciences, Augusta University
[2] LIPN – UMR 7030 Université Sorbonne Paris Nord
[3] CNRS

Abstract. This work leverages an original dependency analysis to parallelize loops regardless of their form in imperative programs. Our algorithm distributes a loop into multiple parallelizable loops, resulting in gains in execution time comparable to state-of-the-art automatic source-to-source code transformers when both are applicable. Our graph-based algorithm is intuitive, language-agnostic, proven correct, and applicable to all types of loops. Importantly, it can be applied even if the loop iteration space is unknown statically or at compile time, or more generally if the loop is not in canonical form or contains loop-carried dependency. As contributions we deliver the computational technique, proof of its preservation of semantic correctness, and experimental results to quantify the expected performance gains. We also show that many comparable tools cannot distribute the loops we optimize, and that our technique can be seamlessly integrated into compiler passes or other automatic parallelization suites.

Artifacts Available VMCAI

Artifacts Evaluated Functional VMCAI

Keywords: Program Transformation · Automatic Parallelization · Loop Optimization · Abstract Interpretation · Program Analysis · Dependency Analysis

[⋆] This research is supported by the Transatlantic Research Partnership of the Embassy of France in the United States and the FACE Foundation. Th. Rubiano and Th. Seiller are also supported by the Île-de-France region through the DIM RFSI project "CoHOp". N. Rusch is supported in part by the Augusta University Provost's office, and the Translational Research Program of the Department of Medicine, Medical College of Georgia at Augusta University.

1 Original Approaches to Automatic Parallelization

1.1 The Challenge of Unknown Iteration Space

Loop fission (a.k.a. loop distribution) is an optimization technique that breaks loops into multiple loops, with the same condition or index range, each taking only a part of the original loop's body. Such transformation creates opportunity for parallelization and reduces program's running time. For instance, the loop

```
while(t[i] != j){
  s1[i] = j*j;
  s2[i] = 1/j;
  i++;}
```

would become

```
while(t[i1] != j)
 {s1[i1] = j*j; i1++;}
while(t[i2] != j)
 {s2[i2] = 1/j; i2++;}
```

under this transformation. In the transformed program, variable `i` is substituted with two copies, `i1` and `i2`, and we obtain two `while` loops that can be executed in parallel.[4] The gain, in terms of time, results from the fact that the original loop could only be executed sequentially, while the transformed loops can each be assigned to one core. If we consider similarly structured loops that perform resource-intensive computation or that can be distributed in e.g., 8 loops running on 8 cores, it becomes intuitive how this technique can yield measurable performance gain.

This example straightforwardly captures the idea behind loop fission. Of course, as a loop with a short body, it misses the richness and complexities of realistic software. It is therefore very surprising that all the existing loop fission approaches fail at transforming such an elementary program! The challenge comes from the kind of loop presented. Applying loop fission to "canonical" (Def. 15) loops or loops whose number of iterations can be pre-determined is an established convention. But our example of a non-canonical loop with a (potentially) unknown iteration space cannot be handled by those approaches (Sect. 4).

In this paper we present a loop fission technique that can resolve this limitation, because it can be applied to all kinds of a loops.[5] The technique is applicable to any programming language in the imperative paradigm, lightweight and proven correct. The loop fission technique derives these capabilities from a graph-based dependency analysis, first introduced in our previous work [33]. Now we refine this dependency analysis and explain how it can be leveraged to obtain *loop-level parallelism*: a form of parallelism concerned with extracting parallel tasks from loops. We substantiate our claim of running time improvement by benchmarking our technique in Sect. 5. The results show, in cases where iteration space is unknown, that we obtain gain up to the number of parallelizable loops, and that in other cases the speedup is comparable to alternative techniques.

[4] In practice, private copies of `i` are automatically created by e.g., the standard parallel programming API for `C`, OpenMP. Its `pragma` directives are illustrated in Fig. 5

[5] We focus on `while` loops, but other kinds of loops (`for`, `do...while`, `foreach`) can always be translated into `while` and general applicability follows.

1.2 Motivations for Correct, Universal and Automatic Parallelization

The increasing need to discover and introduce parallelization potential in programs fuels the demand for loop fission. To leverage the potential speedup available on modern multicore hardware, all programs—including legacy software—should instruct the hardware to take advantage of its available processors.

Existing parallel programming APIs, such as OpenMP [25], PPL [32], and oneTBB [22], facilitate this progression, but several issues remain. For example, classic algorithms are written sequentially without parallelization in mind and require reformatting to fit the parallel paradigm. Suitable sequential programs with opportunity for parallelization must be modified, often manually, by carefully inserting parallelization directives. The state explosion resulting from parallelization makes it impossible to exhaustively test the code running on parallel architectures [12]. These challenges create demand for *correct* automatic parallelization approaches, to transform large bodies of software to semantically equivalent parallel programs.

Compilers offer an ideal integration point for many program analyses and optimizations. Automatic parallelization is already a standard feature in developing industry compilers, optimizing compilers, and specialty source-to-source compilers. Tools that perform local transformations, generally on loops, are frequently conceived as compiler passes. How those passes are intertwined with sequential code optimizations can however be problematic [14]. As an example, OpenMP directives are by default applied early in the compilation and hence the parallelized source code cannot benefit from sequential optimizations such as unrolling. Furthermore, compilers tend to make conservative choices and miss opportunities to parallelize [14,21].

The loop fission technique presented in this paper offers an incremental improvement in this direction. It enables discovery of parallelization potential in previously uncovered cases. In addition, the flexibility of the system makes it suitable to integration and pipelining with existing parallelization tools at various stages of compilation, as discussed in Sect. 6.

1.3 Our Technique: Properties, Benefits and Limitations

Our technique possesses four notable properties, compared to existing techniques:

Suitable to loops with unknown iteration spaces —our method does not require knowing loop iteration space statically nor at compile time, making it applicable to loops which are often ignored.

Loop-agnostic —our method requires practically no structure from the loops: they can be `while`, `do ... while` or `for` loops, have arbitrarily complex update and termination conditions, loop-carried dependencies, and arbitrarily deep loop nests.

Language-agnostic —our method can be used on any imperative language, and without manual annotations, making it flexible and suitable for application and integration with tools and languages ranging from high-level to intermediate representations.

Correct—our method is easy to prove correct and intuitive, largely because it does not apply to loop bodies with pointers or complex function calls.

All the approaches we know of fail in at least one respect. For instance, polyhedral optimizations cannot transform loops with unknown iteration spaces, since they work on static control parts of programs, where all control flow and memory accesses are known at compile time [20, p. 36]. More importantly, all the "popular" [35] automatic tools fail to optimize `do...while` loops, and require `for` and `while` loops to have canonical forms, that generally require the trip count to be known at compilation time. We discuss these alternative approaches in detail in Sect. 4.

The main limitation of our approach is with function calls and memory accesses. Although we can treat loops with pure function calls, we exclude treatment of loops that contain explicit pointer manipulation, pointer arithmetic or certain function calls. We reserve the introduction of these enhancements as future extensions of our technique. In the meantime, and with these limitations in mind, we believe our approach to be a good complement to existing approaches. Polyhedral models [24]—that are also pushing to remove some restrictions [13]—, advanced dependency analyses, or tools developed for very precise cases (such as loop tiling [14]), should be used in conjunction with our technique, as their use cases diverge (Sect. 6).

1.4 Contributions: From Theory to Benchmarks

We deliver a complete perspective on the design and expected real-time efficiency of our loop fission technique, from its theoretical foundations to concrete measurements. We present three main contributions:

1. The loop fission transformation algorithm—Sect. 3.1—that analyzes dependencies of loop condition and body variables, establishes cliques between statements, and splits independent cliques into multiple loops.
2. The correctness proof—Sect. 3.2—that guarantees the semantic preservation of loop transformation.
3. Experimental results [8]—Sect. 5—that evaluate the potential gain of the proposed technique, including loops with unknown iteration spaces, and demonstrates its integrability with existing parallelization frameworks.

But first, we present and illustrate the dependency analysis that enables our loop fission technique.

2 Background: Language and Dependency Analysis

2.1 A Simple While Imperative Language With Parallel Capacities

We use a simple imperative `while` language, with semantics similar to `C`, extended with a `parallel` command, similar to e.g., OpenMP's directives [25], allowing to execute its arguments in parallel.[6] Our language supports arrays but not pointers, and we let `for` and `do...while` loops be represented using `while` loops. It is easy to map to fragments of `C`, Java, or any other imperative programming language with parallel support.

$$\begin{array}{rll}
\mathit{var} ::= & \texttt{i} \mid \texttt{j} \mid \ \ldots \ \mid \texttt{s} \mid \texttt{t} \mid \ldots \mid \texttt{x}_1 \mid \texttt{x}_2 \mid \ \ldots \ \mid \texttt{z}_\texttt{n} \mid \mathit{var}[\mathit{exp}] & \text{(Variables)}\\
\mathit{exp} ::= & \mathit{var} \mid \mathit{val} \mid \mathit{op}(\mathit{exp}, \ldots, \mathit{exp}) & \text{(Expression)}\\
\mathit{com} ::= & \mathit{var} = \mathit{exp} \mid \texttt{if}\ \mathit{exp}\ \texttt{then}\ \mathit{com}\ \texttt{else}\ \mathit{com} \mid & \\
& \texttt{while}\ \mathit{exp}\ \texttt{do}\ \mathit{com} \mid \texttt{use}(\mathit{var}, \ldots, \mathit{var}) \mid \texttt{skip} \mid & \\
& \mathit{com};\mathit{com} \mid \texttt{parallel}\{\mathit{com}\}\{\mathit{com}\} \cdots \{\mathit{com}\} & \text{(Command)}
\end{array}$$

Fig. 1. A simple imperative `while` language

The grammar is given Fig. 1. A variable represents either an undetermined "primitive" datatype, e.g., not a reference variable, or an array, whose indices are given by an expression. We generally use `s` and `t` for arrays. An expression is either a variable, a value (e.g., integer literal) or the application to expressions of some operator *op*, which can be e.g., relational (`==`, `<`, etc.) or arithmetic (`+`, `-`, etc.). We let `V` (resp. `e`, `C`) ranges over variables (resp. expression, command) and `W` range over `while` loops. We also use combined assignment operators and write e.g., `x++` for `x += 1`. We assume commands to be correct, e.g., with operators correctly applied to expressions, no out-of-bounds errors, etc.

A program is thus a sequence of statements, each statement being either an *assignment*, a *conditional*, a *while* loop, a *function call*[7]or a *skip*. *Statements* are abstracted into *commands*, which can be a statement, a sequence of commands, or multiple commands to be run in parallel. The semantics of `parallel` is the following: variables appearing in the arguments are considered local, and the value of a given variable `x` after execution of the `parallel` command is the value of the last modified local variable `x`. This implies possible race conditions, but our transformation (detailed in Sect. 3) is robust to those: it assumes given `parallel`-free programs, and introduces `parallel` commands that either uniformly update the (copy of the) variables across commands, or update them in only one command. The rest of this section assumes `parallel`-free programs, that will be given as input to our transformation explained in Sect. 3.1.

For convenience we define the following sets of variables.

[6] OpenMP's `pragma omp parallel` directive is illustrated in Sect. 5.

[7] The `use` command represents any command which does not modify its variables but use them and should not be moved around carelessly (e.g., a `printf`). In practice, we currently treat all function calls as `use`, even if the function is pure.

Definition 1. *Given an expression* $\mathtt{e}$*, we define the variables occurring in* $\mathtt{e}$ *by:*

$$\mathrm{Occ}(\mathtt{x}) = \mathtt{x} \qquad \mathrm{Occ}(\mathtt{t[e]}) = \mathtt{t} \cup \mathrm{Occ}(\mathtt{e})$$

$$\mathrm{Occ}(\mathrm{val}) = \emptyset \qquad \mathrm{Occ}(\mathtt{op}(\mathtt{e}_1, \dots, \mathtt{e}_\mathtt{n})) = \mathrm{Occ}(\mathtt{e}_1) \cup \dots \cup \mathrm{Occ}(\mathtt{e}_\mathtt{n})$$

Definition 2. *Let* $\mathtt{C}$ *be a command, we let* $\mathrm{Out}(\mathtt{C})$ *(resp.* $\mathrm{In}(\mathtt{C})$*,* $\mathrm{Occ}(\mathtt{C})$*) be the set of variables* modified *by (resp.* used by*,* occurring *in)* $\mathtt{C}$ *as defined in Table 1. In the* **use**$(\mathtt{x}_1, \dots, \mathtt{x}_\mathtt{n})$ *case,* $\mathtt{f}$ *is a fresh variable introduced for this command.*

$\mathtt{C}$	$\mathrm{Out}(\mathtt{C})$	$\mathrm{In}(\mathtt{C})$	$\mathrm{Occ}(\mathtt{C}) = \mathrm{Out}(\mathtt{C}) \cup \mathrm{In}(\mathtt{C})$
$\mathtt{x} = \mathtt{e}$	$\mathtt{x}$	$\mathrm{Occ}(\mathtt{e})$	$\mathtt{x} \cup \mathrm{Occ}(\mathtt{e})$
$\mathtt{t[e_1]} = \mathtt{e_2}$	$\mathtt{t}$	$\mathrm{Occ}(\mathtt{e}_1) \cup \mathrm{Occ}(\mathtt{e}_2)$	$\mathtt{t} \cup \mathrm{Occ}(\mathtt{e}_1) \cup \mathrm{Occ}(\mathtt{e}_2)$
if $\mathtt{e}$ **then** $\mathtt{C}_1$ **else** $\mathtt{C}_2$	$\mathrm{Out}(\mathtt{C}_1) \cup \mathrm{Out}(\mathtt{C}_2)$	$\mathrm{Occ}(\mathtt{e}) \cup \mathrm{In}(\mathtt{C}_1) \cup \mathrm{In}(\mathtt{C}_2)$	$\mathrm{Occ}(\mathtt{e}) \cup \mathrm{Occ}(\mathtt{C}_1) \cup \mathrm{Occ}(\mathtt{C}_2)$
while $\mathtt{e}$ **do** $\mathtt{C}$	$\mathrm{Out}(\mathtt{C})$	$\mathrm{Occ}(\mathtt{e}) \cup \mathrm{In}(\mathtt{C})$	$\mathrm{Occ}(\mathtt{e}) \cup \mathrm{Occ}(\mathtt{C})$
use$(\mathtt{x}_1, \dots, \mathtt{x}_\mathtt{n})$	$\mathtt{f}$	$\{\mathtt{x}_1, \dots, \mathtt{x}_\mathtt{n}\}$	$\{\mathtt{x}_1, \dots, \mathtt{x}_\mathtt{n}, \mathtt{f}\}$
skip	$\emptyset$	$\emptyset$	$\emptyset$
$\mathtt{C}_1; \mathtt{C}_2$	$\mathrm{Out}(\mathtt{C}_1) \cup \mathrm{Out}(\mathtt{C}_2)$	$\mathrm{In}(\mathtt{C}_1) \cup \mathrm{In}(\mathtt{C}_2)$	$\mathrm{Occ}(\mathtt{C}_1) \cup \mathrm{Occ}(\mathtt{C}_2)$

Table 1. Definition of Out, In and Occ for commands

Our treatment of arrays is an over-approximation: we consider the array as a single entity, and that changing one value in it changes it completely. This is however satisfactory: since we do not split loop "vertically" (e.g., distributing the iteration space between threads) but "horizontally" (e.g., distributing the tasks between threads), we want each thread in the **parallel** command to have control of the array it modifies, and not to have to synchronize its writes with other commands.

2.2 Data-flow Graphs for Loop Dependency Analysis

The loop transformation algorithm relies fundamentally on its ability to analyze data-flow dependencies between loop condition and variables in the loop body, to identify opportunities for loop fission. In this section we define the principles of this dependency analysis, founded on the theory of *data-flow graphs*, and how it maps to the presented **while** language. This dependency analysis was influenced by a large body of works related to static analysis [1,26,29], semantics [27,38] and optimization [33]; but is presented here in self-contained and compact manner.

We assume the reader is familiar with semi-rings, standard operations on matrices (multiplication and addition), and on graphs (union and inclusion).

Definition of Data-Flow Graphs A data-flow graph for a given command $\mathtt{C}$ is a weighted relation on the set $\mathrm{Occ}(\mathtt{C})$. Formally, this is represented as a matrix over a semi-ring, with the implicit choice of a denumeration of $\mathrm{Occ}(\mathtt{C})$.[8]

[8] We will use the order in which the variables occur in the program as their implicit order most of the time.

Definition 3 (DFG). *A* data-flow graph *(DFG) for a command* C *is a* $|\operatorname{Occ}(\mathtt{C})| \times |\operatorname{Occ}(\mathtt{C})|$ *matrix over a fixed semi-ring* $(\mathcal{S}, +, \times)$*, with* $|\operatorname{Occ}(\mathtt{C})|$ *the cardinal of* $\operatorname{Occ}(\mathtt{C})$*. We write* $\mathbb{M}(\mathtt{C})$ *the DFG of* C *and* $\mathbb{M}(\mathtt{C})(\mathtt{y}, \mathtt{x})$ *for the coefficient in* $\mathbb{M}(\mathtt{C})$ *at the row corresponding to* x *and column corresponding to* y*.*

How a data-flow graph is constructed, by induction over the command, is explained in Sect. 2.3. To avoid resizing matrices whenever additional variables are considered, we identify $\mathbb{M}(\mathtt{C})$ with its embedding in a larger matrix, i.e., we abusively call the DFG of C any matrix containing $\mathbb{M}(\mathtt{C})$ and the multiplication identity element on the other diagonal coefficients, implicitly viewing the additional rows/columns as variables not in $\operatorname{Occ}(\mathtt{C})$.

2.3 Constructing Data-Flow Graphs

The data-flow graph (DFG) of a command is constructed by induction on the structure of the command. In the remainder of this paper, we use the semiring $(\{0, 1, \infty\}, \max, \times)$ to represent dependencies: ∞ represents *dependence*, 1 represents *propagation*, and 0 represents *reinitialization*.

Base cases (assignment, skip, use) The DFG for an assignment C is computed using $\operatorname{In}(\mathtt{C})$ and $\operatorname{Out}(\mathtt{C})$:

Definition 4 (Assignment). *Given an assignment* C*, its DFG is given by:*

$$\mathbb{M}(\mathtt{C})(\mathtt{y}, \mathtt{x}) = \begin{cases} \infty & \text{if } \mathtt{x} \in \operatorname{Out}(\mathtt{C}) \text{ and } \mathtt{y} \in \operatorname{In}(\mathtt{C}) & \text{(Dependence)} \\ 1 & \text{if } \mathtt{x} = \mathtt{y} \text{ and } \mathtt{x} \notin \operatorname{Out}(\mathtt{C}) & \text{(Propagation)} \\ 0 & \text{otherwise} & \text{(Reinitialization)} \end{cases}$$

We illustrate in Fig. 2 some basic cases and introduce the graphical conventions of using weighted relations, or weighted bi-partite graphs, to illustrate the matrices. Note that in the case of dependencies, $\operatorname{In}(\mathtt{C})$ is exactly the set of variables that are source of a dependence arrow, while $\operatorname{Out}(\mathtt{C})$ is the set of variables that either are targets of dependence arrows or were reinitialized.

Note that we over-approximate arrays in two ways: the dependencies of the value at one index are the dependencies of the whole array, and the index at which the value is assigned is a dependence of the whole array (cf. the solid arrow from i to t in the last example of Fig. 2). This is however enough for our purpose, and simplify our treatment of arrays.

The DFG for **skip** is simply the empty matrix, but the DFG of **use** function calls requires a fresh "effect" variable to anchor the dependencies.

Definition 5 (skip). *We let* $\mathbb{M}(\mathtt{skip})$ *be the matrix with* 0 *rows and columns.*[9]

Definition 6 (use). *We let* $\mathbb{M}(\mathtt{use}(\mathtt{x}_1, \ldots, \mathtt{x}_n))$ *be the matrix with coefficients from each* $\mathtt{x}_i$ *to* f*, and from* f *to* f *equal to* ∞*, and* 0 *coefficients otherwise, for* f *a freshly introduced variable. Graphically, we get:*

[9] Identifying the DFG with its embeddings, it is hence the identity matrix of any size.

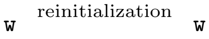
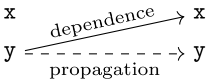
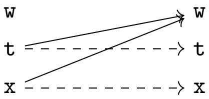
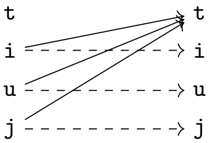
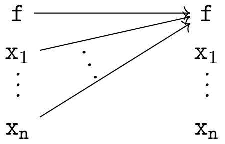
$\mathtt{C}$	$\mathrm{Out}(\mathtt{C})$, $\mathrm{In}(\mathtt{C})$	$\mathbb{M}(\mathtt{C})$ (as a graph)	$\mathbb{M}(\mathtt{C})$
$\mathtt{w} = 3$	$\mathrm{Out}(\mathtt{C}) = \{\mathtt{w}\}$ $\mathrm{In}(\mathtt{C}) = \emptyset$		$\begin{array}{cc} & \mathtt{w} \\ \mathtt{w} & \begin{pmatrix} 0 \end{pmatrix} \end{array}$
$\mathtt{x} = \mathtt{y}$	$\mathrm{Out}(\mathtt{C}) = \{\mathtt{x}\}$ $\mathrm{In}(\mathtt{C}) = \{\mathtt{y}\}$		$\begin{array}{cc} & \begin{array}{cc} \mathtt{x} & \mathtt{y} \end{array} \\ \begin{array}{c} \mathtt{x} \\ \mathtt{y} \end{array} & \begin{pmatrix} 0 & 0 \\ \infty & 1 \end{pmatrix} \end{array}$
$\mathtt{w} = \mathtt{t}[\mathtt{x} + 1]$	$\mathrm{Out}(\mathtt{C}) = \{\mathtt{w}\}$ $\mathrm{In}(\mathtt{C}) = \{\mathtt{t}, \mathtt{x}\}$		$\begin{array}{cc} & \begin{array}{ccc} \mathtt{w} & \mathtt{t} & \mathtt{x} \end{array} \\ \begin{array}{c} \mathtt{w} \\ \mathtt{t} \\ \mathtt{x} \end{array} & \begin{pmatrix} 0 & 0 & 0 \\ \infty & 1 & 0 \\ \infty & 0 & 1 \end{pmatrix} \end{array}$
$\mathtt{t}[\mathtt{i}] = \mathtt{u} + \mathtt{j}$	$\mathrm{Out}(\mathtt{C}) = \{\mathtt{t}\}$ $\mathrm{In}(\mathtt{C}) = \{\mathtt{i}, \mathtt{u}, \mathtt{j}\}$		$\begin{array}{cc} & \begin{array}{cccc} \mathtt{t} & \mathtt{i} & \mathtt{u} & \mathtt{j} \end{array} \\ \begin{array}{c} \mathtt{t} \\ \mathtt{i} \\ \mathtt{u} \\ \mathtt{j} \end{array} & \begin{pmatrix} 0 & 0 & 0 & 0 \\ \infty & 1 & 0 & 0 \\ \infty & 0 & 1 & 0 \\ \infty & 0 & 0 & 1 \end{pmatrix} \end{array}$

Fig. 2. Statement examples, sets, and representations of their dependences



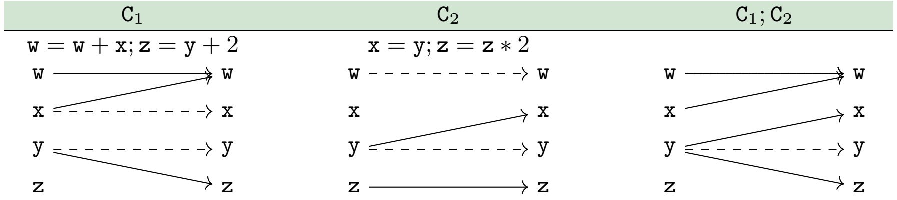
Composition and multipaths The definition of DFG for a (sequential) *composition* of commands is an abstraction that allows treating a block of statements as one command with its own DFG.

Definition 7 (Composition). *We let* $\mathbb{M}(\mathtt{C}_1; \dots; \mathtt{C}_n)$ *be* $\mathbb{M}(\mathtt{C}_1) \times \cdots \times \mathbb{M}(\mathtt{C}_n)$.

For two graphs, the product of their matrices of weights is represented in a standard way, as a graph of length 2 paths; as illustrated in Fig. 3—where $\mathtt{C}_1$ and $\mathtt{C}_2$ are themselves already the result of compositions of assignments involving disjoint variables, and hence straightforward to compute.

Correction Conditionals and loops both requires a *correction* to compute their DFGs. Indeed, the DFGs of **if** $\mathtt{e}$ **then** $\mathtt{C}_1$ **else** $\mathtt{C}_2$ and **while** $\mathtt{e}$ **do** $\mathtt{C}$ require more than the DFG of its body. The reason for this is that all the modified variables in $\mathtt{C}_1$ and $\mathtt{C}_2$ or $\mathtt{C}$ (e.g., $\mathrm{Out}(\mathtt{C}_1) \cup \mathrm{Out}(\mathtt{C}_2)$ or $\mathrm{Out}(\mathtt{C})$) depend on the variables occurring in $\mathtt{e}$ (e.g., in $\mathrm{Occ}(\mathtt{e})$). To reflect this, a *correction* is needed:

Definition 8 (Correction). *For* $\mathtt{e}$ *an expression and* $\mathtt{C}$ *a command, we define* $\mathtt{e}$'s correction for $\mathtt{C}$, $\mathrm{Corr}(\mathtt{e})_{\mathtt{C}}$, *to be* $E^t \times O$, *for*



$$\begin{array}{c|cccc} & \mathtt{w} & \mathtt{x} & \mathtt{y} & \mathtt{z} \\ \hline \mathtt{w} & \infty & 0 & 0 & 0 \\ \mathtt{x} & \infty & 1 & 0 & 0 \\ \mathtt{y} & 0 & 0 & 1 & \infty \\ \mathtt{z} & 0 & 0 & 0 & 0 \end{array} \times \begin{array}{c|cccc} & \mathtt{w} & \mathtt{x} & \mathtt{y} & \mathtt{z} \\ \hline \mathtt{w} & 1 & 0 & 0 & 0 \\ \mathtt{x} & 0 & 0 & 0 & 0 \\ \mathtt{y} & 0 & \infty & 1 & 0 \\ \mathtt{z} & 0 & 0 & 0 & \infty \end{array} = \begin{array}{c|cccc} & \mathtt{w} & \mathtt{x} & \mathtt{y} & \mathtt{z} \\ \hline \mathtt{w} & \infty & 0 & 0 & 0 \\ \mathtt{x} & \infty & 0 & 0 & 0 \\ \mathtt{y} & 0 & \infty & 1 & \infty \\ \mathtt{z} & 0 & 0 & 0 & 0 \end{array}$$

Fig. 3. Data-Flow Graph of Composition.

- E^t *the (column) vector with coefficient equal to* ∞ *for the variables in* $\mathrm{Occ}(\mathtt{e})$ *and* 0 *for all the other variables,*
- O *the (row) vector with coefficient equal to* ∞ *for the variables in* $\mathrm{Out}(\mathtt{C})$ *and* 0 *for all the other variables.*

As an example, let us re-use the programs $\mathtt{C}_1$ and $\mathtt{C}_2$ from Fig. 3, to construct $\mathtt{w} > \mathtt{x}$'s correction for $\mathtt{C}_1;\mathtt{C}_2$, that we write $\mathrm{Corr}(\mathtt{w} > \mathtt{x})_{\mathtt{C}_1;\mathtt{C}_2}$:

E^t	O	$E^t \times O$
$\begin{array}{c|c} \mathtt{w} & \infty \\ \mathtt{x} & \infty \\ \mathtt{y} & 0 \\ \mathtt{z} & 0 \end{array}$	$\begin{array}{ccccccc} & \mathtt{w} & \mathtt{x} & \mathtt{y} & \mathtt{z} & \\ & (\infty & 0 & 0 & \infty) & (\mathrm{Out}(\mathtt{C}_1)) \\ + & (0 & \infty & 0 & \infty) & (\mathrm{Out}(\mathtt{C}_2)) \\ = & (\infty & \infty & 0 & \infty) & (\mathrm{Out}(\mathtt{C}_1;\mathtt{C}_2)) \end{array}$	$\begin{array}{c|cccc} & \mathtt{w} & \mathtt{x} & \mathtt{y} & \mathtt{z} \\ \hline \mathtt{w} & \infty & \infty & 0 & \infty \\ \mathtt{x} & \infty & \infty & 0 & \infty \\ \mathtt{y} & 0 & 0 & 0 & 0 \\ \mathtt{z} & 0 & 0 & 0 & 0 \end{array}$

This last matrix represents the fact that `w` and `x`, through the expression `w > x`, control the values of `w`, `x` and `z` if $\mathtt{C}_1$ and $\mathtt{C}_2$'s execution depend of it.

Conditionals. To construct the DFG of **`if`** `e` **`then`** $\mathtt{C}_1$ **`else`** $\mathtt{C}_2$, there are two aspects to consider:

1. First, our analysis does not seek to evaluate whether $\mathtt{C}_1$ or $\mathtt{C}_2$ will get executed. Instead, it will overapproximate and assume that both will get executed, hence using $\mathbb{M}(\mathtt{C}_1) + \mathbb{M}(\mathtt{C}_2)$.
2. Second, all the variables assigned in $\mathtt{C}_1$ and $\mathtt{C}_2$ (e.g., $\mathrm{Out}(\mathtt{C}_1) \cup \mathrm{Out}(\mathtt{C}_2)$) depends on the variables occurring in `e`. For this reason, $\mathrm{Corr}(\mathtt{e})_{\mathtt{C}_1;\mathtt{C}_2}$ needs to be added to the previous matrix.

Putting it together, we obtain:

Definition 9 (`if`). *We let* $\mathbb{M}($**`if`** $\mathtt{e}$ **`then`** $\mathtt{C}_1$ **`else`** $\mathtt{C}_2)$ *be* $\mathbb{M}(\mathtt{C}_1)+\mathbb{M}(\mathtt{C}_2)+\mathrm{Corr}(\mathtt{e})_{\mathtt{C}_1;\mathtt{C}_2}$.

Re-using the programs $\mathtt{C}_1$ and $\mathtt{C}_2$ from Fig. 3 and $\mathrm{Corr}(\mathtt{w} > \mathtt{x})_{\mathtt{C}_1;\mathtt{C}_2}$, we obtain:

$$\mathbb{M}\begin{pmatrix}\textbf{if}(\mathtt{w>x}) \\ \quad \textbf{then } \mathtt{w} = \mathtt{w} + \mathtt{x}; \\ \qquad\qquad \mathtt{z} = \mathtt{y} + 2 \\ \quad \textbf{else } \mathtt{x} = \mathtt{y}; \\ \qquad\qquad \mathtt{z} = \mathtt{z} * 2\end{pmatrix} = \begin{array}{c@{}c} & \begin{array}{cccc} \mathtt{w} & \mathtt{x} & \mathtt{y} & \mathtt{z}\end{array} \\ \begin{array}{c}\mathtt{w}\\ \mathtt{x}\\ \mathtt{y}\\ \mathtt{z}\end{array} & \begin{pmatrix}\infty & 0 & 0 & 0\\ \infty & \boxed{1} & 0 & 0\\ 0 & 0 & 1 & \infty\\ 0 & 0 & 0 & 0\end{pmatrix}\end{array} + \begin{array}{c@{}c} & \begin{array}{cccc} \mathtt{w} & \mathtt{x} & \mathtt{y} & \mathtt{z}\end{array} \\ \begin{array}{c}\mathtt{w}\\ \mathtt{x}\\ \mathtt{y}\\ \mathtt{z}\end{array} & \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & \boxed{0} & 0 & 0\\ 0 & \infty & 1 & 0\\ 0 & 0 & 0 & \infty\end{pmatrix}\end{array} + \begin{array}{c@{}c} & \begin{array}{cccc} \mathtt{w} & \mathtt{x} & \mathtt{y} & \mathtt{z}\end{array} \\ \begin{array}{c}\mathtt{w}\\ \mathtt{x}\\ \mathtt{y}\\ \mathtt{z}\end{array} & \begin{pmatrix}\infty & \infty & \enclose{circle}{0} & \infty\\ \infty & \boxed{\infty} & 0 & \infty\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\end{pmatrix}\end{array}$$

The boxed value represents the impact of x on itself: $\mathtt{C}_1$ has the value 1, since x is not assigned in it. On the other hand, $\mathtt{C}_2$ has 0 for coefficient, since the value of x is reinitialized in it. The correction, however, has a ∞, to represent the fact that the value of x controls the values assigned in the body of $\mathtt{C}_1$ and $\mathtt{C}_2$—and x itself is one of them. As a result, we have again the value ∞ in the matrix summing them three, since x controls the value it gets assigned to itself—as it controls which branch ends up being executed. On the other hand, the circled value at $(\mathtt{w}, \mathtt{y})$ is a 0 since y's value is not controlled by w, since neither $\mathtt{C}_1$ nor $\mathtt{C}_2$ assign y: regardless of e's truth value, y's value will remain the same.

While Loops. To define the DFG of a command **while** e **do** C from $\mathbb{M}(\mathtt{C})$, we need, as for conditionals, the correction $\mathrm{Corr}(\mathtt{e})_{\mathtt{C}}$, to account for the fact that all the modified variables in C depend on the variables used in e:

Definition 10 (while). *We let* $\mathbb{M}(\textbf{while}\ \mathtt{e}\ \textbf{do}\ \mathtt{C})$ *be* $\mathbb{M}(\mathtt{C}) + \mathrm{Corr}(\mathtt{e})_{\mathtt{C}}$.[10]

As an example, we let the reader convince themselves that the DFG of

```
while(t[i] != j){
  s1[i] = j*j;
  s2[i] = 1/j;
  i++
}
```

is

$$\begin{array}{c@{}c} & \begin{array}{ccccc} \mathtt{t} & \mathtt{i} & \mathtt{j} & \mathtt{s1} & \mathtt{s2}\end{array} \\ \begin{array}{c}\mathtt{t}\\ \mathtt{i}\\ \mathtt{j}\\ \mathtt{s1}\\ \mathtt{s2}\end{array} & \begin{pmatrix}1 & \infty & 0 & \infty & \infty\\ 0 & \infty & 0 & \infty & \infty\\ 0 & \infty & 1 & \infty & \infty\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\end{pmatrix}\end{array}$$

. Intuitively, one can note that the rows for s1 and s2 are filled with 0s, since those variables do not control any other variable and are assigned in the body of the loop. On the other hand, t, i and j all three control the values of i, s1 and s2, since they determine if the body of the loop will execute. The variables t and j are the only one whose value is propagated (e.g., with a 1 on their diagonal), since they are not assigned in this short example. The command i++ is the only command that has the potential to impact the loop's condition. We call it an update command:

Definition 11 (Update command). *Given a loop* $\mathtt{W} := \textbf{while}\ \mathtt{e}\ \textbf{do}\ \mathtt{C}$, *the* update commands $\mathtt{C_u}$ *are the commands in* C *such that* $\mathbb{M}(\mathtt{W})(\mathtt{y}, \mathtt{x}) = \infty$ *for* $\mathtt{x} \in \mathrm{Out}(\mathtt{C_u})$ *and* $\mathtt{y} \in \mathrm{Occ}(\mathtt{e})$.

[10] This is different from our previous treatment of **while** loop [33, Definition 5], that required to compute the transitive closure of $\mathbb{M}(\mathtt{C})$: for the transformation we present in Sect. 3, this is not needed, as all the relevant dependencies are obtained immediately—this also guarantees that our analysis can distribute loop-carried dependencies.

3 Loop Fission Algorithm

We now present our loop transformation technique and prove its correctness.

3.1 Algorithm, Presentation and Intuition

Our algorithm, presented in Algo. 1, requires essentially to

1. Pick a loop at top level,
2. Compute its condensation graph (Def. 13)—this requires first the dependence graph (Def. 12), which itself uses the DFG,
3. Compute a covering (Def. 14) of the condensation graph,
4. Create a loop per element of the covering.

Even if our technique could distribute nested loops, it would require adjustments that we prefer to omit to simplify our presentation. None of our examples in this paper require to distribute nested loops. Note, however, that our algorithm handles loops containing themselves loops.

Definition 12 (Dependence graph). *The* dependence graph *of the loop* $\texttt{W} :=$ **while** $\texttt{e}$ **do** $\{\texttt{C}_1; \cdots; \texttt{C}_\texttt{n}\}$ *is the graph whose vertices is the set of commands* $\{\texttt{C}_1; \cdots; \texttt{C}_\texttt{n}\}$*, and there exists a directed edge from* $\texttt{C}_i$ *to* $\texttt{C}_j$ *if and only if there exists variables* $\texttt{x} \in \mathrm{Out}(\texttt{C}_j)$ *and* $\texttt{y} \in \mathrm{In}(\texttt{C}_i)$ *such that* $\mathbb{M}(\texttt{W})(\texttt{y}, \texttt{x}) = \infty$*.*

The last example of Sect. 2.3 gives $\overset{\circlearrowleft}{\texttt{s1[i] = j*j}} \longrightarrow \overset{\circlearrowleft}{\texttt{i++}} \longleftarrow \overset{\circlearrowleft}{\texttt{s2[i] = 1/j}}$. Note that all the commands in the body of the loop are the sources of dependence edges whose target is the update commands: for our example, this means that every command will be the source of an arrow whose target is `i++`. This comes from the correction, even if the condition does not explicitly appear in the dependence graph.

The remainder of the loop transforming principle is simple: once the graph representing the dependencies between commands is obtained, it remains to determine the cliques in the graph and forms *strongly connected components* (SCCs); and then to separate the SCCs into subgraphs to produce the final parallelizable loops that contain a copy of the loop header and update commands.

Definition 13 (Graph helpers). *Given the dependence graph of a loop* $\texttt{W}$*,*

- *its* strongly connected components *(*SCC*s) are its strongly connected subgraphs,*
- *its* condensation graph $\mathrm{G}_\texttt{W}$ *is the graph whose vertices are* SCC*s and edges are the edges whose source and target belong to distinct* SCC*s.*

In our example, the SCCs are the nodes themselves, and the condensation graph is $\texttt{s1[i] = j*j} \longrightarrow \texttt{i++} \longleftarrow \texttt{s2[i] = 1/j}$. Excluding the update command `i++`, there are now two nodes in the condensation graph, and we can construct the parallel loops by 1. inserting a `parallel` command, 2. duplicating the loop header and update command, 3. inserting the command in the remaining nodes of the condensation graph in each loop. For our example, we obtain, as expected,

$$\texttt{parallel} \left\{ \begin{array}{l} \texttt{while(t[i] != j)\{} \\ \quad \texttt{s1[i] = j*j;} \\ \quad \texttt{i++\}} \end{array} \right\} \left\{ \begin{array}{l} \texttt{while(t[i] != j)\{} \\ \quad \texttt{s2[i] = 1/j;} \\ \quad \texttt{i++\}} \end{array} \right\} .$$

Formally, what we just did was to split the *saturated covering*.

Definition 14 (Coverings [16]). *A* covering *of a graph G is a collection of subgraphs $G_1, G_2, \ldots, G_j$ such that $G = \cup_{i=1}^{j} G_i$.*

A saturated covering *of G is a covering $G_1, G_2, \ldots, G_k$ such that for all edge in G with source in G_i, its target belongs to G_i as well. It is* proper *if none of the subgraph is a subgraph of another.*

The algorithm then simply consists in finding a proper saturated covering of the loop's condensation graph, and to split the loop accordingly. In our example, the only proper saturated covering is

$$\{ \texttt{s1[i] = j*j} \longrightarrow \texttt{i++} \ , \ \texttt{i++} \longleftarrow \texttt{s2[i] = 1/j} \ \}.$$

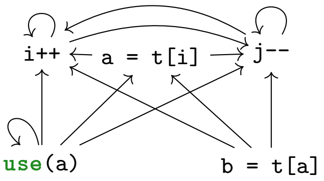
If the covering was not proper, then the `i++` node on its own would be in it, leading to create a useless loop that performs nothing but updating its own condition.

Algorithm 1 Loop fission

```
Input: A loop $\texttt{W} := \texttt{while e do} \{\texttt{C}_1; \cdots ; \texttt{C}_\texttt{n}\}$        ▷ Pick a loop W at top level
  Compute the condensation graph $\mathrm{G}_\texttt{W}$ of W,                  ▷ cf. Def. 13
  Compute the saturated covering $G_1, \ldots, G_j$ of $\mathrm{G}_\texttt{W}$:          ▷ cf. Def. 14
  while a node $n$ in $\mathrm{G}_\texttt{W}$ is not part of a subgraph $G_l$ do
      Create a new subgraph $G_i$ containing $n$,
      Recursively add to $G_i$ the nodes targeted by edges whose source is in $G_i$,
  Compute the proper saturated covering $G_1, \ldots, G_k$ of $\mathrm{G}_\texttt{W}$:
  for all $G_i$ in the saturated covering do
      If $\exists G_l$ in the saturated covering s.t. $G_i$ is a subgraph of $G_l$, then remove $G_i$
  end for
  Create one while loop per subgraph in the proper saturated covering:
  for all $G_i$ in the proper saturated covering do
      Let $\texttt{W}_i := \texttt{while e do} \{\texttt{C}_{i_1}; \cdots ; \texttt{C}_{i_m}\}$ where $\{\texttt{C}_{i_1}, \ldots, \texttt{C}_{i_m}\}$ are the vertices of $G_i$,
  inserted in the same order as they are in W.
  end for
Output: if $k > 1$, $\tilde{\texttt{W}} := \texttt{parallel}\{\texttt{W}_1\}\{ \ldots \}\{\texttt{W}_k\}$, else $\tilde{\texttt{W}} := \texttt{W}$.
```

Sometimes, duplicating commands that are not update commands is needed to split the loop. We illustrate this principle with a more complex example that involve function call and multiple update commands in Fig. 4.

$$\mathbb{M}\left(\begin{array}{l}\texttt{while(i != j)\{}\\ \quad\texttt{i++;}\\ \quad\texttt{j--;}\\ \quad\texttt{a = t[i];}\\ \quad\texttt{use(a);}\\ \quad\texttt{b = t[a]}\\ \texttt{\}}\end{array}\right) = \begin{array}{c|cccccc} & \texttt{i} & \texttt{j} & \texttt{a} & \texttt{t} & \texttt{b} & \texttt{f} \\ \hline \texttt{i} & \infty & \infty & \infty & 0 & \infty & \infty \\ \texttt{j} & \infty & \infty & \infty & 0 & \infty & \infty \\ \texttt{a} & 0 & 0 & 0 & 0 & \infty & \infty \\ \texttt{t} & 0 & 0 & \infty & 1 & \infty & 0 \\ \texttt{b} & 0 & 0 & 0 & 0 & 0 & 0 \\ \texttt{f} & 0 & 0 & \infty & 0 & 0 & \infty \end{array}$$



The proper saturated covering has two subgraphs: one contains everything but `use(a)` and the other contains everything but `b = t[a]`. Since both `use(a)` and `b = t[a]` depend on `a = t[i]`, this latter command needs to be duplicated, even if it is not an update command:

```
parallel { while(i != j){      } { while(i != j){   }
         {   i++;              } {   i++;           }
         {   j--;              } {   j--;           }
         {   a = t[i];         } {   a = t[i];      }
         {   use(a)}           } {   b = t[a]}      }
```

Fig. 4. Distributing a more complex **while** loop

3.2 Correctness of the Algorithm

We now need to prove that the semantics of the initial loop W is equal to the semantics of $\tilde{\texttt{W}}$ given by Algo. 1. This is done by showing that for any variable x appearing in W, its final value after running W is equal to its final value after running $\tilde{\texttt{W}}$. We first prove that the loops in $\tilde{\texttt{W}}$ has the same iteration space as W:

Lemma 1. *The loops in $\tilde{\texttt{W}}$ have the same number of iterations as* W*.*

Proof. Let $\texttt{W}_i$ be a loop in $\tilde{\texttt{W}}$. By property of the saturated covering, the update commands are in the body of $\texttt{W}_i$: there is always an edge from any command to the update commands due to the loop correction, and hence the update commands are part of all the subgraphs in the saturated covering. Furthermore, if there exists a command C that is the target of an edge whose source is an update command $\texttt{C}_\texttt{u}$, then C and $\texttt{C}_\texttt{u}$ are always both present in any subgraph of the saturated covering. Indeed, since there are edges from $\texttt{C}_\texttt{u}$ to C and from C to $\texttt{C}_\texttt{u}$, they are part of the same node in the condensation graph.

Since the condition of $\texttt{W}_i$ is the same as the condition of W, and since all the instructions that impact (directly or indirectly) the variables occurring in that condition are present in $\texttt{W}_i$, we conclude that the number of iterations of $\texttt{W}_i$ and W are equal. □

Theorem 1. *The transformation* $\texttt{W} \rightsquigarrow \tilde{\texttt{W}}$ *given in Algo. 1 preserves the semantic.*

Proof (sketch). We show that for every variable x, the value of x after the execution of W is equal to the value of x after the execution of $\tilde{\texttt{W}}$. Variables are considered local to each loop $\mathtt{W}_i$ in $\tilde{\mathtt{W}}$, so we need to avoid race condition. To do so, we prove the following more precise result: for each variable `x` and each loop $\mathtt{W}_i$ in $\tilde{\mathtt{W}}$ in which the value of `x` is modified, the value of `x` after executing `W` is equal to the value of `x` after executing $\mathtt{W}_i$.

The previous claim is then straightforward to prove, based on the property of the covering. One shows by induction on the number of iterations k that for all the variables $\mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{h}$ appearing in $\mathtt{W}_i$, the values of $\mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{h}$ after k loop iterations of $\mathtt{W}_i$ are equal to the values of $\mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{h}$ after k loop iterations of `W`. Note some other variables may be affected by the latter but the variables $\mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{h}$ do not depend on them (otherwise, they would also appear in $\mathtt{W}_i$ by definition of the dependence graph and the covering). Since the number of iteration match (Lemma 1), the claim is proven. □

4 Limitations of Existing Alternative Approaches

In the beginning of this paper, we made the bold claim that other loop fission approaches do not handle unknown iteration spaces, which makes our loop-agnostic technique interesting. In this section we discuss these alternative approaches, their capabilities, and provide evidence to support this claim. We also give justification for the need to introduce our loop analysis into this landscape.

4.1 Comparing Dependency Analyses

Since its first inception, loop fission [2] has been implemented using different techniques and dependency mechanisms. Program dependence graph (PDG) [18] can be used to identify when a loop can be distributed [3, p. 844], but other—sometimes simpler—mechanisms are often used in practice. For instance, a patch integrating loop fission into LLVM [28] tuned the simpler data dependence graph (DDG) to obtain a Loop Fission Interference Graph (FIG) [30]. GCC, on the other hand, build a partition dependence graph (PG) based on the data dependency given by a reduced dependence graph (RG) to perform the same task [19]. In this paper, we introduce another loop dependency analysis, not to further obfuscate the landscape, but because it allows us to express our algorithm simply and—more importantly—to verify it mathematically.[11]

We assume that the more complex mechanisms listed above (PDG, DDG or PG) could be leveraged to implement our transformation, but found it more natural to express ourselves in this language. We further believe that the way we compute the data dependencies is among the lightest, and with a very low memory footprint, as it requires only one pass on the source code to construct a matrix whose size is the number of variables in the program.

[11] This analysis also shares interesting links to a static analysis of values growth [10,9], as discussed more in-depth in a first draft [7].

4.2 Assessment of Existing Automated Loop Transformation and Parallelization Tools

While we conjecture that other mechanisms *could*, in theory, treat loops of any kind like we do, we now substantiate our claim that none of them do: in short, any loop with non-basic condition or update statement is excluded from the optimizations we now discuss. We limit this consideration to tools that support `C` language transformations, because it is our choice implementation language for experimental evaluation in Sect. 5. We also focus on presenting the kinds of loops that other "popular" [35] automatic loop transformation frameworks *do not* distribute, but that our algorithm can distribute. In particular, we do not discuss loops containing control-flow modifiers (such as **`break`**`;` or **`continue`**`;`): neither our algorithm nor OpenMP nor the underlying dependency mechanisms of the discussed tools—to the best of our knowledge—can accommodate those.

Tools that fit the above specification include Cetus, a compiler infrastructure for the source-to-source transformation; Clava, a `C/C++` source-to-source tool based on Clang; Par4All, an automatic parallelizing and optimizing compiler; Pluto, an automatic parallelizer and locality optimizer for affine loop nests; ROSE, a compiler-based infrastructure for building source-to-source program transformations and analysis tools; Intel's `C++` compiler (icc), and TRACO, an automatic parallelizing and optimizing compiler, based on the transitive closure of dependence graphs. While these tools perform various automatic transformations and optimizations, only ROSE and icc perform loop fission [35, Section 3.1].

Based on our assessment, most of these tools process only *canonical loops*:

Definition 15 (Canonical Loop [25, 4.4.1 Canonical Loop Nest Form]). *A* canonical loop *is a loop of the form*

```
for (init-expr; test-expr; incr-expr) structured-block
```

for `incr-expr` *a (single) increment or decrement by a constant or a variable, and* `test-expr` *a single comparison between a variable and a variable or a constant.*

Additional constraints on loop dependences are sometimes needed, e.g., the absence of loop-carried dependency for Cetus. It seems further that some tools cannot parallelize loops whose body contains e.g., **`if`** or **`switch`** statements [35, p. 18], but we have not investigated this claim further. However, our algorithm can handle **`if`**—and **`switch`** too, if it was part of our syntax—present in the body of the loop seamlessly.

It is always hard to infer the absence of support, but we evaluated the lack of formal discussion or example of e.g., **`while`** loop to be sufficient to determine that the tool cannot process **`while`** loops, unless of course they can trivially be transformed into **`for`** loops of the required form [39, p. 236]. We refer to a recent study [35, Section 2] for more detail on those notions and on the limitations of some of the tools discussed in Table 2.

Name	Fission	`for` loop	`while` loop	`do` ... `while` loop	ref.
Cetus	–	In canonical form	–		[17, p. 39], [11, p. 761]
Clava	–	In canonical form	–		[6]
icc	✓	Only if countable		–	[23, p. 2126]
Par4All	–	Unknown			[4,5]
Pluto	–	Only static control structures			[15]
ROSE	✓	In canonical form	–		[36, p. 124]
TRACO	–	In canonical form	–		[34]
OpenMP	–	In canonical form	–		[25]

Table 2. Feature support comparison of automated transformation and parallelization tools.

5 Evaluation

We performed an experimental evaluation of our loop fission technique on a suite of parallel benchmarks. Taking the sequential baseline, we applied the loop fission transformation and parallelization. We compared the result of our technique to the baseline and to an alternative loop fission method implemented in ROSE.

We conducted this experiment in `C` programming language because it naturally maps to the syntax of the imperative **`while`** language presented in Sect. 2. We implement the `parallel` command as OpenMP directives. For instance, the sequential baseline program on the left of Fig. 5 becomes the parallel version on right,[12] after applying our loop fission transformation and parallelization.

The evaluation experimentally substantiated two claims about our technique:

1. It can parallelize loops that are completely ignored by other automatic loop transformation tools, and results in appreciable gain, upper-bounded by the number of parallelizable loops produced by loop fission.
2. Concerning loops that other automatic loop transformation tools can distribute, it yields comparable results in speedup potential. We also demonstrate how insertion of parallel directives can be automated, which supports the practicality of our method.

These results combined confirm that our loop fission technique can easily be integrated into existing tools to improve the performances of the resulting code.

5.1 Benchmarks

Special consideration was necessary to prepare an appropriate benchmark suite for evaluation. We wanted to test our technique on a range of standard problems, across different domains and data sizes, and to include problems containing **`while`** loops. Because our technique is specifically designed for loop fission, we also needed to identify problems that offered potential to apply this transformation.

[12] This example is inspired by benchmark `bicg` from PolyBench/C and presented in our artifact.

```
j = 0;
while (j<M)
{
  s[j] += r[j]*A[j];
  q[j] += A[j]*p[j];
  j++;
}
```

```
#pragma omp parallel private(j)
{ // Each "pragma" block below
  // have its own copy of j.
  #pragma omp single nowait
  { // "nowait" lets the next
    // block start in parallel.
    j = 0;
    while (j<M) {
      s[j] += r[j]*A[j];
      j++;
    }
  }
  #pragma omp single
  {
    j = 0;
    while (j<M) {
      q[j] += A[j]*p[j];
      j++;
    }
  }
} // Both blocks must be terminated
  // before passing this point.
```

Fig. 5. Code transformation example

Finding a suite to fit these parameters is challenging, because standard parallel programming benchmark suites offer mixed opportunity for various program optimizations and focus on loops in canonical form.

We resolved this challenge by preparing a curated set, pooling from three standard parallel programming benchmark suites. PolyBench/C is a polyhedral benchmark suite, representing e.g., linear algebra, data mining and stencils; and commonly used for measuring various loop optimizations. NAS Parallel Benchmarks are designed for performance evaluation of parallel supercomputers, derived from computational fluid dynamics applications. MiBench is an embedded benchmark suite, with everyday programming applications e.g., image-processing libraries, telecommunication, security and office equipment routines. From these suites, we extracted problems that offered potential for loop fission, or already assumed expected form, resulting in 12 benchmarks. We detail these benchmarks in Table 4. Because these three suites are not mutually compatible, we leveraged the timing utilities from PolyBench/C to establish a common and comparable measurement strategy. To assess performance of other kinds of loops that our algorithm can distribute, but which do not occur prevalently in these benchmarks, we converted a portion of problems to use `while` loops.

Comparison target We compared our approach to ROSE Compiler. It is a rich compiler architecture that offers various program transformations and automatic parallelization, and supports multiple compilation targets. ROSE's built-in LoopProcessor tool supports loop fission for C-to-C programs. This input/output specification was necessary to allow observation of the transformation results and fit with the measurement strategy we defined previously. To our knowledge, ROSE is the only tool that satisfies these evaluation requirements.

Experimental setup We ran the benchmarks using a Linux 5.10.0-18-amd64 #1 SMP Debian 5.10.140-1 (2022-09-02) x86_64 GNU/Linux machine, with 4 Intel(R) Core(TM) i5-6300U CPU @ 2.40GHz processors, and gcc compiler version 7.5.0. The evaluation was performed in a containerized environment on Docker version 20.10.18, build b40c2f6. For each benchmark, we recorded the clock time 5 times, excluded min and max, and averaged the remaining 3 times to obtain the result. We constrained variance between recorded times not to exceed 5%. We ran experiments on 5 input data sizes, as defined in PolyBench/C: MINI, SMALL, MEDIUM, LARGE and EXTRALARGE (abbr. XS, S, M, L, XL). We also tested 4 gcc compiler optimization levels -O0 through -O3. Speedup is the ratio of sequential and parallel executions, $S = T_{\text{Seq}}/T_{\text{Par}}$, where a value greater than 1 indicates parallel is outperforming the sequential execution. In presentation of these results, the sequential benchmarks are always considered the baseline, and speedup is reported in relation to the transformed versions. Our open source benchmarks, and instructions for reproducing the results, are available online [8]. It should be noted that some results may be sensitive to the particular setup on which those experiments are run.

5.2 Results

In analyzing the results, we distinguish two cases: distributing and parallelizing loops with potentially unknown iterations, and loops with pre-determined iterations (typically **while** and **for** loops, respectively). The difficulty of parallelizing the former arises from the need to synchronize evaluation of the loop recurrence and termination condition. Improper synchronization results in overshooting the iterations [37], rendering such loops effectively sequential.

Loop fission addresses this challenge by recognizing independence between statements and producing parallelizable loops. Special care is needed when inserting parallelization directives for such loops. This remains a limitation of automated tools and is not natively supported by OpenMP. We resolved this issue by using the OpenMP single directive, to prevent overshooting the loop termination condition and need for synchronization between threads, enabling parallel execution by multiple threads on individual loop statements. The strategy is simple, implementable, and we show it to be effective. However, it is also upper-bounded in speedup potential by the number of parallelizable loops produced by the transformation. This is a syntactic constraint, rather than one based on number of available cores.

The results, presented in Table 3, show that our approach, paired with the described parallelization strategy, yields a gain relative to the number of independent parallelizable loops in the transformed benchmark. We observe this e.g., for benchmarks `bicg`, `gesummv`, and `mvt`, as presented in Fig. 6. We also confirm that ROSE's approach did not transform these loops, and report no gain for the alternative approach.

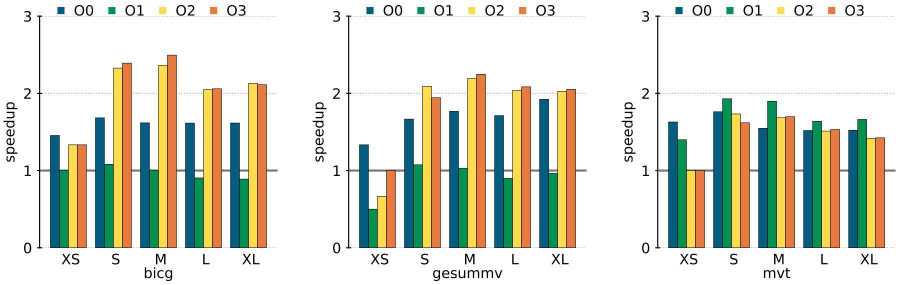


Fig. 6. Speedup of selected benchmarks implemented using `while` loops. Note the influence of various compiler optimization levels, `-O0` to `-O3` on each problem, and how parallelization overhead tends to decrease as input data size grows from `MINI` to `EXTRALARGE`. The gain is lower for `mvt` because it assumes fissioned form in the original benchmark. `bicg` and `gesummv` obtain higher gain from applied loop distribution.

Comparison with ROSE The remaining benchmarks, with known iteration spaces, can be transformed by both evaluated loop fission techniques: ours and ROSE's LoopProcessor. In terms of transformation results, we observed relatively similar results for both techniques. We discovered one interesting transformation difference, with benchmark `gemm`, which ROSE handles differently from our technique.

After transformation, the program must be parallelized by inserting OpenMP directives. This parallelization step can be fully automatic and performed with e.g., ROSE or `Clava`, demonstrating that pipelining the transformed programs is feasible. For evaluations, we used manual parallelization for our technique and automatic approach for ROSE. However, we also noted that the automatic insertion of parallelization directives yielded, in some cases, suboptimal choices, such as parallelization of loop nests. This added unnecessary overhead to execution time, and negatively impacted the results obtained for ROSE, e.g., for benchmarks `fdtd-2d` and `gemm`, as observable in the results. It is possible this issue could be mitigated by providing annotations and more detailed instructions for applying the parallelization directives. In other experiments with alternative parallelization tools [7, Sect. 4.3], we have been successful at finding optimal parallelization directives automatically, and therefore conclude it is achievable. We again refer to Table 3 for a detailed presentation of the experimental evaluation results.

Benchmark		-O0		-O1		-O2		-O3	
Name	Size	ours	rose	ours	rose	ours	rose	ours	rose
3mm	XS	2.71	0.07	2.26	0.02	1.71	0.02	1.73	0.01
	S	2.80	0.22	3.78	0.09	3.49	0.05	3.35	0.05
	M	2.20	0.46	3.44	0.27	3.08	0.13	3.05	0.13
	L	2.85	1.92	3.11	1.16	2.89	0.66	2.97	0.66
	XL	2.16	2.31	3.13	1.83	2.24	1.05	2.25	1.04
bicg	XS	1.45	0.96	1.00	1.00	1.33	1.00	1.33	1.00
	S	1.68	0.98	1.08	1.00	2.33	1.01	2.39	1.02
	M	1.62	0.97	1.00	0.98	2.36	0.96	2.50	1.00
	L	1.61	0.96	0.90	0.94	2.05	0.95	2.06	0.95
	XL	1.62	0.96	0.89	0.95	2.13	0.93	2.11	0.94
colormap	XS	2.14	1.01	1.50	1.02	1.54	1.04	1.52	1.01
	S	2.08	0.97	1.57	1.00	1.54	1.02	1.43	0.99
	M	1.98	0.95	1.46	0.96	1.49	0.98	1.19	1.00
	L	1.93	1.03	1.42	0.98	1.44	0.98	1.20	1.01
	XL	1.82	1.00	1.53	0.97	1.55	0.99	1.16	1.00
conjgrad	XS	2.43	1.45	1.82	0.69	2.77	0.65	2.50	0.52
	S	2.50	2.39	1.91	2.03	2.84	1.88	2.96	1.65
	M	2.56	2.58	1.94	2.66	2.93	2.44	3.20	2.33
	L	2.38	2.62	1.73	2.96	2.92	2.92	3.24	2.91
	XL	2.29	2.61	1.59	2.55	2.72	2.57	2.99	2.39
cp50	XS	1.90	0.97	1.97	1.00	2.18	1.01	2.09	1.01
	S	1.94	0.95	2.00	1.02	2.08	1.00	2.07	1.00
	M	1.89	0.98	1.76	0.97	1.83	0.99	1.82	0.98
	L	1.74	0.98	1.49	0.96	1.51	0.96	1.50	0.96
	XL	1.63	0.99	1.16	0.96	1.07	0.98	1.11	0.96
deriche	XS	2.00	0.90	1.93	0.51	2.18	0.53	2.11	0.51
	S	2.30	1.49	2.16	1.05	2.17	1.04	2.14	1.03
	M	2.68	2.35	2.88	2.20	2.68	2.22	2.72	2.20
	L	1.79	1.75	2.08	2.03	2.05	2.05	2.07	2.04
	XL	1.12	1.12	1.65	1.61	1.67	1.67	1.60	1.64

Benchmark		-O0		-O1		-O2		-O3	
Name	Size	ours	rose	ours	rose	ours	rose	ours	rose
fdtd-2d	XS	2.34	0.27	1.48	0.05	1.81	0.06	1.15	0.03
	S	2.57	0.59	2.68	0.15	3.12	0.17	2.47	0.09
	M	2.23	0.82	2.01	0.29	2.47	0.30	2.60	0.24
	L	2.15	1.20	1.89	0.65	1.98	0.61	2.16	0.71
	XL	2.17	1.38	1.47	0.79	1.50	0.73	1.68	0.86
gemm	XS	2.73	0.09	2.33	0.02	2.43	0.02	1.20	0.01
	S	2.87	0.21	3.98	0.05	3.09	0.04	3.01	0.02
	M	2.57	0.56	3.42	0.12	3.40	0.12	2.73	0.05
	L	2.44	1.50	1.79	0.35	1.87	0.36	2.20	0.25
	XL	2.44	1.95	1.85	0.60	1.85	0.70	1.96	0.50
gesummv	XS	1.33	1.00	0.50	0.67	0.67	0.67	1.00	1.00
	S	1.67	0.95	1.08	1.03	2.09	1.03	1.94	1.01
	M	1.77	0.98	1.03	1.00	2.19	1.00	2.25	1.00
	L	1.71	0.94	0.90	0.93	2.04	0.93	2.08	0.97
	XL	1.92	0.98	0.96	0.98	2.03	0.99	2.05	0.98
mvt	XS	1.63	1.00	1.40	0.88	1.00	1.00	1.00	1.00
	S	1.76	1.01	1.93	1.01	1.73	1.02	1.62	1.00
	M	1.55	0.96	1.90	1.00	1.69	1.02	1.70	1.03
	L	1.52	0.98	1.64	0.97	1.51	0.98	1.53	1.00
	XL	1.52	0.98	1.66	0.99	1.42	1.00	1.42	1.00
remap	XS	1.43	0.97	0.54	1.00	0.54	1.00	0.64	1.00
	S	2.07	0.94	1.20	1.02	1.13	1.03	1.19	1.01
	M	2.43	0.99	3.13	0.96	3.36	0.98	2.89	0.97
	L	2.09	1.00	1.34	0.97	1.54	1.02	1.74	1.00
	XL	2.11	1.00	1.28	0.99	1.52	0.99	1.57	1.00
tblshft	XS	3.19	3.27	2.70	2.65	2.68	2.73	2.82	2.82
	S	3.37	3.45	2.82	2.84	2.89	2.86	3.05	3.08
	M	3.31	3.62	2.93	3.00	2.79	2.85	3.21	3.19
	L	3.05	3.40	2.17	2.32	2.38	2.32	2.40	2.39
	XL	3.08	3.48	1.91	1.85	1.64	1.69	1.96	1.96

Table 3. Speedup comparison between original sequential and transformed parallel benchmarks, comparing our loop fission technique with ROSE Compiler, for various data sizes and compiler optimization levels. We note that the problems containing only **`while`** loop (in **bold**) are not transformed by ROSE and therefore report no gain. The other results vary depending on parallelization strategy, but as noted with e.g., problems **`conjgrad`** and **`tblshft`**, we obtain similar speedup for both fission strategies when automatic parallelization yields optimal OpenMP directives.

Benchmark	Description	**`for`** loop	**`while`** loop	Source
3mm	3D matrix multiplication	✓		PolyBench/C
bicg	BiCG sub kernel of BiCGStab linear solver		✓	PolyBench/C
colormap	TIFF image conversion of photometric palette		✓	MiBench
conjgrad	Conjugate gradient routine	✓		NAS-CG
cp50	Ghostscript/CP50 color print routine	✓	✓	MiBench
deriche	Edge detection filter	✓		PolyBench/C
fdtd-2d	2-D finite different time domain kernel	✓		PolyBench/C
gemm	Matrix-multiply C=alpha.A.B+beta.C	✓		PolyBench/C
gesummv	Scalar, vector and matrix multiplication		✓	PolyBench/C
mvt	Matrix vector product and transpose		✓	PolyBench/C
remap	4D matrix memory remapping	✓		NAS-UA
tblshift	TIFF PixarLog compression main table bit shift	✓	✓	MiBench

Table 4. Descriptions of evaluated parallel benchmarks.

6 Conclusion

This work is only the first step in a very exciting direction. “Ordinary code”, and not only code that was specifically written for e.g., scientific calculation or other resource-demanding operations, should be executed in parallel to leverage our modern architectures. As a consequence, the much larger codebase concerned with parallelization is much less predictable and offers more diverse loop structures. Focusing on resource-demanding programs led previous efforts not only to focus on predictable loop structures, but to completely ignore other non-canonical loops. Our effort, based on an original dependency analysis, leads to re-integrate such loops in the realm of parallel optimization. This alone, in our opinion, justifies further investigation in integrating our algorithm into specialized tools.

As presented in Fig. 6, our experimental results offer some variability, but they need to be put in context: loop distribution is often only *the first step* in the optimization pipeline. Loops that have been split can then be vectorized, blocked, unrolled, etc., providing additional gain in terms of speed. Exactly as for loop fusion [31], a more global treatment of loops is needed to strike the right balance and find the optimum code transformation. Such a journey will be demanding and complex, but we believe this work enables it by reintegrating *all* loops in the realm of parallel optimization.

Acknowledgments The authors wish to express their gratitude to João Bispo for explaining how to integrate `AutoPar-Clava` in the first version of their benchmark, to Assya Sellak for her contribution to the first steps of this work, and to the reviewers for their insightful comments.

References

1. Abel, A., Altenkirch, T.: A predicative analysis of structural recursion. Journal of Functional Programming **12**(1), 1–41 (2002). `https://doi.org/10.1017/S0956796801004191`
2. Abu-Sufah, Kuck, Lawrie: On the performance enhancement of paging systems through program analysis and transformations. IEE Transactions on Computers **C-30**(5), 341–356 (1981). `https://doi.org/10.1109/TC.1981.1675792`
3. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison Wesley (Aug 2006)
4. Amini, M.: Source-to-Source Automatic Program Transformations for GPU-like Hardware Accelerators. Theses, Ecole Nationale Supérieure des Mines de Paris (Dec 2012), `https://pastel.archives-ouvertes.fr/pastel-00958033`
5. Amini, M., Creusillet, B., Even, S., Keryell, R., Goubier, O., Guelton, S., Mcmahon, J.O., Pasquier, F.X., Péan, G., Villalon, P.: Par4All: From Convex Array Regions to Heterogeneous Computing. In: IMPACT 2012 : Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012. Paris, France (Jan 2012), `https://hal-mines-paristech.archives-ouvertes.fr/hal-00744733`
6. Arabnejad, H., Bispo, J., Cardoso, J.M.P., Barbosa, J.G.: Source-to-source compilation targeting openmp-based automatic parallelization of C applications. The Journal of Supercomputing **76**(9), 6753–6785 (Sep 2020). `https://doi.org/10.1007/s11227-019-03109-9`
7. Aubert, C., Rubiano, T., Rusch, N., Seiller, T.: A Novel Loop Fission Technique Inspired by Implicit Computational Complexity (May 2022), `https://hal.archives-ouvertes.fr/hal-03669387v1`, draft
8. Aubert, C., Rubiano, T., Rusch, N., Seiller, T.: Loop fission benchmarks (Sep 2022). `https://doi.org/10.5281/zenodo.7080145`, `https://github.com/statycc/loop-fission`
9. Aubert, C., Rubiano, T., Rusch, N., Seiller, T.: mwp-analysis improvement and implementation: Realizing implicit computational complexity. In: Felty, A.P. (ed.) 7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022). Leibniz International Proceedings in Informatics, vol. 228, pp. 26:1–26:23. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2022). `https://doi.org/10.4230/LIPIcs.FSCD.2022.26`
10. Aubert, C., Rubiano, T., Rusch, N., Seiller, T.: pymwp: MWP analysis in Python (Sep 2022), `https://github.com/statycc/pymwp/`
11. Bae, H., Mustafa, D., Lee, J., Aurangzeb, Lin, H., Dave, C., Eigenmann, R., Midkiff, S.P.: The cetus source-to-source compiler infrastructure: Overview and evaluation. Int. J. Parallel Program. **41**(6), 753–767 (2013). `https://doi.org/10.1007/s10766-012-0211-z`
12. Baier, C., Katoen, J., Larsen, K.: Principles of Model Checking. MIT Press (2008)
13. Benabderrahmane, M., Pouchet, L., Cohen, A., Bastoul, C.: The polyhedral model is more widely applicable than you think. In: Gupta, R. (ed.) Compiler Construction, 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6011, pp. 283–303. Springer (2010). `https://doi.org/10.1007/978-3-642-11970-5_16`
14. Bertolacci, I.J., Strout, M.M., de Supinski, B.R., Scogland, T.R.W., Davis, E.C., Olschanowsky, C.: Extending openmp to facilitate loop optimization. In: de Supinski, B.R., Valero-Lara, P., Martorell, X., Bellido, S.M., Labarta, J. (eds.) Evolving

OpenMP for Evolving Architectures - 14th International Workshop on OpenMP, IWOMP 2018, Barcelona, Spain, September 26-28, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11128, pp. 53–65. Springer (2018). https://doi.org/10.1007/978-3-319-98521-3_4

15. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. SIGPLAN Not. **43**(6), 101–113 (jun 2008). https://doi.org/10.1145/1379022.1375595
16. Chung, F.R.K.: On the coverings of graphs. Discrete Mathematics **30**(2), 89–93 (1980). https://doi.org/10.1016/0012-365X(80)90109-0
17. Dave, C., Bae, H., Min, S., Lee, S., Eigenmann, R., Midkiff, S.P.: Cetus: A source-to-source compiler infrastructure for multicores. Computer **42**(11), 36–42 (2009). https://doi.org/10.1109/MC.2009.385
18. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems **9**(3), 319–349 (jul 1987). https://doi.org/10.1145/24039.24041
19. gcc.gnu.org git - gcc.git/blob - gcc/tree-loop-distribution.c, https://gcc.gnu.org/git/?p=gcc.git;a=blob;f=gcc/tree-loop-distribution.c;h=65aa1df4abae2c6acf40299f710bc62ee6bacc07;hb=HEAD#l39
20. Grosser, T.: Enabling Polyhedral Optimizations in LLVM. Master's thesis, Universität Passau (4 2011), https://polly.llvm.org/publications/grosser-diploma-thesis.pdf
21. Holewinski, J., Ramamurthi, R., Ravishankar, M., Fauzia, N., Pouchet, L.N., Rountev, A., Sadayappan, P.: Dynamic trace-based analysis of vectorization potential of applications. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 371–382. PLDI '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2254064.2254108
22. Intel: oneTBB documentation (2022), https://oneapi-src.github.io/oneTBB/
23. Intel Corporation: Intel C++ Compiler Classic Developer Guide and Reference, https://www.intel.com/content/dam/develop/external/us/en/documents/cpp_compiler_classic.pdf
24. Karp, R.M., Miller, R.E., Winograd, S.: The organization of computations for uniform recurrence equations. Journal of the ACM **14**(3), 563–590 (1967). https://doi.org/10.1145/321406.321418
25. Klemm, M., de Supinski, B.R. (eds.): OpenMP Application Programming Interface Specification Version 5.2. OpenMP Architecture Review Board (Nov 2021), https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf
26. Kristiansen, L., Jones, N.D.: The flow of data and the complexity of algorithms. In: Cooper, S.B., Löwe, B., Torenvliet, L. (eds.) New Computational Paradigms, First Conference on Computability in Europe, CiE 2005, Amsterdam, The Netherlands, June 8-12, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3526, pp. 263–274. Springer (2005). https://doi.org/10.1007/11494645_33
27. Laird, J., Manzonetto, G., McCusker, G., Pagani, M.: Weighted relational models of typed lambda-calculi. In: LICS. pp. 301–310. IEEE Computer Society (2013). https://doi.org/10.1109/LICS.2013.36
28. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA. pp. 75–88. IEEE Computer Society (2004). https://doi.org/10.1109/CGO.2004.1281665, https://ieeexplore.ieee.org/xpl/conhome/9012/proceeding

29. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: Hankin, C., Schmidt, D. (eds.) Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001. pp. 81–92. ACM (2001). https://doi.org/10.1145/360204.360210
30. [loopfission]: Loop fission interference graph (fig), https://reviews.llvm.org/D73801
31. Mehta, S., Lin, P., Yew, P.: Revisiting loop fusion in the polyhedral framework. In: Moreira, J.E., Larus, J.R. (eds.) ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014. pp. 233–246. ACM (2014). https://doi.org/10.1145/2555243.2555250
32. microsoft: Parallel patterns library (ppl) (2021), https://docs.microsoft.com/en-us/cpp/parallel/concrt/parallel-patterns-library-ppl?view=msvc-170
33. Moyen, J., Rubiano, T., Seiller, T.: Loop quasi-invariant chunk detection. In: D'Souza, D., Kumar, K.N. (eds.) Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10482. Springer (2017). https://doi.org/10.1007/978-3-319-68167-2_7
34. Palkowski, M., Klimek, T., Bielecki, W.: TRACO: an automatic loop nest parallelizer for numerical applications. In: Ganzha, M., Maciaszek, L.A., Paprzycki, M. (eds.) 2015 Federated Conference on Computer Science and Information Systems, FedCSIS 2015, Lódz, Poland, September 13-16, 2015. Annals of Computer Science and Information Systems, vol. 5, pp. 681–686. IEEE (2015). https://doi.org/10.15439/2015F34
35. Prema, S., Nasre, R., Jehadeesan, R., Panigrahi, B.: A study on popular auto-parallelization frameworks. Concurrency and Computation: Practice and Experience **31**(17), e5168 (Feb 2019). https://doi.org/10.1002/cpe.5168
36. Quinlan, D., Liao, C., Panas, T., Matzke, R., Schordan, M., Vuduc, R., , Yi, Q.: Rose user manual: A tool for building source-to-source translators draft user manual (version 0.9.11.115), http://rosecompiler.org/uploads/ROSE-UserManual.pdf
37. Rauchwerger, L., Padua, D.A.: Parallelizing while loops for multiprocessor systems. In: Proceedings of the 9th International Symposium on Parallel Processing. p. 347–356. IPPS '95, IEEE Computer Society, USA (1995)
38. Seiller, T.: Interaction graphs: Full linear logic. In: Grohe, M., Koskinen, E., Shankar, N. (eds.) Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016. pp. 427–436. ACM (2016). https://doi.org/10.1145/2933575.2934568
39. Vitorović, A., Tomašević, M.V., Milutinović, V.M.: Manual parallelization versus state-of-the-art parallelization techniques. In: Hurson, A. (ed.) Advances in Computers, vol. 92, pp. 203–251. Elsevier (2014). https://doi.org/10.1016/B978-0-12-420232-0.00005-2

I.3 pymwp: A Static Analyzer Determining Polynomial Growth Bounds (Tool User Guide)

pymwp: A Static Analyzer Determining Polynomial Growth Bounds

Tool User Guide

Clément Aubert Thomas Rubiano Neea Rusch Thomas Seiller

May 10, 2023

Contents

1 Introduction

pymwp (“pai m-w-p”) is a tool for automatically performing static analysis on programs written in a subset of the C language. It analyzes resource usage and determines if program variables’ growth rates are no more than polynomially related to their inputs sizes.

The theoretical foundations are described in paper the “*mwp-Analysis Improvement and Implementation: Realizing Implicit Computational Complexity*” (cf. “Learn More” for additional references and links). The technique is generic and applicable to any imperative language. pymwp is an implementation demonstrating this technique concretely on C programs. The technique is originally inspired by “*A Flow Calculus of mwp-Bounds for Complexity Analysis*”.

This guide explains pymwp usage and behavior through several examples.

1.1 Property of Interest

For an imperative program, the goal is to discover a polynomially bounded data-flow relation, between the *initial values* of the variables (denoted `X1,…,Xn`) and its *final values* (denoted `X1',…,Xn'`).

For a program written in C language, this property can be presented as follows.

```
void main(int X1, int X2, int X3){
   // initial values  ↑

   /*
    * various commands involving
    * variables X1, X2, X3
    */

   // X1', X2', X3' (final values)
}
```

Question: $\forall n$, is $\texttt{X}_n \rightsquigarrow \texttt{X}'_n$ polynomially bounded in inputs?

We answer this question using the mwp-flow analysis, implemented in the pymwp static analyzer.

1.2 What mwp-Flow Analysis Computes

The mwp-flow analysis works to establish a polynomial growth bound for input variables by applying inference rules to program’s commands.

Internally, the analysis tracks *coefficients* representing *dependencies* between program’s variables. These coefficients (or “flows”) are $0, m, w, p$ and ∞. They characterize how data flows between variables.

- 0 — no dependency
- m — maximal (of linear)
- w — weak polynomial
- p — polynomial
- ∞ — infinite

with ordering: $0 < m < w < p < \infty$. The analysis name also comes from these coefficients.

After analysis, two outcomes are possible. (A) The program’s variables values can be bounded by a polynomial in the input’s values, or (B) the analysis determines it is impossible to establish such a bound. Due to non-determinism, many derivation paths need to be explored to determine this result.

The analysis succeeds if – for some derivation – no pair of variables is characterized by ∞-flow. That is, obtaining an ∞-free derivation implies existence of a polynomial growth bound; i.e., the program has the property of interest, or we can say that the program is *derivable*. The soundness theorem of the mwp-calculus guarantees that if such derivation exists, the program variables' value growth is polynomially bounded in inputs.

Program fails the analysis if every derivation contains an ∞ coefficient. Then it is not possible to establish polynomial growth bound. For these programs, pymwp reports an ∞-result.

1.3 Interpreting mwp-Bounds

If the analysis is successful, i.e., polynomial growth bound exists, it is represented using *mwp-bounds*.

An mwp-bound is a number theoretic expression of form: $\max(\vec{x}, poly_1(\vec{y})) + poly_2(\vec{z})$.

Disjoint variable lists $\vec{x}$, $\vec{y}$ and $\vec{z}$ capture dependencies of an input variable. Dependencies characterized by m-flow are in $\vec{x}$, w-flow in $\vec{y}$, and p-flow in $\vec{z}$. The polynomials $poly_1$ and $poly_2$ are built up from constants, variables, and operators $+$ and $\times$. Each variable list may be empty and $poly_1$ and $poly_2$ may not be present.

For multiple input variables, the result is a conjunction of mwp-bounds, one for each input variable.

Example 1. Assume program has one input variable named `X`, and we have obtained a bound: $\mathtt{X}' \leq \mathtt{X}$. The bound expression means the final value $\mathtt{X}'$ depends only on its own initial value `X`.

Example 2. Assume program has two inputs, `X` and `Y`, and we obtained a bound: $\mathtt{X}' \leq \mathtt{X} \wedge \mathtt{Y}' \leq \max(\mathtt{X}, 0) + \mathtt{Y}$.

- Final value $\mathtt{X}'$ depends on its own initial value `X`.
- Final value $\mathtt{Y}'$ depends on initial values of inputs `X` and `Y`.
- The bound expression can be simplified to $\mathtt{X}' \leq \mathtt{X} \wedge \mathtt{Y}' \leq \mathtt{X} + \mathtt{Y}$.

2 Installation

57 This section explains how to get started using pymwp. We recommend installing pymwp locally for an interactive tutorial experience.

Check environment requirements

pymwp requires Python runtime 3.7 – 3.11 (current latest).

To check currently installed version, run:

```
python3 --version
```

On systems that default to Python 3 runtime, use `python` instead of `python3`.

Instructions for installing Python for different operating systems can be found at python.org ↗.

Install pymwp

Install pymwp from Python Package Index:

```
pip3 install pymwp==0.4.2
```

Download examples

Download a set of examples.

```
wget https://github.com/statycc/pymwp/releases/download/0.4.2/examples.zip
```

Alternatively, download the set of examples using a web browser:

https://github.com/statycc/pymwp/releases/download/0.4.2/examples.zip

Unzip the examples

Unzip `examples.zip` using your preferred approach.

This completes the setup.

3 Examples

58 We now demonstrate use of pymwp on several examples. To ease the presentation, we will use multiple command line arguments.

- `--fin` — always run analysis to completion (even on failure)
- `--info` — reduces amount of terminal output to info level
- `--no_time` — omits timestamps from output log

For a complete list of available command arguments, run:

```
pymwp --help
```

3.1 Binary Assignment

This example shows that assigning a compounded expression to a variable results in correct analysis.

Analyzed Program: assign_expression.c

```
int foo(int y1, int y2){
    y2 = y1 + y1;
}
```

It is straightforward to observe that this program has a polynomial growth bound. The precise value of that bound is $\texttt{y1}' = \texttt{y1} \wedge \texttt{y2}' \leq 2 * \texttt{y1}$. Although the program is simple, it is interesting because binary operations introduce complexity in program analysis.

CLI Command

The current working directory should be the location of unzipped examples from Installation Step 4.

```
pymwp basics/assign_expression.c --fin --info --no_time
```

Output:

```
INFO (result): Bound: y1' ≤ y1 ∧ y2' ≤ y1
INFO (result): Bounds: 3
INFO (result): Total time: 0.0 s (0 ms)
INFO (file_io): saved result in output/assign_expression.json
```

Discussion

The analysis correctly assigns a polynomial bound to the program. The bound obtained by the analyzer is $\texttt{y1}' \leq \texttt{y1} \wedge \texttt{y2}' \leq \texttt{y1}$. Comparing to the precise value determined earlier, this bound is correct, because we omit constants in the analysis results.

Due to non-determinism, the analyzer finds three different derivations that yield a bound. From the `.json` file, that captures the analysis result in more technical detail, it is possible to determine these three bounds are:

- $\texttt{y1}' \leq \texttt{y1} \wedge \texttt{y2}' \leq$ `max(0,0) + y1`
- $\texttt{y1}' \leq \texttt{y1} \wedge \texttt{y2}' \leq$ `max(0,0) + y1`
- $\texttt{y1}' \leq \texttt{y1} \wedge \texttt{y2}' \leq$ `max(0,y1) + 0`

They all simplify to $\texttt{y1}' \leq \texttt{y1} \wedge \texttt{y2}' \leq \texttt{y1}$. This concludes the obtained result matches the expected result.

3.2 Exponential Program

A program computing the exponentiation returns an infinite coefficient, no matter the derivation strategy chosen.

Analyzed Program: exponent_2.c

```
int foo(int base, int exp, int i, int result){
    while (i < exp){
        result = result * base;
        i = i + 1;
    }
}
```

This program's variable `result` grows exponentially. It is impossible to find a polynomial growth bound, and the analysis is expected to report ∞-result. This example demonstrates how pymwp arrives to that conclusion.

CLI Command

```
pymwp infinite/exponent_2.c --fin --info --no_time
```

Output:

```
INFO (result): foo is infinite
INFO (result): Possibly problematic flows:
INFO (result): base → result ‖ exp → result ‖ i → result ‖ result → result
INFO (result): Total time: 0.0 s (2 ms)
INFO (file_io): saved result in output/exponent_2.json
```

Discussion

The output shows that the analyzer correctly detects that no bound can be established, and we obtain ∞-result. The output also gives a list of problematic flows. This list indicates all variable pairs, that along some derivation paths, cause ∞ coefficients to occur. The arrow direction means data flows from `source` → `target`. We can see the problem with this program is the data flowing to `result` variable. This clearly indicates the origin of the problem, and allows programmer to determine if the issue can be repaired, to improve program's complexity properties.

3.3 While Analysis

A program that shows infinite coefficients for some derivations.

Analyzed Program: notinfinite_3.c

```
int foo(int X0, int X1, int X2, int X3){
    if (X1 == 1){
        X1 = X2+X1;
        X2 = X3+X2;
    }
    while(X0<10){
        X0 = X1+X2;
    }
}
```

This program contains decision logic, iteration, and multiple variables. Determining if a polynomial growth bound exists is not immediate by inspection. It is therefore an interesting candidate for analysis with pymwp!

CLI Command

```
pymwp not_infinite/notinfinite_3.c --fin --info --no_time
```

Output:

```
INFO (result): Bound: X0' ≤ max(X0,X1)+X2*X3 ∧ X1' ≤ X1+X2 ∧ X2' ≤ X2+X3 ∧ X3' ≤ X3
INFO (result): Bounds: 9
INFO (result): Total time: 0.0 s (3 ms)
INFO (file_io): saved result in output/notinfinite_3.json
```

Discussion

Compared to previous examples, the analysis is now getting more complicated. We can observe this in the number of discovered bounds and the form of the bound expression. The number of times the loop iterates, or which branch of the `if` statement is taken, is not a barrier to determining the result.

From the bound expression, we can determine the following.

- `X0` has the most complicated dependency relation. Its mwp-bound combines the impact of the `if` statement, the `while` loop, and the chance that the loop may not execute.
- `X1` and `X2` have fairly simple growth dependencies, originating from the `if` statement.
- `X3` is the simplest case – it never changes. Therefore, it only depends on itself.

Overall, the analysis concludes the program has a polynomial growth bound.

3.4 Infinite Program

A program that shows infinite coefficients for all choices.

Analyzed Program: infinite_3.c

```
int foo(int X1, int X2, int X3){
    if (X1 == 1){
        X1 = X2+X1;
        X2 = X3+X2;
    }
    while(X1<10){
        X1 = X2+X1;
    }
}
```

If you studied the previous example carefully, you might notice that this example is *very similar*. There is a subtle differences: variable X0 has been removed and its usages changed to X1. This example demonstrates how this seemingly small change impacts the analysis result.

CLI Command

```
pymwp infinite/infinite_3.c --fin --info --no_time
```

Output:

```
INFO (result): foo is infinite
INFO (result): Possibly problematic flows:
INFO (result): X1 → X1 ‖ X2 → X1 ‖ X3 → X1
INFO (result): Total time: 0.0 s (2 ms)
INFO (file_io): saved result in output/infinite_3.json
```

Discussion

We can observe the result is ∞. Thus, even a small change can change the analysis result entirely.

The output reveals the problem arises from how data flows from source variables X1, X2, and X3, to target variable X1. Observe that even though there is no direct assignment from X3 to X1, the analysis correctly identifies this dependency relation, that occurs through X2.

From the output, we have identified the point and source of failure. Conversely, we know other variable pairs are not problematic. By focusing on how to avoid "too strong" dependencies targeting variable X1, programmer may be able to refactor and improve the program's complexity properties.

3.5 Challenge Example

Try to guess the analysis outcome before determining the result with pymwp.

Analyzed Program

```
int foo(int X0, int X1, int X2){
    if (X0) {
      X2 = X0 + X1;
    }
    else{
      X2 = X2 + X1;
    }
    X0 = X2 + X1;
    X1 = X0 + X2;
    while(X2){
      X2 = X1 + X0;
    }
}
```

After seeing the various preceding examples – with and without polynomial bounds – we present the following challenge. By inspection, try to determine if this program is polynomially bounded w.r.t. its input values.

It is unknown which `if` branch will be taken, and whether the `while` loop will terminate, but this is not a problem for determining the result.

CLI Command

```
pymwp other/dense_loop.c --fin --no_time --info
```

Output:

```
INFO (result): Bound: X0' ≤ max(X0,X2)+X1 ∧ X1' ≤ X0*X1*X2 ∧ X2' ≤ max(X0,X2)+X1
INFO (result): Bounds: 81
INFO (result): Total time: 0.0 s (29 ms)
INFO (file_io): saved result in output/dense_loop.json
```

Discussion

Even with just 3 variables we can see—in the obtained bound expression and the number of bounds—that this is a complicated derivation problem. The analyzer determines the program has a polynomial growth bound. Let us reason informally and intuitively why this obtained result is correct.

We can observe in the bound expression, that all three variables have complicated dependencies on one another; this corresponds to what is also observable in the input program.

Regarding variables `X0` and `X1`, observe there is no command, with either as a target variable, that would give rise to exponential value growth (need iteration). Therefore, they must have polynomial growth bounds.

Variable `X2` is more complicated. The program has a `while` loop performing assignments to `X2` (potentially problematic), and the `while` loop may or may not execute.

- Case 1: loop condition is initially false. Then, final value of `X2` depends on the `if` statement, and in either branch, it will have polynomially bounded growth.

- Case 2: loop condition true – at least one iteration will occur. The program iteratively assigns values
  to `X2` inside the `while` loop. However, notice the command is loop invariant. No matter how many
  64 times the loop iterates, the final value of `X2` is `X1` + `X0`. We already know those two variables have
  polynomial growth bounds. Therefore, `X2` also grows polynomially w.r.t. its input values.

This reasoning concurs with the result determined by pymwp.

---

4 Learn More

This guide has offered only a *very short* introduction to mwp-analysis and pymwp.
If you wish to explore more, have a look at:

- "*mwp-Analysis Improvement and Implementation: Realizing Implicit Computational Complexity*"[1]
  Describes theoretical foundations behind pymwp.
- "*A Flow Calculus of mwp-Bounds for Complexity Analysis*"[2]
  Original mwp-flow analysis technique.
- Documentation[3] – Includes information about supported C language features, etc.
- Online demo[4] – Run pymwp online on more than 40 examples.
- Source code[5] – pymwp is open source.
- License[6] – pymwp is licensed under GPLv3.

[1] https://doi.org/10.4230/LIPIcs.FSCD.2022.26
[2] https://doi.org/10.1145/1555746.1555752
[3] https://statycc.github.io/pymwp
[4] https://statycc.github.io/pymwp/demo/
[5] https://github.com/statycc/pymwp
[6] https://github.com/statycc/pymwp/blob/main/LICENSE

I.4 Three Minute Thesis

The Three Minute Thesis (3MT) is an academic research communication competition. It celebrates the research conducted by graduate students, while challenging them to effectively explain the research—in spoken words in three minutes—in a language appropriate to a non-specialist audience. Competitors are expected to have one static illustrative slide. The 3MT competition has been organized annually at Augusta University since 2017. I participated in the competition in 2024 with the following presentation.

I.4.1 Smashing Software Bugs with Proofs

We live in a world where software is everywhere. It wasn't always so. When I was little, my parents had a push-button telephone and "updating a TV" meant buying a new one from a store. But we continue to experience fast exciting transformation with technology. However, the more software we come across, the more bugs we see. A bug is a fault in software that causes incorrect or unexpected behavior. Since we depend heavily on software, there is a high incentive to eliminate bugs. This is especially true for critical systems like cars and airplanes. Unfortunately, eliminating bugs is not easy. There are multiple challenges. Fundamentally, software is made of 0s and 1s. Even a simple computer has more ways to arrange the 0s and 1s than there are atoms in the universe. It is impossible to test everything. Software engineers are under pressure to publish releases, so quality control steps get overlooked.

I research ways to reduce these challenges. I want to empower software engineers to build high quality correct applications. The strategy is to build specialized software, called verifiers, to check other software. But if we use software to verify software, what is there to say the verifier is correct? Answering this question is my research topic. See, I have a technique for software verification. It allows me to check that memory usage doesn't get out of control. But to trust this technique, I have to prove it gives correct results. I do this by building mathematical proofs. But instead of pen and paper, I use a computer. First, I define a model. Then, I construct proofs about the model. I put those proofs together, like lego blocks, to form an even big proof. I can also check the proofs with a computer. These kind of mechanized proofs are the most rigorous ways we know for asserting correctness.

If I can complete my proof, it shows that achieving strong correctness guarantees is possible. It would enable building verifiers we can trust, and those would help software engineers eliminate bugs that impact us all. But there is an even more extended impact. Remember, everything in tech moves fast. Only this month we were introduced to the world's first AI software engineer. You see, in their infinite wisdom the software engineers decided that software engineers should be among the first jobs replaced by AI. Talk about job security! But all software, whether created by humans or AI, need quality control checks. If we replace the human, we only have an increasing urgency for the techniques I am researching.

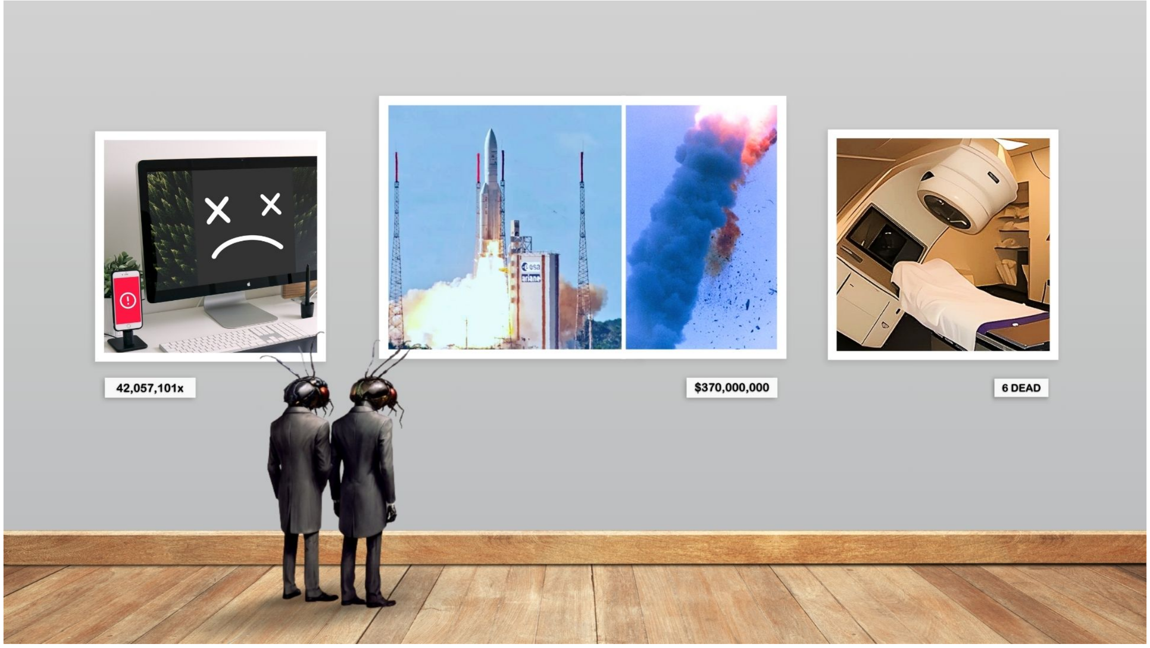


Figure 1: 3MT competition slide, “Smashing Software Bugs with Proofs”.