Distributing and Parallelizing Non-canonical Loops

Clément Aubert, Thomas Rubiano, Neea Rusch, Thomas Seiller

Distributing and Parallelizing Non-canonical Loops

Clément Aubert, Thomas Rubiano, Neea Rusch, Thomas Seiller

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.

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

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 research is supported by the Thomas Jefferson Fund 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".

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.

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

[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.

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.

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

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

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

Fig. 1. A simple imperative `while` language

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.

Definition 1. *Given an expression* `e`*, we define the variables occurring in* `e` *by:*

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

$$\mathrm{Occ}(\mathrm{val}) = \emptyset \qquad \mathrm{Occ}(\texttt{op(e}_1, \ldots, \texttt{e}_\texttt{n}\texttt{)}) = \mathrm{Occ}(\texttt{e}_1) \cup \cdots \cup \mathrm{Occ}(\texttt{e}_\texttt{n})$$

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

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.

[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.

C	Out(C)	In(C)	Occ(C) = Out(C) ∪ In(C)
$\mathtt{x} = \mathtt{e}$	$\mathtt{x}$	$\mathrm{Occ}(\mathtt{e})$	$\mathtt{x} \cup \mathrm{Occ}(\mathtt{e})$
$\mathtt{t}[\mathtt{e}_1] = \mathtt{e}_2$	$\mathtt{t}$	$\mathrm{Occ}(\mathtt{e}_1) \cup \mathrm{Occ}(\mathtt{e}_2)$	$\mathtt{t} \cup \mathrm{Occ}(\mathtt{e}_1) \cup \mathrm{Occ}(\mathtt{e}_2)$
$\mathtt{if}\ \mathtt{e}\ \mathtt{then}\ \mathtt{C}_1\ \mathtt{else}\ \mathtt{C}_2$	$\mathrm{Out}(\mathtt{C}_1) \cup \mathrm{Out}(\mathtt{C}_2)$	$\mathrm{Occ}(\mathtt{e}) \cup \mathrm{In}(\mathtt{C}_1) \cup \mathrm{In}(\mathtt{C}_2)$	$\mathrm{Occ}(\mathtt{e}) \cup \mathrm{Occ}(\mathtt{C}_1) \cup \mathrm{Occ}(\mathtt{C}_2)$
$\mathtt{while}\ \mathtt{e}\ \mathtt{do}\ \mathtt{C}$	$\mathrm{Out}(\mathtt{C})$	$\mathrm{Occ}(\mathtt{e}) \cup \mathrm{In}(\mathtt{C})$	$\mathrm{Occ}(\mathtt{e}) \cup \mathrm{Occ}(\mathtt{C})$
$\mathtt{use}(\mathtt{x}_1, \ldots, \mathtt{x}_n)$	$\mathtt{f}$	$\{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$	$\{\mathtt{x}_1, \ldots, \mathtt{x}_n, \mathtt{f}\}$
$\mathtt{skip}$	$\emptyset$	$\emptyset$	$\emptyset$
$\mathtt{C}_1; \mathtt{C}_2$	$\mathrm{Out}(\mathtt{C}_1) \cup \mathrm{Out}(\mathtt{C}_2)$	$\mathrm{In}(\mathtt{C}_1) \cup \mathrm{In}(\mathtt{C}_2)$	$\mathrm{Occ}(\mathtt{C}_1) \cup \mathrm{Occ}(\mathtt{C}_2)$

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

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]

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

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 $\mathtt{C}$ 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 $\mathrm{Occ}(\mathtt{C})$.

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 semi-ring $(\{0, 1, \infty\}, \max, \times)$ to represent dependencies: ∞ represents *dependence*, 1 represents *propagation*, and 0 represents *reinitialization*.

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

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

Definition 4 (Assignment). *Given an assignment* $\mathtt{C}$*, its* DFG *is given by:*

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

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, $\mathrm{In}(\mathtt{C})$ is exactly the set of variables that are source of a dependence arrow, while $\mathrm{Out}(\mathtt{C})$ is the set of variables that either are targets of dependence arrows or were reinitialized.

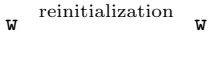
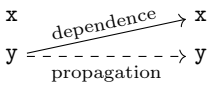
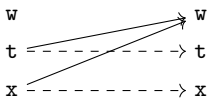
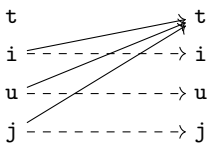
$\mathtt{C}$	$\mathrm{Out}(\mathtt{C})$, $\mathrm{In}(\mathtt{C})$	$\mathbb{M}(\mathtt{C})$ (as a graph)	$\mathbb{M}(\mathtt{C})$
$\mathtt{w} = 3$	$\mathrm{Out}(\mathtt{C}) = \{\mathtt{w}\}$ $\mathrm{In}(\mathtt{C}) = \emptyset$		$\begin{array}{c} \\ \mathtt{w} \\ \mathtt{w} \end{array}\begin{array}{c}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

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

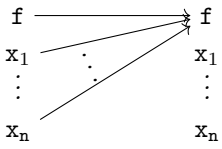
Note that we over-approximate arrays in two ways: the dependencies of the value at one index are the dependencies of the whole array, and the index at which the value is assigned is a dependence of the whole array (cf. the solid arrow from `i` to `t` in the last example of Fig. 2). This is however enough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

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

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

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



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* $\mathbb{M}(\texttt{C}_1; \ldots; \texttt{C}_n)$ *be* $\mathbb{M}(\texttt{C}_1) \times \cdots \times \mathbb{M}(\texttt{C}_n)$.

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 $\texttt{C}_1$ and $\texttt{C}_2$ are themselves already the result of compositions of assignments involving disjoint variables, and hence straightforward to compute.

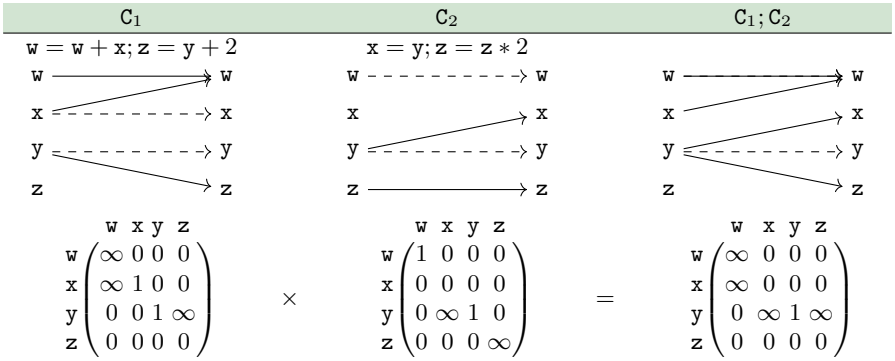


Fig. 3. Data-Flow Graph of Composition.

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

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

– 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		$E^t \times O$
		w x y z		w x y z
w ∞		$(\infty\ 0\ 0\ \infty)$	$(\mathrm{Out}(\mathtt{C}_1))$	w $\infty\ \infty\ 0\ \infty$
x ∞	$+$	$(0\ \infty\ 0\ \infty)$	$(\mathrm{Out}(\mathtt{C}_2))$	x $\infty\ \infty\ 0\ \infty$
y 0	$=$	$(\infty\ \infty\ 0\ \infty)$	$(\mathrm{Out}(\mathtt{C}_1;\mathtt{C}_2))$	y 0 0 0 0
z 0				z 0 0 0 0

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 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:

$$\mathbb{M}\left(\begin{array}{l}\texttt{if(w>x)}\\ \quad\texttt{then } \mathtt{w} = \mathtt{w} + \mathtt{x};\\ \qquad\quad \mathtt{z} = \mathtt{y} + 2\\ \quad\texttt{else } \mathtt{x} = \mathtt{y};\\ \qquad\quad \mathtt{z} = \mathtt{z} * 2\end{array}\right) = \begin{array}{c}\\ \mathtt{w}\\ \mathtt{x}\\ \mathtt{y}\\ \mathtt{z}\end{array}\!\!\begin{array}{c}\begin{array}{cccc}\mathtt{w} & \mathtt{x} & \mathtt{y} & \mathtt{z}\end{array}\\ \begin{pmatrix}\infty & 0 & 0 & 0\\ \infty & \boxed{1} & 0 & 0\\ 0 & 0 & 1 & \infty\\ 0 & 0 & 0 & 0\end{pmatrix}\end{array} + \begin{array}{c}\\ \mathtt{w}\\ \mathtt{x}\\ \mathtt{y}\\ \mathtt{z}\end{array}\!\!\begin{array}{c}\begin{array}{cccc}\mathtt{w} & \mathtt{x} & \mathtt{y} & \mathtt{z}\end{array}\\ \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & \boxed{0} & 0 & 0\\ 0 & \infty & 1 & 0\\ 0 & 0 & 0 & \infty\end{pmatrix}\end{array} + \begin{array}{c}\\ \mathtt{w}\\ \mathtt{x}\\ \mathtt{y}\\ \mathtt{z}\end{array}\!\!\begin{array}{c}\begin{array}{cccc}\mathtt{w} & \mathtt{x} & \mathtt{y} & \mathtt{z}\end{array}\\ \begin{pmatrix}\infty & \infty & \textcircled{0} & \infty\\ \infty & \boxed{\infty} & 0 & \infty\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\end{pmatrix}\end{array}$$

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* $\mathbb{M}(\texttt{while e do C})$ *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 $\begin{array}{c} \\ \mathtt{t} \\ \mathtt{i} \\ \mathtt{j} \\ \mathtt{s1} \\ \mathtt{s2} \end{array} \begin{array}{c} \begin{array}{ccccc} \mathtt{t} & \mathtt{i} & \mathtt{j} & \mathtt{s1} & \mathtt{s2} \end{array} \\ \begin{pmatrix} 1 & \infty & 0 & \infty & \infty \\ 0 & \infty & 0 & \infty & \infty \\ 0 & \infty & 1 & \infty & \infty \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$. 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* $\mathtt{W} \coloneqq \texttt{while e do C}$*, the* update commands $\mathtt{C_u}$ *are the commands in* `C` *such that* $\mathbb{M}(\mathtt{W})(\mathtt{x},\mathtt{y}) = \infty$ *for* $\mathtt{x} \in \mathrm{Out}(\mathtt{C_u})$ *and* $\mathtt{y} \in \mathrm{Occ}(\mathtt{e})$.

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.

[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.

Definition 12 (Dependence graph). *The* dependence graph *of the loop* `W` := `while e do {C1;⋯;Cn}` *is the graph whose vertices is the set of commands* {`C1`;⋯;`Cn`}*, and there exists a directed edge from* `Ci` *to* `Cj` *if and only if there exists variables* `x` ∈ Out(`Cj`) *and* `y` ∈ In(`Ci`) *such that* $\mathbb{M}$(`W`)(`x`, `y`) = ∞.

The last example of Sect. 2.3 gives `s1[i] = j*j` ⟶ `i++` ⟵ `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* `W`*,*

- *its* strongly connected components *(*SCC*s) are its strongly connected subgraphs,*
- *its* condensation graph G_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 `s1[i] = j*j` ⟶ `i++` ⟵ `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,

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

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

{ `s1[i] = j*j` ⟶ `i++` , `i++` ⟵ `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 $\mathtt{W} \coloneqq \mathtt{while\ e\ do}\ \{\mathtt{C}_1; \cdots ; \mathtt{C}_n\}$ ▷ Pick a loop `W` at top level

Compute the condensation graph $\mathrm{G}_{\mathtt{W}}$ *of* `W`, ▷ cf. Def. 13

Compute the saturated covering $G_1, \ldots, G_j$ *of* $\mathrm{G}_{\mathtt{W}}$*:* ▷ cf. Def. 14

while a node n in $\mathrm{G}_{\mathtt{W}}$ 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* $\mathrm{G}_{\mathtt{W}}$*:*

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 $\mathtt{W}_i \coloneqq \mathtt{while\ e\ do}\ \{\mathtt{C}_{i_1}; \cdots ; \mathtt{C}_{i_m}\}$ where $\{\mathtt{C}_{i_1}, \ldots, \mathtt{C}_{i_m}\}$ are the vertices of G_i, inserted in the same order as they are in `W`.

end for

Output: if $k > 1$, $\tilde{\mathtt{W}} \coloneqq \mathtt{parallel\{W_1\}\{\ \ldots\ \}\{W_k\}}$, else $\tilde{\mathtt{W}} \coloneqq \mathtt{W}$.

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.

3.2 Correctness of the Algorithm

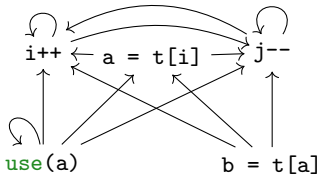
We now need to prove that the semantics of the initial loop `W` is equal to the semantics of $\tilde{\mathtt{W}}$ given by Algo. 1. This is done by showing that for any variable `x` appearing in `W`, its final value after running `W` is equal to its final value after running $\tilde{\mathtt{W}}$. We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

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

Proof. Let $\mathtt{W}_i$ be a loop in $\tilde{\mathtt{W}}$. By property of the saturated covering, the update commands are in the body of $\mathtt{W}_i$: 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 $\mathtt{C_u}$, then `C` and $\mathtt{C_u}$ are always both present in any subgraph of the saturated covering. Indeed, since there are edges from $\mathtt{C_u}$ to `C` and from `C` to $\mathtt{C_u}$, they are part of the same node in the condensation graph.

Since the condition of $\mathtt{W}_i$ is the same as the condition of `W`, and since all the instructions that impact (directly or indirectly) the variables occurring in that

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



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:

$$\texttt{parallel}\left\{\begin{array}{l}\texttt{while(i != j)\{}\\ \quad\texttt{i++;}\\ \quad\texttt{j--;}\\ \quad\texttt{a = t[i];}\\ \quad\texttt{use(a)\}}\end{array}\right.\left\{\begin{array}{l}\texttt{while(i != j)\{}\\ \quad\texttt{i++;}\\ \quad\texttt{j--;}\\ \quad\texttt{a = t[i];}\\ \quad\texttt{b = t[a]\}}\end{array}\right.$$

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

condition are present in $\mathtt{W}_i$, we conclude that the number of iterations of $\mathtt{W}_i$ and $\mathtt{W}$ are equal. □

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

Proof (sketch). We show that for every variable $\mathtt{x}$, the value of $\mathtt{x}$ after the execution of $\mathtt{W}$ is equal to the value of $\mathtt{x}$ after the execution of $\tilde{\mathtt{W}}$. 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 $\mathtt{x}$ and each loop $\mathtt{W}_i$ in $\tilde{\mathtt{W}}$ in which the value of $\mathtt{x}$ is modified, the value of $\mathtt{x}$ after executing $\mathtt{W}$ is equal to the value of $\mathtt{x}$ 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 $\mathtt{x}_1, \ldots, \mathtt{x}_h$ appearing in $\mathtt{W}_i$, the values of $\mathtt{x}_1, \ldots, \mathtt{x}_h$ after k loop iterations of $\mathtt{W}_i$ are equal to the values of $\mathtt{x}_1, \ldots, \mathtt{x}_h$ after k loop iterations of $\mathtt{W}$. Note some other variables may be affected by the latter but the variables $\mathtt{x}_1, \ldots, \mathtt{x}_h$ 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.

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

[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].

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.

```
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

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.

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

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. 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].

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.

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

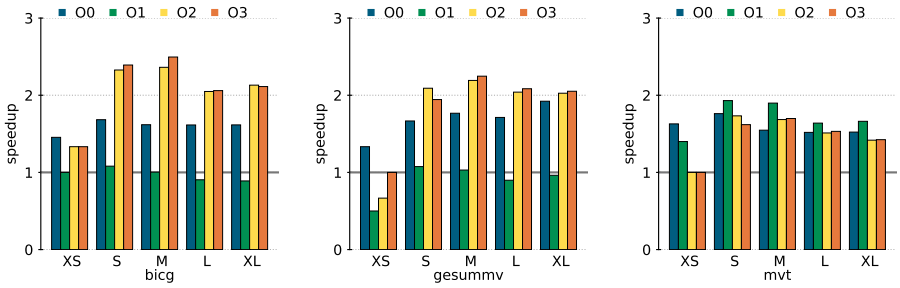


Fig. 6. Speedup of selected benchmarks implemented using `while` loops. Note the influence of various compiler optimization levels, `-O0` to `-O3` on each problem, and how parallelization overhead tends to decrease as input data size grows from `MINI` to `EXTRALARGE`. The gain is lower for `mvt` because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

`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.

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.

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.

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