Orchard: Heterogeneous Parallelism and Fine-grained Fusion for Complex Tree Traversals*

VIDUSH SINGHAL, Purdue University, West Lafayette, USA
LAITH SAKKA†, Meta, Menlo Park, USA
KIRSHANTHAN SUNDARARAJAH‡, Virginia Tech, Blacksburg, USA
RYAN R. NEWTON, Meta and Purdue University, Menlo Park, USA
MILIND KULKARNI, Purdue University, West Lafayette, USA

Many applications are designed to perform traversals on *tree-like* data structures. Fusing and parallelizing these traversals enhance the performance of applications. Fusing multiple traversals improves the locality of the application. The runtime of an application can be significantly reduced by extracting parallelism and utilizing multi-threading. Prior frameworks have tried to fuse and parallelize tree traversals using coarse-grained approaches, leading to missed fine-grained opportunities for improving performance. Other frameworks have successfully supported fine-grained fusion on heterogeneous tree types but fall short regarding parallelization. We introduce a new framework Orchard built on top of Grafter. Orchard's novelty lies in allowing the programmer to transform tree traversal applications by automatically applying *fine-grained* fusion and extracting *heterogeneous* parallelism.Orchard allows the programmer to write general tree traversal applications in a simple and elegant embedded Domain-Specific Language (eDSL). We show that the combination of fine-grained fusion and heterogeneous parallelism performs better than each alone when the conditions are met.

CCS Concepts: • **Software and its engineering → Software performance**; **Source code generation**.

Additional Key Words and Phrases: Automatic parallelization, fusion, compilers, optimization

1 INTRODUCTION

Tree traversals are an important component in many application domains. Compilers regularly perform multiple traversals over abstract syntax trees to analyze and transform the source code. Similarly, layout engines for web browsers perform multiple traversals of render trees to compute attributes of visual elements on web pages. Lastly, kd-trees can be used to represent numerical piece-wise functions over multidimensional spaces [Harrison et al. 2016], and operations over these functions require tree traversals. Performing multiple traversals over a tree can be time-consuming and inefficient. *Fusing* together multiple traversals to form coarse-grained tree traversals gives better performance due to improved locality and reduced number of traversals [Meyerovich and Bodik 2010; Meyerovich et al. 2013; Petrashko et al. 2017; Rajbhandari et al. 2016; Rajbhandari et al. 2016; Sakka et al. 2017].

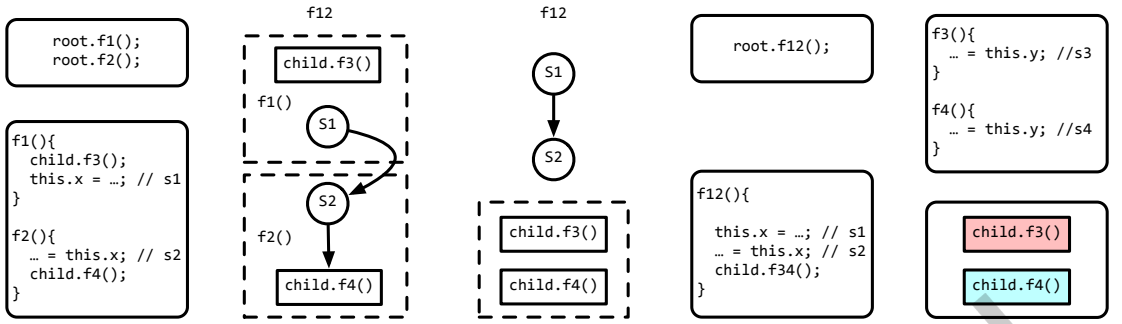
*Extension of Conference Paper: This manuscript extends Grafter published at PLDI 2019 [Sakka et al. 2019]. It achieves so by extending Grafter into a new framework Orchard, which is able to effectively extract parallelism from tightly interdependent tree traversals. We also add new insights about fusion and parallelism as mutually beneficial transformations.

*Most Work done while at Purdue University.

†Most Work done while at Purdue University.

Authors' addresses: Vidush Singhal, Purdue University, West Lafayette, USA, singhav@purdue.edu; Laith Sakka, Meta, Menlo Park, USA, lsakka@meta.com; Kirshanthan Sundararajah, Virginia Tech, Blacksburg, USA, kirshanthans@vt.edu; Ryan R. Newton, Meta and Purdue University, Menlo Park, USA, newton@meta.com; Milind Kulkarni, Purdue University, West Lafayette, Indiana, USA, milind@purdue.edu.

ACM 1544-3566/2024/3-ART
https://doi.org/10.1145/3652605



(a) Unfused code (b) Outline and inline (c) Reorder step (d) Fused code (e) exposed parallelism

Fig. 1. Fusion process in Grafter and parallelism opportunity identified by Orchard

In a coarse-grained approach to fusion, the computation of two separate traversals can be fully combined only if the combined work is computable in a single pass over the tree. If not, the two traversals are not fusible. On the other hand, it may be possible to combine the execution of two traversals only over some portion of the tree (subtree). Additional passes over the tree may be required to execute the combined work correctly. This results in partial fusion [Sakka et al. 2017, 2019]. Partial fusion yields locality and traversing overhead reduction benefits where the coarse-grained fusion approach would fail to do so.

A tree traversal may be executed in parallel to utilize multi-core performance. It is straightforward to parallelize a tree traversal by visiting different subtrees in parallel in the absence of dependences in the computation. We coin the term *homogeneous* parallelism for this type of classic tree-based parallelism, where all subtrees are treated identically and independently. Homogeneous parallelism restricts speedups as it only allows *complete and independent* traversals to run in parallel. In contrast, *heterogeneous* parallelism allows different subtrees of a traversal to be processed independently in parallel, even while other subtrees are processed sequentially to preserve dependences. The latter is a more fine-grained approach and exposes greater opportunities for parallelism. In Cilk-style parallel programming [Blumofe et al. 1995], homogeneous parallelism is equivalent to *spawn*ing all children traversals on a node in the tree, while heterogeneous parallelism selectively spawns and syncs subtrees to preserve dependences. Additional opportunities for heterogeneous parallelism in the face of dependences can be exposed by reordering the traversal order of children in a tree [Sakka et al. 2017].

Consider the example code in Figure 1. In Figure 1(a), we call two functions, f1 and f2, on the root node of a tree. To an untrained eye, it may seem that these two functions can be executed readily in parallel. However, a close inspection of the function bodies of f1 and f2 reveals that this.x has written to in f1 and then read in f2. There is a read-after-write dependence between f1 and f2. Hence, these are not trivially parallelizable traversals. If a programmer tries to parallelize the calls to f1 and f2 on the root node, it will lead to a race condition, thus breaking the correctness of the code. The process of outlining and inlining, followed by fusion, brings the computation of f1 and f2 closer. The dependence analysis that Grafter does realizes this dependence and does fusion such that it does not break any dependences. The only difference is that f1 changes from a post-order traversal to a pre-order traversal in the fused function f12. In addition, the child calls are brought close together in the new fused function f34. We see that f3 and f4 are independent functions without any dependences (See Figure 1(e)). By bringing the child calls to f3 and f4 closer, we are able to trivially parallelize the child traversals. Figure 1(e) shows f34 with the two calls in different colors, symbolizing that the calls are executed in parallel. By re-ordering the traversal and performing fusion, Orchard is able to exploit heterogeneous parallelism in the face of dependences.

There is an inherent tension between writing simple, maintainable code and fully exploiting fusion and parallelism. The most programmer-friendly approach prioritizes writing several small traversals, where each traversal performs a small, logically coherent operation [Petrashko et al. 2017; Qasem and Kennedy 2006; Rajbhandari et al. 2016; Rajbhandari et al. 2016; Sakka et al. 2017, 2019; Sarkar and Waddell 2005]. But short, lightweight traversals both reduce locality and have little to no parallelism. The same tree must be traversed many times, reducing temporal locality, and fine-grained parallel tasks may be overwhelmed due to parallel overheads. Manually fusing and parallelizing traversals requires a lot of time and effort and may introduce new concurrency bugs. Due to complex dependences between operations, performing multiple operations in a single traversal may require splitting up a single logical computation across multiple traversals or putting unrelated operations together into a single traversal. Programmers may not be able to maintain these complex dependences while rewriting software.

Meyerovich et al. [Meyerovich et al. 2013] addressed this problem by using attribute grammars to automatically create parallel schedules for tree traversals. However, their approach neither supports fine-grained fusion nor creates a heterogeneous parallel schedule. Rajbhandari et al. [Rajbhandari et al. 2016] describe the creation of a compiler that optimizes kd-tree traversals using fusion in the MADNESS framework. Their fusion approach requires that the traversals act in either a pre-order or post-order manner only. The traversals must also visit all the tree nodes in the tree, and the nodes must be of the same type. In addition, the recursive functions are only parallelized in a homogeneous pre-order or post-order manner, which is a coarse-grained approach. These limitations mean that their framework does not support partial fusion, heterogeneous tree types, or heterogeneous parallelism.

Grafter by Sakka et al. [Sakka et al. 2019] addresses the concept of partial fusion and fusing traversals of heterogeneous trees. However, Grafter falls short because it does not exploit parallelism, limiting the speedup potential. Hecate [Chen et al. 2022] is a tree traversal synthesis framework that uses an SMT solver to generate fused schedules. Their framework generates programs that perform better than Grafter, presenting the most novel fusion framework to date. Although they parallelize their programs to get better performance, they rely on the programmer to find such parallelism opportunities in the source code. Their framework is limited in extracting parallelism because it requires significant manual analysis. The programmer has to manually use specific data structures, such as *vectors*, to find parallelism. In short, there is no existing framework that allows programmers to write general tree traversal applications and, at the same time, make them perform faster *automatically* by extracting both fine-grained fusion *and* heterogeneous parallelism.

This paper showcases a new framework, Orchard, which is an extension to Grafter that automatically transforms complex tree traversal applications by extracting both fine-grained fusion and heterogeneous parallelism to generate faster code. To our knowledge, Orchard is the first framework to combine both fine-grained fusion and heterogeneous parallelism.

(1) We show that this combination of transformations is mutually beneficial: fine-grained fusion coarsens task granularity, hence making parallel computations more efficient as it compensates for various parallel overheads. In addition, it reduces overall work and cache misses. In applications with abundant parallelism, the programmer can allow greater fusion and vice-versa.
(2) Orchard allows the programmer to fine-tune applications for bigger parallel speedups by controlling the granularity of fusion.
(3) Orchard is able to extract *heterogeneous* parallelism in situations where it is not possible to spawn whole subtrees in parallel. Heterogeneous parallelism can extract more parallel work by using a combination of spawns and syncs to preserve dependences.
(4) We show across three case studies that Orchard can provide substantial speedups over Grafter and is able to exploit significant parallelism from complex, tightly interdependent tree traversals.

(5) We compare the performance of Orchard with Hecate, the best prior framework, and show that it is faster than Hecate on the benchmarks that they evaluated.

The rest of the paper is organized as follows: Section 2 gives an overview of Grafter. Section 3.1 discusses *fusion and parallelism* as mutually beneficial optimizations and their trade-offs. Section 3.2 introduces Algorithm 1 to extract parallelism. Section 3.3 explains our parallel code-generation strategy. We showcase our results in Section 4. We discuss related work in Section 5, and Section 6 presents future work. Finally, Section 7 concludes the paper.

2 BACKGROUND

We build Orchard on top of Grafter. Frameworks other than Grafter that allow programmers to automatically fuse tree traversals cannot express general tree traversals. Those approaches miss opportunities for fusion as they use a coarse-grained approach. Coarse-grained fusion only allows fusion among tree traversals that can execute fully on a tree without breaking dependences. If traversals cannot be fully fused, they are left unmodified. But multiple traversals may be executed together in the same pass over different parts of the tree. This creates opportunities for partial fusion. Partially fused traversals give increasingly better performance as the size of the tree increases. In addition, frameworks that only support coarse-grained fusion do not support heterogeneous trees, which limits the types of tree traversals optimized.

Grafter is a Clang-based tool that performs fusion on the input source code and produces a fused version [Sakka et al. 2019]. Grafter's fused version gives better performance than the original unfused version because of better locality by reducing the reuse distance of memory accesses. Grafter supports fine-grained fusion by allowing traversals to be *partially-fused*. One of the main novelties of Grafter lies in its dependence graph representation [Sakka et al. 2017] of the program, which captures all the dependences in program order. It creates a Directed Acyclic Graph (DAG) representation where statement nodes and call nodes are distinguished from each other, and directed edges among the nodes represent dependences. To construct a DAG representation capturing the various dependences in the program, Grafter constructs *access automaton* for each statement and call [Sakka et al. 2019]. The automaton summarizes the possible sets of reads (or writes) that a statement or call can execute on the tree. By taking intersections of the access automata for calls and statements, Grafter builds a DAG that captures all the dependences among calls and statements in program order.

Grafter follows a set of steps repeatedly to fuse traversals. First, it finds an ordered sequence of functions invoked on the same tree child, which are subject to an outlining step followed by an inlining step. Then Grafter reorders statements to bring them closer and creates new sequences of traversal functions invoked on the same child. In short, this new sequence constitutes the newly fused version of the functions. Figure 1 shows an illustration of this process. Grafter creates a topological ordering of the nodes using the newly created fused dependence graph to generate the fused version of the source code. This topological sort generates a sequential program order that respects the original dependences. Grafter supports "code motion," which is the process of moving calls around to expose more fusion opportunities. In the fused version of the source code, a call can move from being a post-order to a pre-order traversal if no dependences are broken.

Ultimately, Grafter's version of the output code is almost always faster than the original unfused version of the code. Programs with multiple fine-grained traversals fuse into coarse-grained ones that perform more computation per traversal. This improves locality (by bringing accesses to the same tree node closer) and increases efficiency (by requiring a fewer number of traversals). When the overhead added after fusion outweighs the locality benefits, the version produced by Grafter becomes slower. However, Grafter does not exploit the parallelism of these traversals. Orchard provides programmers with the ability to automatically get the benefits of both fine-grained fusion *and* heterogeneous parallelism, which together give better performance than either one of fine-grained fusion *or* heterogeneous parallelism.

2.1 Grafter Overview

Grafter adopts a strategy for fusing traversals where a programmer writes individual tree traversals in a standard C++-like language [Sakka et al. 2019] as functions that traverse a tree structure. The fields of each tree node can be heterogeneous and part of a complex class hierarchy, and the (mutually) recursive functions that perform a tree traversal can leverage dynamic dispatch when visiting the children of a node. For illustration purposes, we use a simple example shown in Figure 1(a), with two calls to the functions f_1 and f_2. Each of those functions consists of a single statement (s_1 and s_2, respectively) and a *traversing* call on the root's child (f_3 and f_4, respectively). These two function calls are not independent of each other: s_1 in f_1 updates this.x while s_2 in f_2 reads this.x. Grafter performs fusion on such traversals to generate a new set of mutually recursive functions that perform fewer traversals of the tree. The fusion process starts with a sequence of traversals that are invoked on the same node of a tree, in this case, the root. Note that it is clearly safe to outline the two calls in Figure 1(a) into one call to function f_{12} that executes the two functions back-to-back in their original order as shown in Figure 1(b) (the two calls are outlined and then inlined).

Grafter creates a *dependence graph* representation. This dependence graph represents each call and statement of the traversals as a vertex, and (directed) edges are placed between vertices if two statements *when executing at a particular node in the tree* can access the same memory location. In Figure 1(b), there is a dependence between s_1 and s_2. We also assume, for illustration purposes, that there is a dependence between s_2 and the call $child.f_4()$. Grafter finds dependences between calls and statements by considering the transitive closure of what calls may access. Section 2.2 describes how Grafter analyzes accesses to find dependences and construct the dependence graph.

Function f_{12} is now a new, single one that performs multiple pieces of work on root and invokes multiple traversals on root.child. To optimize the body of f_{12}, it is desirable to have $s1$ and $s2$ executed closer to each other for locality benefits – if they access shared fields of root, then that data is likely to remain in the cache. Furthermore, if the calls f_3 and f_4 on child are back-to-back, then we can further fuse them into one call, and both save a function invocation and "visit" root.child only once, instead of twice.

The key challenge to performing this fusion is that it is not always safe. Grafter performs reordering for the statements using the dependence graph, trying to bring traversals of the same child closer to each other without reordering any edges (and hence violating dependences). This reordering is done by grouping the traversal calls that visit the same node together. Figure 1(c) shows the results of such reordering.

Note that in this example, the reordering step changes the traversal f_1 from a post-order traversal to a pre-order one ($s1$ is executed at parents before their children in f_{12}, while it is executed at the children before their parents in the original function $f1$). In other words, this reordering involves performing implicit code motion that can safely changes the schedule of the traversals for the purpose of achieving more fusion.

As calls are grouped together, Grafter is presented with *new* sequences of functions that this same fusion process can be applied to. In our example, the two calls grouped in the dashed box in Figure 1(c) are invoked on the same node of the tree (root.child), so Grafter can *repeat* this merging process, creating a new merged function f_{34}, building a new dependence graph for the merged function, rearranging its statements, and so on. Each time this reordering is performed, more and more operations from multiple logical traversals on the same node(s) of the tree are brought closer together, improving locality, and more and more function invocations from multiple logical traversals are collapsed, reducing invocation overhead and the total number of times the collection of traversals visit nodes of the tree.

If Grafter encounters a sequence of functions that has been fused before, it can simply call the already-fused implementation. Grafter bounds the number of functions that can be fused together to ensure this process terminates. Section 2.3 describes in detail how Grafter performs its fusion, including how it handles virtual functions. Note that encountering cases where an already created fused function is being called again is the

```
1  int CHAR_WIDTH;
2
3  class Element {
4    Element *Next;
5    int Height = 0, Width = 0;
6    int MaxHeight = 0, TotalWidth = 0;
7    virtual void computeWidth(){};
8    virtual void computeHeight(){};
9  };
10
11 class TextBox: public Element {
12   String Text;
13   void computeWidth(){
14     Next->computeWidth();
15     Width = Text.Length;
16     TotalWidth = Next->Width + Width;
17   };
18   void computeHeight(){
19     Next->computeHeight();
20     Height = Text.Length * (Width/CHAR_WIDTH) + 1;
21     MaxHeight = Height;
22     if(Next->Height > Height)
23       MaxHeight = Next->Height;
24   };
25 };
```

```
26 class Group: public Element {
27   Element *Content;
28   BorderInfo Border;
29   void computeWidth(){
30     Content->computeWidth();
31     Next->computeWidth();
32     Width = Content->Width + Border.Size*2;
33     TotalWidth = Width + Next->Width;
34   };
35   void computeHeight(){
36     Content->computeHeight();
37     Next->computeHeight();
38     Height = Content->MaxHeigh + Border.Size*2
39     MaxHeight = Height;
40     if(Next->Height > Height )
41       MaxHeight = Next->Heigh;
42   };
43 };
44
45 class End: public Element {
46 };
47
48 int main(){
49   Element *ElementsList = ...;
50   ElementsList->computeWidth();
51   ElementsList->computeHeight();
52 }
```

Fig. 2. An example of a program written in Grafter.

key to having significant performance improvement since the locality enhancement and traversing overhead reduction will be achieved recursively. In the limit, instead of 2 traversals visiting each node of the tree once each, we will have a single traversal that visits each node only once – *total fusion*. However, any amount of collapsing still promotes locality and reduces node visits.

The end result of Grafter's fusion process is a set of mutually recursive functions that together form a partially fused traversal. Crucially, these fused functions are analyzed on a per-type basis. In other words, the fusion can occur *partially* – not all sequences of calls in a function need to be fused – and *type-specifically* – fusion can occur for some concrete instantiations of virtual functions, but not for others. This process leads to more fine-grained, precise fusion decisions than prior work.

The following section describes the details of Grafter fusion process in details. First, we describe Grafter's analysis of traversal functions to identify dependences and the dependence graph representation used to drive fusion (Section 2.2). Then, we explain how Grafter uses the dependence representation to synthesize new, fused functions(Section 2.3). For detailed information on Grafter's language, refer to the grafter paper [Sakka et al. 2019].

2.2 Dependence Graphs and Access Representations

The primary representation that Grafter uses to drive its fusion process is the dependence graph [Sakka et al. 2017]. As described in Section 2.1, this graph has one vertex for each *top-level* statement and edges between statements if there are dependences between them. More precisely, an edge exists between two vertices v_1 and v_2, arising from functions f_a and f_b (f_a and f_b could be the same function) if, when invoking f_a and f_b *on the same tree node* (*i.e.*, when this is bound to the same object in both functions), either:

(1) v_1 and v_2 may access the same memory location, with one of them being a write or
(2) v_1 is control dependent on v_2 (in Grafter's language, this can only happen if v_1 and v_2 are in the same function and either v_1 or v_2 could return from the function).

So, how does Grafter compute these data dependences?

Access automata. To compute dependences between statements in different traversals, the first step is for Grafter to capture the set of accesses made by any statement or call in a given traversal function. To do so, Grafter builds *access automata* for each statement. These can be thought of as an extension of the regular expression-based access paths used by prior work [Sakka et al. 2017; Weijiang et al. 2015] to account for the complexities of virtual function calls and mutual recursion.

An access path for a simple statement such as n.x = n.l.y + 1 is straightforward. The statement *reads n.l.y* and *writes n.x*. A simple abstract interpretation suffices to compute these access paths (intuitively, we perform an alias analysis on the function using access paths as our location abstraction [Larus and Hilfinger 1988; Wiedermann and Cook 2007]). The abstract interpretation associates with each local variable an access path, or set of access paths, when merging across conditionals, aliased to that variable. At each read (or write) of a variable, the access path(s) are added to the read (or write) set of access paths for that statement. Grafter collects the set of access paths for each top-level simple statement in each traversal function. We do not elaborate further on this process, as this analysis is standard (and is similar to TreeFuser [Sakka et al. 2017]).

The more complicated question is how to deal with building access paths for traversing function calls. Our goal is to build a representation that captures *all possible access paths* that could arise as a result of invoking the function. Rather than trying to construct path expressions to summarize the behavior of function calls, Grafter directly constructs *access automata* to account for this complexity. Note that these access automata are not quite like the aliasing structures computed by Larus and Hilfinger [Larus and Hilfinger 1988], because Grafter's representation is deliberately parameterized on the current node that a function is invoked on. We describe how Grafter builds these automata next.

Building access automata for statements. Each top-level statement in Grafter has six automata associated with it that represent reads and writes of local, global, and tree accesses that can happen during the execution of the statement. Grafter starts by creating primitive automata. For each access path, a primitive automaton is constructed which is a simple sequence of states and transitions. Transitions in the automata are the member accesses in the primitive access path except for two special transitions: (1) the *traversed-node* transition, which appears only at the start of tree-node-field access and replaces **this**, and (2) the "*any*" transition that happens on any member access. If a primitive access path is read, then each prefix of the primitive access is also being read, and accordingly, each state in the primitive automata is an accept state except the initial state. If a primitive access is being written, then only the full sequence is written to while the prefixes are read. There are some special cases to deal with while constructing primitive automata. If an access ends with a non-primitive type (a C++ object), then accessing that location involves accessing any possible member within that structure. Such cases are handled by extending the last state with a transition to itself on any possible member using an *any* transition. Likewise, tree locations that are manipulated using *delete* and *new* statements, writes to any possible sub-field accessed within the manipulated node, and their automata uses *any* transition to capture that. After the construction of the primitive automata, access automata of simple statements can be constructed from the union of the primitive automata. For example, the tree reads automaton for a simple statement is the union of the primitive automata of the tree read accesses in the statement. Figure 3 shows the tree read automaton for the statement:

```
Width = Content->Width + Border.Size*2;
```

Finding dependences between statements. These automata provide the information needed to find dependences between statements. Because each statement's automata captures the full set of access paths read (or written) for a statement, and we are interested in whether the statements have a dependence *when invoked on the same tree node*, we can simply intersect the write automaton for a statement with the read and write automata for another

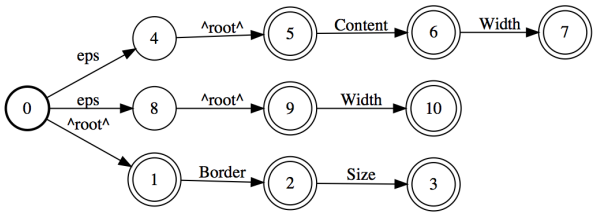


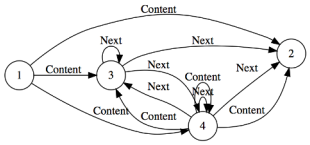
Fig. 3. Automata that represents summary of read accesses for the simple statement. *eps* is epsilon transition, and *root* is the traversed node transition (*i.e.*, *root* corresponds to **this** object). The double circle represents an accept state.

statement to determine if a dependence could exist—a non-empty automaton means the two statements could access the same location.

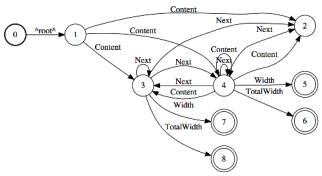
Building access automata for traversing calls. Representing accesses of traversal calls is not as simple. For a given call statement, we want to construct a finite automaton that captures *any possible access path that could arise during the call relative to the tree node being traversed by caller* – including the fact that a call may invoke more traversals. When building the access automaton for a traversal call, Grafter first creates a call graph that includes all the possibly (transitively) reachable functions from that call. We first note that any *off-tree* data accesses made by any of these reachable functions are, inherently, not parameterized by the receiver of the traversal calls—regardless of when and where the function gets called, those access paths will be the same. Thus, we can simply union those automata together for the functions in the call graph to capture those accesses. The situation is more complicated for tree-node-field accesses, as those are parameterized by the receiver of the call, this (*i.e.*, the node that is being traversed). To find the accessed locations relative to this, we need to find two things: the functions that are reachable during the call (to know which statements are executed) and the tree nodes that those functions are invoked on.

Consider building the access paths for some function f. For each function q reachable from f, the sequence of children traversed to get to the invocation of q gives an access path for the node that q is invoked on relative to the receiver of f. This access path can be prepended to the statements access paths in q to produce the access paths relative to this (the traversed node in f). For example, when a function f invokes another function g on a child x of this, it invokes x.g(); the receiver object of g is this.x. Thus, to incorporate the effects of g into the access paths of f, we can prefix the access paths of g with this.x. If g, in turn, calls h through child y, then we can prefix the access paths of h with this.x.y and add them to the access paths of f.

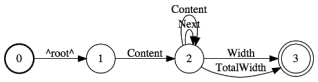
To account for tree-node-field accesses, Grafter takes the call graph for the traversing call and labels each edge with the traversed field that is the receiver for that call. If the receiver for the call is the currently traversed node (this), the edge is labeled with the epsilon (*eps*) transition. Figure 4(a) shows the call graph that is generated for `Content->computeWidth()` from our running example. Thus, paths in this graph correspond to possible sequences of child-node accesses to reach each function in the graph. For each function, Grafter then attaches the *statement* automata of each simple statement in that function (see the previous section) to the corresponding node in the call graph. This has the effect of treating the regular language from the statement automata as the suffix attached to the prefix that designates the receiver object. Figure 4(b) shows the resulting automaton, and Figure 4(c) shows the reduced version. Note that the constructed automata handle the possibility of non-statically bounded trees; whenever we encounter a function for which we already have created a state, we add a "back edge" in the automaton to the state that corresponds to that function. Unbounded recursion is hence represented by loops in the automaton.



(a) Step 1: Create labeled call graph. States 2, 3 and 4 corresponds to computeWidth() functions for End, TextBox and Group types respectively.



(b) Step 2: Attach simple statements' automata and traversed node transition.



(c) Step 3: Minimize automata.

Fig. 4. Construction of tree writes automata for the traversing statement *content->computeWidth(). root* is the traversed node transition.

Finding dependences between statements and calls. The access automata constructed for calls are no different than those constructed for statements. By using the call graph to construct access paths for receiver objects of functions, all the access paths generated by the final access automata are rooted at the same receiver object as the automata for statements. Hence, finding dependences between statements and calls (or between calls) can be achieved by intersecting the automata and testing for emptiness.

2.3 Fusing Traversals

At a high level, Grafter performs fusion by repeatedly invoking the following steps:

(1) Find a sequence L of consecutive traversal functions invoked on the same tree node n.
(2) *Outline* these traversal functions into a new function f_L that is called on n. If such an f_L has already been created in an earlier iteration of this process, simply call the existing f_L.
(3) *Inline* each of the individual traversal functions in f_L to expose the work done on the node n.
(4) *Reorder* the statements in f_L to bring statements that access the same fields closer together *and* to create new sequences of traversal functions invoked on the same tree node (typically, some child node of n).
(5) Repeat the process for these newly created sequences of calls.

These steps constitute the fusion algorithm of Grafter. In particular, every time a sequence L is encountered *more than once*, and hence an existing f_L can be reused, Grafter has exploited an opportunity for fusion. We now explain this fusion process in more detail, and also sketch a proof of correctness.

Details. Fusion starts with a sequence of traversal functions that are invoked at the same tree node (*e.g.*, the root). Grafter searches for such candidates in the compiled program and initiates the fusion process for each of them. For example, the sequence ElementsList− > computeWidth() followed by ElementsList− > computeHeight() in Figure 2 line 50.

Because a given function may be virtual, Grafter first computes all possible sequences of *concrete* functions that may be invoked as a result of a sequence of function calls. For each type T that the sequence of calls can be invoked on, Grafter constructs a sequence L of concrete calls. In our example, there are three, depending on whether `ElementList` points to a `TextBox`, `Group` or `End`.

For each function sequence L a fused function with label f_L is created (if one has not already been generated). If the label f_L does not already exist, then its corresponding function needs to be generated. A dependence graph G_L is constructed for the statements in the traversals in L. In other words, the fused function is essentially a function containing the *inlined* statements from each call in the sequence L, in order. Note that a sequence L may contain the same static function more than one time (i.e., the same function can be invoked on a given node

of a tree more than once). In this case, G_L contains statements from multiple copies of that function, and the statements from the two copies are treated as coming from different traversal functions.

Once G_L is constructed, the statements (nodes) in the statement can be reordered as long as no dependences are violated (as long as a pair of dependent statements are not reordered). Grafter thus reorders the statements and try to group invocations on the same node together. It then generates the fused function code, as explained in the next section. This newly generated function has grouped traversals invocations on the same node together (and these invocations may have come from different functions than the original sequence L), creating new sequences of functions. Grafter then process these new sequences of functions to generate more fused functions. Whenever Grafter encounters a sequence of functions it has seen before, it does not need to generate a new function, but instead inserts a call to the already generated function. Crucially, if this new sequence is *the same as for a function currently being generated*, Grafter just inserts a recursive call to that function.

The end result is a set of mutually recursive fused functions, each for a different set of traversals that are executed together at some point. Furthermore each of those functions is fused independently of the others. This process introduces *type-specific-partial-fusion*, since for an invocation of a traversals on a super type, the set of the called functions that corresponds to each dynamic type are fused independently, and hence some of them actually might be fusible while others are not. Note that Grafter only generates new functions for sequences it has not seen before. Because Grafter limits the number of functions that can be fused together [Sakka et al. 2019], the number of sequences of functions is finite, and hence fusion is guaranteed to terminate.

Proof sketch of soundness. The argument for the soundness of Grafter's fusion procedure is straightforward. First, we note that the outlining and inlining steps (steps 2 and 3) in the fusion process are trivially safe because they do not reorder any computations, and hence cannot break any dependences. Step 4 could potentially break dependences, but because Grafter performs a dependence analysis, it can ensure that statements are only reordered if dependences are preserved. Hence, this step is also clearly sound. The tricky step in Grafter's fusion algorithm is the step where it gains the advantage of fusion: if a sequence of calls to a particular sequence of traversal functions matches a sequence that Grafter has already generated a fused function for, Grafter immediately replaces the original sequence of calls with a call to the fused function rather than generating another new function. This is only safe if the already-fused function will do the same thing as the original function sequence.

To see that this is safe, we note that the process of outlining followed by inlining means that the code of the fused function f_L is not dependent on the node f_L is invoked on—in other words, if the original sequence of traversal functions are invoked on `root.left`, after outlining and inlining, the statements within f_L are relative to the formal parameter `n` of f_L, and will be exactly the same as if f_L were produced from the same sequence of functions invoked on a different tree node, such as `root.right`. In other words, two identical sequences of traversal functions, L and L' that are invoked on different tree nodes will yield *identical* functions f_L and $f_{L'}$ after outlining and inlining. Because the dependence graph for these functions are identical, any reordering Grafter does to create a fused function can be applied to both f_L and $f_{L'}$. It is obvious, then, that, upon encountering the same sequence of traversal functions L, even if those functions are invoked on different nodes, Grafter can reuse an existing synthesized function.

However, there remains one gap: if, while fusing a sequence of functions L to generate f_L, Grafter encounters the same sequence of invocations L that is reachable (transitively) from f_L, Grafter will substitute a call to f_L. In the simplest case, if f_L contains L, then a new invocation to f_L will be inserted into the body of f_L. Hence, in these situations, Grafter is changing the behavior of f_L while using it to replace L. This process feels circular. However, a straightforward inductive argument on the depth of the call stack (*i.e.*, the number of recursive invocations of f_L before reaching the end of the tree or some base case) shows that this new invocation of (the

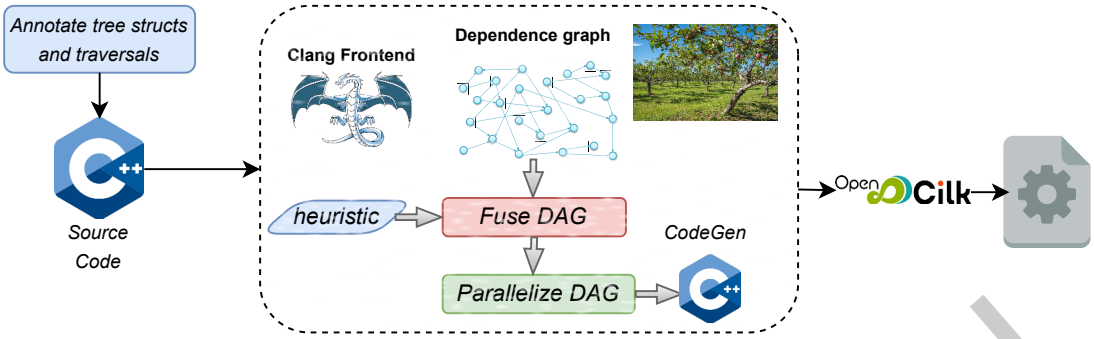


Fig. 5. Orchard design overview

rewritten) f_L behaves the same as the original sequence L. This argument mirrors the proof for the soundness of TreeFuser [Sakka et al. 2017, Section 7].

3 ORCHARD DESIGN

Orchard's goal is to generate efficient parallel code by automating the task of extracting parallelism from the source code. Manually parallelizing traversals is not a straightforward task and requires careful tweaking in order to get performance. Our key insight is to use the dependence graph generated by Grafter to parallelize traversals.

Figure 5 gives an overview of the framework. As shown, the dependence graph is at the core of the transformations. We first fuse the dependence graph using a fusion heuristic and subsequently extract parallelism. This is followed by a code generation step and a compilation step, which generates a target binary using OpenCilk. Before we delve further, we talk about fusion heuristics.

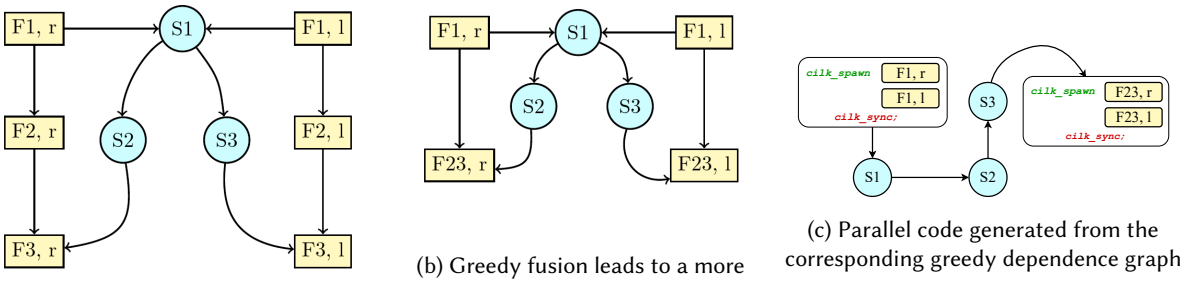
3.1 Fusion Heuristics

We use Orchard to generate two different types of code. One uses the greedy fusion heuristic from Grafter, and the other is representative of the original parallelism in the source code by completely ignoring the fusion step. We show these as two extreme strategies on the scale of possible fusion schedules. Future work will focus on the space of fusion heuristics such that it increases parallelism optimally.

The greedy fusion heuristic. The greedy heuristic starts to fuse all the possible candidate functions for fusion it can find as long as there are no more functions to be fused. The idea behind this heuristic is to fuse the maximum amount of functions together. As shown in Section 4, fusion reduces overall work, improving spatial and temporal locality due to fewer cache misses. It coarsens the granularity of functions, which makes parallelism efficient.

Figure 6 shows the difference between parallel schedules generated with and without greedy fusion. As shown, the parallel schedule created after greedy fusion is more efficient than the parallel schedule with no fusion. It does not reduce parallelism but coarsens the granularity of parallelism as functions 2 and 3 are now fused together.

Theoretically, as shown in Figure 7, the greedy fusion heuristic may not always lead to a more efficient parallel schedule. Allowing excessive fusion can reduce opportunities for parallelism. Suppose two functions are fusible and at the same time can execute in parallel, which is the better choice? While fusion does help in increasing locality, parallelism brings more dramatic increases in speedups. However, parallel overheads may prevent speedups if a function is not computationally heavy. In such a scenario, fusing those functions is a better choice.



(a) An unfused dependence graph

(b) Greedy fusion leads to a more efficient graph

(c) Parallel code generated from the corresponding greedy dependence graph

Fig. 6. Figure shows benefits of greedy fusion.

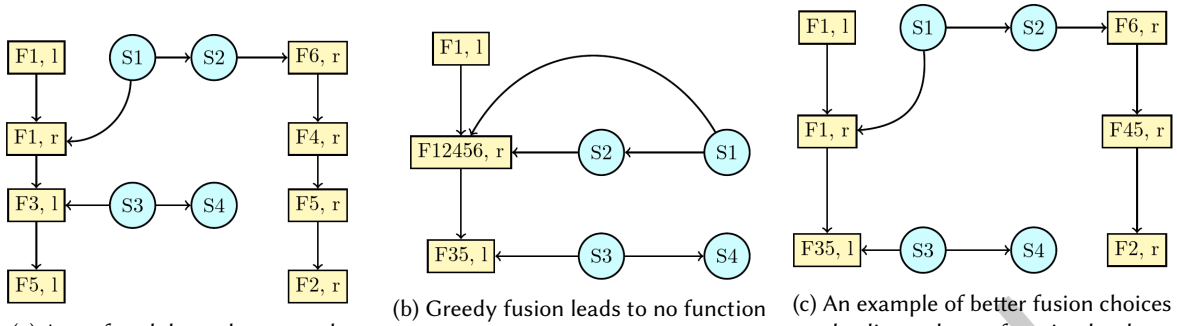
Fusion generates computationally dense traversals. Figure 7(c) shows a dependence graph where carefully chosen fusion choices lead to an increase in computational density of traversals without decreasing parallelism.

3.2 Parallelizing Traversals

Algorithm 1 Pseudo-code for the topological sort algorithm

```
1:  function updateReadyList(readyList, node)
2:      for each successor ϵ node do
3:          if not visited[successor] and allPredessorsVisited(successor) then readyList ← readyList + [successor]
4:          end if
5:      end for
6:  end function
7:
8:  function topologicalSort(dependenceGraph)
9:      for each node ϵ dependenceGraph do
10:         if node is root then readyList ← readyList + [node]
11:         end if
12:     end for
13:     while readyList is not empty do
14:         for each node ϵ readyList do
15:             if node is not a CallNode then
16:                 if visited[node] then readyList ← readyList − [node]
17:                 else
18:                     statementOrder ← [node] ; executionOrder ← executionOrder + [statementOrder]
19:                     visited[node] ← true ; readyList ← readyList − [node]
20:                     updateReadyList(readyList, node)
21:                     for each callnode ϵ readyList do
22:                         if not visited[callnode] then
23:                             visited[callnode] ← true ; parallelFunctions ← parallelFunctions + [callnode]
24:                         end if
25:                     end for
26:                     readyList ← [ ]
27:                     for each callnode ϵ parallelFunctions do updateReadyList(readyList, callnode)
28:                     end for
29:                     executionOrder ← executionOrder + [parallelFunctions]
30:                 end if
31:             end if
32:         end for
33:     end while
34:     return executionOrder
35: end function
```

As mentioned previously in Section 2.1, the dependence graph contains vertices and edges between vertices capture data flow dependencies. These vertices are either function calls or statements. The dependence graph



(a) An unfused dependence graph

(b) Greedy fusion leads to no function level parallelism

(c) An example of better fusion choices leading to better function level parallelism

Fig. 7. Figure shows how greedy fusion may be bad and provides a more efficient dependence graph.

preserves dependences among function calls and statements in program order. Vertices contain vectors storing its predecessor and successor vertices. At runtime, data accesses made by the currently executing function are dependent on accesses made previously by its predecessors. Likewise, the successors depend on data accesses made by the currently executing function. To prevent a race condition, function calls executed in parallel must be completely *independent* of each other. As a function executes, any set of vertices in the dependence graph that are not yet visited and whose predecessors have all been visited are candidates to execute in parallel without breaking any dependences. A topological sort (Algorithm 1) orders the vertices in the dependence graph according to the dependence order, and it finds function calls that can execute in parallel along the critical path of the dependence graph. Orchard does not extract statement-level parallelism because it is expensive. Individual statements are computationally lightweight; parallelizing statements will not lead to efficient parallelism because of various parallel overheads. All statements are thus executed serially within the body of a function, and they are ordered to be consecutive in the function body.

After the algorithm has visited all the vertices in the dependence graph, it outputs a parallel schedule that defines the order in which statements and function calls should be executed. Imagine this parallel schedule as a multi-story building. Each level contains either a bunch of function calls that can execute in parallel or an individual statement that executes serially. The levels in this building respect the program order and execute serially. Each individual concrete function in the dependence graph is parallelized separately in accordance with dependences. Function calls can be invoked on different children (`f1(node.r)` and `f2(node.l)` act on different children), or descendants of those children at different levels in the tree (`f1(node.r)` and `f2(node.r.l)` act at different levels). These functions can traverse the tree in any traversal order as well. This enables Orchard to generate a parallel schedule that is heterogeneous and fine-grained.

Figure 6(a) shows a dependence graph representation of some source code. Inside the call vertex (yellow rectangles) is information containing the call name followed by the tree child it traverses in a binary tree. Statements are shown in blue circles. Figure 6(c) shows a parallel schedule of the dependence graph in Figure 6(b). Our evaluation shows that parallelizing the dependence graph after Grafter does fusion on the dependence graph yields considerable speedups. This mainly stems from the coarsening of functions after fusion, which amortizes various parallelism overheads and lowers cache misses. In addition, fusion may lead to more parallelism by reordering the traversal order as highlighted in Figure 1.

```
1  void _fuse__F69F70F71F72F73_parallel(PageListInner *_r,
2                                       struct FontInfo _f2_ParentFontStyle,
3                                       unsigned int truncate_flags,
4                                       int depth,
5                                       int maxDepth) {
6
7  #ifdef COUNT_VISITS
8    _VISIT_COUNTER++;
9  #endif
10   PageListInner *_r_f0 = (PageListInner *)(_r);
11   PageListInner *_r_f1 = (PageListInner *)(_r);
12   PageListInner *_r_f2 = (PageListInner *)(_r);
13   PageListInner *_r_f3 = (PageListInner *)(_r);
14   PageListInner *_r_f4 = (PageListInner *)(_r);
15   /*Parallel Call 1*/
16   if ((truncate_flags & 0b11111)) /*call*/ {
17     unsigned int AdjustedTruncateFlags = 0;
18     AdjustedTruncateFlags <<= 1;
19     AdjustedTruncateFlags |= (0b01 & (truncate_flags >> 4));
20     AdjustedTruncateFlags <<= 1;
21     AdjustedTruncateFlags |= (0b01 & (truncate_flags >> 3));
22     AdjustedTruncateFlags <<= 1;
23     AdjustedTruncateFlags |= (0b01 & (truncate_flags >> 2));
24     AdjustedTruncateFlags <<= 1;
25     AdjustedTruncateFlags |= (0b01 & (truncate_flags >> 1));
26     AdjustedTruncateFlags <<= 1;
27     AdjustedTruncateFlags |= (0b01 & (truncate_flags >> 0));
28     if (depth < maxDepth) {
29       cilk_spawn _fuse__F16F17F18F19F20_parallel(
30         _r_f0->Content,
31         _f2_ParentFontStyle,
32         AdjustedTruncateFlags,
33         depth + 1,
34         maxDepth);
35     } else {
36       _fuse__F16F17F18F19F20_serial(
37         _r_f0->Content,
38         _f2_ParentFontStyle,
39         AdjustedTruncateFlags);
40     }
```

```
26   } /*Parallel Call 2*/
27   if ((truncate_flags & 0b11111)) /*call*/ {
28     unsigned int AdjustedTruncateFlags = 0;
29     AdjustedTruncateFlags <<= 1;
30     AdjustedTruncateFlags |= (0b01 & (truncate_flags >> 4));
31     AdjustedTruncateFlags <<= 1;
32     AdjustedTruncateFlags |= (0b01 & (truncate_flags >> 3));
33     AdjustedTruncateFlags <<= 1;
34     AdjustedTruncateFlags |= (0b01 & (truncate_flags >> 2));
35     AdjustedTruncateFlags <<= 1;
36     AdjustedTruncateFlags |= (0b01 & (truncate_flags >> 1));
37     AdjustedTruncateFlags <<= 1;
38     AdjustedTruncateFlags |= (0b01 & (truncate_flags >> 0));
39     if (depth < maxDepth) {
40       _r_f0->NextPage->__virtualStub15_parallel(
41         _f2_ParentFontStyle,
42         AdjustedTruncateFlags,
43         depth + 1,
44         maxDepth);
45     } else {
46       _r_f0->NextPage->__virtualStub15_serial(
47         _f2_ParentFontStyle,
48         AdjustedTruncateFlags);
49     }
50   }
51   cilk_sync;
52   return;
53 };
```

Fig. 8. RenderTree code-gen: A fused function parallelized by Orchard. The function contains 2 parallel calls. The "maxDepth" parameter for granularity control is set to 1024.

3.3 Traversal Code Generation

The code generation step in Orchard is similar to the code generation step in Grafter [Sakka et al. 2019]. Figure 8 shows an example of a function generated by Orchard from a tree rendering program. To parallelize the source code, Orchard uses the topological sort (Algorithm 1) to order the vertices (statements or calls) in the graph G. Each vertex in the sorted graph is then written to the corresponding fused function body. This vertex can either execute sequentially or in parallel. For vertices that are not function calls, Orchard writes them to the function sequentially as mentioned in Section 3.2. For vertices that are function calls, Orchard inserts a cilk_spawn (Figure 8 line 29) in front of all calls except for the last call that executes in parallel. It inserts a cilk_sync (Figure 8 line 51) after the last call to ensure dependences do not break.

Orchard provides control over the granularity of parallelism by allowing the programmer to restrict the maximum depth allowed to spawn recursive calls. For instance, let us assume the programmer wants to set the maximum depth to 10. This means that Orchard will only spawn 10 consecutive recursive calls for every function in the generated code. After spawning 10 recursive calls Orchard switches over to the serial version of the code. In order to switch over to the serial code on reaching the maximum depth, Orchard generates replicas of the parallel version of the functions. These replica functions have different function names. They do not have Cilk directives and depth controlling code so that the compiler can distinguish between them. For now,

Orchard uses a fixed threshold depth to enable granularity-control. This enables us to reduce overhead and code complexity. Exploring static or dynamic parallelization granularity could serve as future work.

3.4 Writing traversals in Orchard

Orchard uses Grafter's language to enable programmers to write tree traversal applications. Programmers define their trees using a base node class and subclasses to enable heterogeneity. A programmer can then write general tree traversals in a C++-like language, using virtual functions to provide different behavior for different node types. As described in Section 2, Orchard then uses Grafter's dependence analysis to build a dependence graph representation of the traversals in the program. Crucially, the programmer does not need to worry about the task of enabling parallelism while writing applications. Orchard fully automates it.

4 EVALUATION

We evaluate Orchard on three different benchmarks: ASTs, render trees, and piece-wise functions. For each benchmark, we present a detailed section discussing our results. We show strong scaling, weak scaling, and cache misses for two versions of code generated by Orchard. One parallel version uses the greedy fusion heuristic. The other parallel version, named "solely parallel", is compiled by ignoring the fusion step in Orchard. It is the speedup achieved by parallelism *without* fusion (*i.e.*, traditional Cilk-style parallelism, without Orchard's combination of the two). The speedups are normalized relative to the fused serial version of the code, which is based on Grafter's default greedy fusion heuristic.

Background on Cilk: Cilk [Blumofe et al. 1995] is a runtime system written in C that allows programmers to parallelize their applications within a constant factor of optimal. The performance of a parallel program is accurately described by measures such as a "work" and "critical path" at runtime. The Cilk language extension includes statements like cilk_spawn and cilk_sync. A programmer can spawn child tasks using cilk_spawn, which works like a subroutine call; however, the calling thread can function concurrently with the child task. To wait for child tasks to return, a programmer can use cilk_sync, which causes all threads to join at that point in the program.

Cilk follows the design of a randomized work-stealing parallel scheduler. This allows a *starving* thread to steal spawned tasks from the queue of a *victim* thread. The expected theoretical runtime of a program parallelized using a work-stealing scheduler like Cilk is upper bounded by Equation 1 given the work and span of the program in seconds. In the following equation, T_P is the runtime of the parallel program in seconds. T_w is the work in seconds and T_∞ is the span in seconds. The number of processors a program uses is given by p. We use Equation 1 to get an upper bound on the runtime of our benchmarks and use it to explain performance trends.

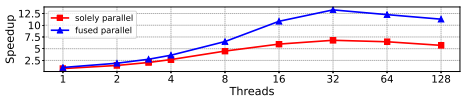
$$T_P = T_w/p + O(T_\infty) \tag{1}$$

Implementation: Orchard* is a Clang-based software tool† that performs source-to-source transformations on input programs by fusing and parallelizing traversals as described in Section 3. To enable parallelism, Orchard inserts Cilk directives such as cilk_spawn and cilk_sync as described in Section 3.3. Orchard uses a greedy fusion heuristic by default; however, the choice of fusion heuristic is open to future investigation.

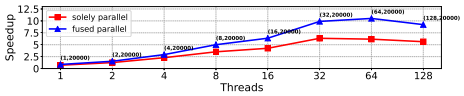
Experimental platform: We evaluate Orchard on an AMD Ryzen Threadripper 3990X with 2.9 GHz clock speed. It has 64 CPUs, each with two threads, totaling 128 threads. The L1d and L1i caches are 32 KB each, the L2 cache is 512 KB, and the L3 cache is 16 MB. We used Opencilk release 2.0, which is based on the LLVM compiler infrastructure and implements Tapir [Schardl et al. 2017]. It provides support for Cilk task-parallel language

*https://github.com/vidsinghal/Orchard

†https://clang.llvm.org/

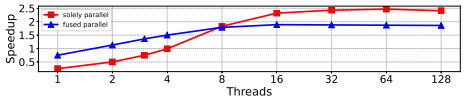


(a) Showing strong scaling(f = 300, s = 20,000). The lowest runtime for fused parallel is 0.052s and solely parallel is 0.080s both at 32 threads.

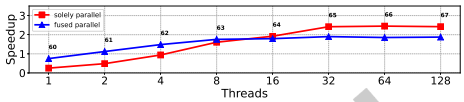


(b) Showing weak scaling. For the input (128, 20000) the runtime for fused parallel is 0.220s and solely parallel is 0.347s.

Fig. 9. Speedups for AST Tree Traversals



(a) Showing strong scaling (n = 78). The lowest runtime for fused parallel is 0.176s at 16 threads and solely parallel is 0.133s at 32 threads.



(b) Showing weak scaling. For input 67, fused parallel runs in 1.257s and solely parallel in 0.971s.

Fig. 10. Speedups for RenderTree benchmark 1

extensions to C/C++ and optimizes Cilk programs to allow for efficient parallel execution on shared-memory multi-core machines. We used `clang++` with `-O3` optimization to generate binaries. We used `perf` to collect cache misses, a Linux-based profiling tool.

4.1 Case Study 1: AST traversals

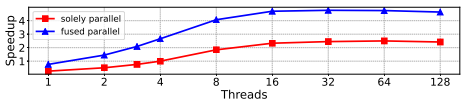
Our first case study looks at an AST traversal benchmark based on Grafter's AST benchmark. It uses a variety of abstract syntax tree passes written in a simple imperative language [Sakka et al. 2019]. The AST tree consists of a variable number of functions containing a variable number of statements set by the programmer. The statements are populated in the function based on a random order containing multiple if, assignment, increment and decrement statements. For evaluating strong scaling and cache-misses, the benchmark generates 300 functions, each with 20,000 statements. The parallelism mainly comes from AST traversals intra-procedurally optimizing multiple functions in parallel.

Orchard gives speedups (baseline greedy fused serial) for both the parallel versions of the AST source code. Figure 9 shows both strong and weak scaling. For strong scaling, we fix the input size but scale the number of threads logarithmically. Whereas, for weak scaling, we scale both the input size and the number of threads. The fused parallel version performs significantly faster than the solely parallel version. We used the Cilkscale tool available with Opencilk to get the work and span times for each benchmark and thread combination as shown in Table 1. We show the approximate runtime of the program calculated based on the randomized work stealing Equation 1. The span is higher for the fused parallel version which is expected—fusion may fuse independent functions which could have otherwise run in parallel—however the overall runtime decreases due to the reduction in work. The span increases as threads increase because multiple starving threads have to compete with each other to perform a steal. Some steals may even be unsuccessful if a victim thread's work queue is empty. This increases communication overhead. We used the Opencilk runtime to obtain statistics for the total steals and saw that the total steals increased by ~3-8 times on average when run on 128 threads relative to 32 threads for the AST benchmark.

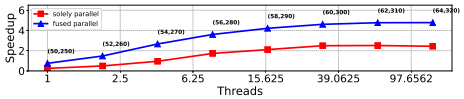
Figure 15(a) shows that the fused parallel version has significantly lower cache-misses than the solely parallel version due to increased locality. Hence, fusion is all to the good. Firstly, it coarsens the granularity of parallelism by fusing multiple functions together. This makes parallelism more efficient by compensating for parallel overheads

Table 1. AST: Showing approximate runtimes for the programs when calculated using Equation 1. For 300 functions and 20,000 statements.

Threads	fused-parallel		runtime (~s)	solely-parallel		runtime (~s)
	Work (s)	Span (s)		Work (s)	Span (s)	
1	6.86	3.25	10.11	7.01	3.32	10.33
2	7.04	3.27	6.79	7.25	3.32	6.95
4	7.19	3.27	5.07	7.40	3.31	5.16
8	7.73	3.27	4.24	7.89	3.32	4.31
16	8.08	3.28	3.79	9.13	3.32	3.89
32	10.17	3.28	3.60	12.59	3.38	3.77
64	19.18	3.36	3.66	24.91	3.52	3.91
128	50.02	3.69	4.08	54.66	3.59	4.02

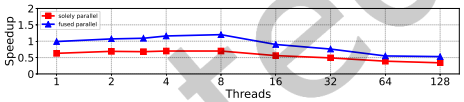


(a) Showing strong scaling (p = 310, n = 62). The lowest runtime for fused parallel is 3.248s and solely parallel is 6.156s both at 32 threads.

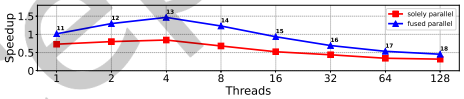


(b) Showing weak scaling. For the input (64, 320) fused parallel runs in 5.0s and solely parallel runs in 9.738s.

Fig. 11. Speedups for Render Tree benchmark 2



(a) Showing strong scaling (depth = 15). The lowest runtime for fused parallel is 0.180s at 8 threads and solely parallel is 0.301s at 2 threads.



(b) Showing weak scaling. For the input size of 18, fused parallel runs in 3.787s and solely parallel runs in 5.262s.

Fig. 12. PieceWise Functions Program 1

and communication costs. Secondly, it decreases overall work without substantially increasing the span. Lastly, fusion increases locality which is shown by the significant reduction in cache-misses.

4.2 Case Study 2: Render Tree

Case study 2 uses Grafter's render tree benchmark [Sakka et al. 2019]. Render Trees are a combination of CSSOM and DOM trees used to render documents on web pages. These render trees are traversed multiple times to calculate visual elements that are present on web pages.

For this case study, we use two types of experiments. The first type creates a document with only one page. This page has n horizontal containers, and each container has n elements with a random number of deeply nested components. For this benchmark, we set n to 78. The second experiment creates a book with p multiple pages; each of these pages is identical to the one in the first experiment. For this benchmark, we set p to 310 pages and n to 62. We refer to the first experiment as the "single page benchmark" and the second as the "multi page benchmark".

Single Page Benchmark: Figure 10 shows the speedups of the two parallel versions of the benchmark. The solely parallel version of the benchmark is faster than the greedy fused parallel version after ~8 threads. However, if a machine has less cores, the greedy fused parallel version will give a lower runtime.

Table 2. Single page benchmark: Showing approximate runtimes for the program when calculated using Equation 1. n is fixed to 78.

Threads	fused-parallel		runtime (~s)	solely-parallel		runtime (~s)
	Work (s)	Span (s)		Work (s)	Span (s)	
1	24.42	12.45	36.87	68.58	10.15	78.73
2	24.41	12.41	24.62	68.80	10.14	44.54
4	24.65	12.51	18.67	71.20	10.07	27.87
8	24.99	12.47	15.59	73.12	10.34	19.48
16	25.36	12.48	14.07	73.97	10.24	14.86
32	26.25	12.56	13.38	77.25	10.37	12.78
64	28.96	12.57	13.02	88.83	10.35	11.74
128	43.15	12.94	13.28	163.30	10.82	12.10

This is an example where fusion is hurting parallelism. From Table 2 we see that although the overall work reduces for the fused parallel version, the dominant term in the runtime as threads increase is the span of the program. The fused parallel version has a higher span, which is expected since this render tree benchmark has a single page with deeply-nested components. The presence of deeply nested components introduces greater dependencies post-fusion. This increases the span as dependencies may prevent functions from executing in parallel.

There is always a balance between fusion and parallelism. Too much fusion may substantially decrease parallelism if the source code has limited parallelism. On machines with fewer threads available, allowing fusion is better. However, when threads are in abundance, the span term may dominate the runtime of the program. In such scenarios, we should allow less fusion. As expected, we observe a reduction in cache misses as shown in Figure 15(b) for the fused version.

Multi Page Benchmark: Figure 11 shows the speedups for the two versions in the case of the multi-page benchmark. The greedy fused parallel version is always faster than the solely parallel version. This is contrary to the single page benchmark. Although fusion reduces opportunities for parallelism on a single page, the presence of multiple pages ensures enough parallelism overall. As shown in Table 3, there in an overall reduction in work after fusion; however, the span doesn't increase substantially. The fused parallel version has lower cache misses, as shown in Figure 15(c).

This is an important result because it shows the benefits of partial fusion in relation to parallelism. Programs with abundant parallelism can benefit from partial fusion even as threads increase. Render tree benchmarks in the real world are like the multi-page benchmark requiring rendering multiple pages. Partial fusion and parallelism show promise for increased speedups in such benchmarks.

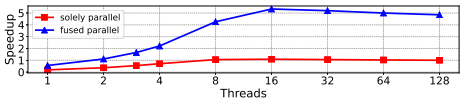
4.3 Case Study 3: Piecewise Functions

Our final case study looks at MADNESS-style kd-tree traversals, as in Rajbhandari et al [Rajbhandari et al. 2016]. This benchmark performs multiple traversals over piece-wise functions stored in kd-trees, such as computing the square, adding a constant, multiplying with a variable, and differentiating a node, among others. We look at three different "programs" that do different sets of operations (i.e., perform different traversals) on the piece-wise functions. When measuring strong scaling and cache misses for program 2, we generate a tree of depth 25. For program 1 and program 3, the tree depth is 15 and 18, respectively.

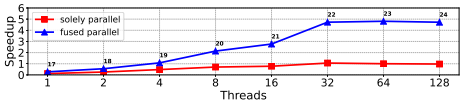
Figures 12, 13 and 14 show the speedups for program 1, 2 and 3 respectively. Table 4 shows the work, span and runtime and Figures 16(a), 16(b) and 16(c) show the cache-misses. The fused parallel version is always faster

Table 3. Multi page benchmark: Showing approximate runtimes for the program when calculated using Equation 1. p is 310 and n is 62.

Threads	fused-parallel		runtime (~s)	solely-parallel		runtime (~s)
	Work (s)	Span (s)		Work (s)	Span (s)	
1	95.05	39.97	135.02	270.89	39.44	310.33
2	95.15	39.69	87.27	265.94	39.09	172.06
4	96.59	40.32	64.47	270.55	39.09	106.73
8	99.98	40.17	52.67	280.12	38.82	73.84
16	110.72	42.89	49.81	282.41	38.91	56.56
32	104.89	39.73	43.01	292.83	38.36	47.51
64	139.62	39.60	41.78	336.66	38.89	44.15
128	208.4	40.18	41.81	573.73	39.26	43.74

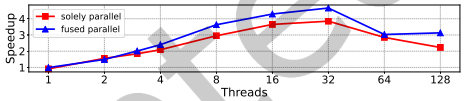


(a) Showing strong scaling (depth = 25). The lowest runtime for fused parallel is 0.021s, and solely parallel is 0.081s, both at 16 threads.

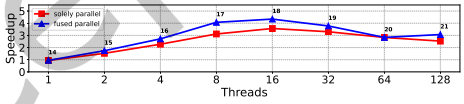


(b) Showing weak scaling. For the input size of 24, fused parallel runs in 0.073s and solely parallel runs in 0.352s.

Fig. 13. PieceWise Functions Program 2

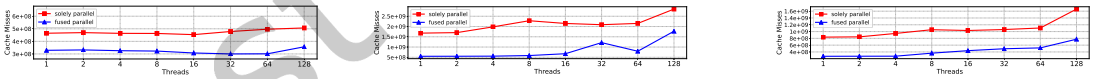


(a) Showing strong scaling (depth = 18). The lowest runtime for fused parallel is 0.178s and solely parallel is 0.220s both at 32 threads.



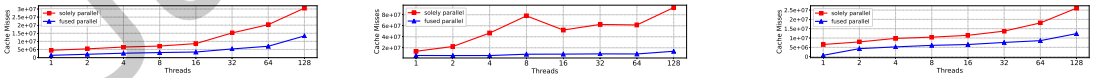
(b) Showing weak scaling. For the input size of 21, fused parallel runs in 2.301s and solely parallel in 2.765s.

Fig. 14. PieceWise Functions Program 3



(a) AST (b) RenderTree single-page (c) RenderTree multi-page

Fig. 15. Cache misses for AST and RenderTree



(a) Program 1 (b) Program 2 (c) Program 3

Fig. 16. Cache misses for PieceWise Functions

than the solely parallel version. This is a consequence of reduction in overall work, coarsening of the granularity which benefits parallelism and lower cache misses.

Table 4. PieceWise functions 1,2 and 3: Showing approximate runtimes for the programs when calculated using Equation 1.

Threads	fused-parallel		runtime (~s)	solely-parallel		runtime (~s)
	Work (s)	Span (s)		Work (s)	Span (s)	
1	0.06	0.002	0.06	0.10	0.002	0.10
2	0.10	0.002	0.05	0.16	0.002	0.08
4	0.18	0.002	0.05	0.29	0.003	0.08
8	0.32	0.002	0.04	0.57	0.003	0.07
16	0.81	0.003	0.05	1.29	0.004	0.08
32	2.17	0.005	0.07	3.31	0.006	0.11
64	6.80	0.009	0.12	7.33	0.010	0.12
128	16.46	0.018	0.15	15.57	0.041	0.16

Threads	fused-parallel		runtime (~s)	solely-parallel		runtime (~s)
	Work (s)	Span (s)		Work (s)	Span (s)	
1	9.11	3.59	12.70	27.69	3.62	31.31
2	8.95	3.55	8.03	27.80	3.58	17.48
4	9.12	3.61	5.89	29.52	3.57	10.95
8	9.71	3.81	5.02	31.52	3.54	7.48
16	9.38	3.70	4.29	30.31	3.55	5.44
32	9.91	3.66	3.97	31.91	3.67	4.67
64	10.95	3.56	3.73	36.69	3.72	4.29
128	15.63	3.64	3.76	59.58	3.80	4.27

Threads	fused-parallel		runtime (~s)	solely-parallel		runtime (~s)
	Work (s)	Span (s)		Work (s)	Span (s)	
1	0.97	0.035	1.01	1.19	0.037	1.23
2	1.22	0.034	0.64	1.53	0.036	0.80
4	1.31	0.037	0.36	1.59	0.036	0.43
8	1.67	0.036	0.24	2.25	0.037	0.32
16	2.73	0.035	0.21	3.34	0.032	0.24
32	4.43	0.040	0.18	5.66	0.040	0.22
64	16.12	0.044	0.30	17.95	0.060	0.34
128	36.71	0.060	0.35	47.11	0.158	0.53

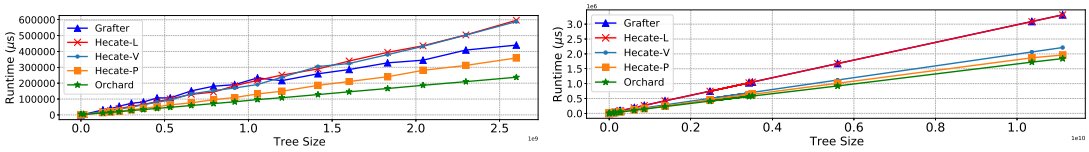
4.4 Comparison with Hecate:

We compare the performance of Orchard with Hecate, the best state of the art prior work in the domain. We used the render tree and AST traversals as in the Hecate paper[‡] and plotted all the implementations as shown in Figure 17[§]. We see that Orchard performs better than Hecate-L (list based traversal), Hecate-V (sequential vector based traversal) and Hecate-P (parallel vector based traversal) for both the benchmarks, especially as the tree size increases. This shows that Orchard scales better than Hecate.

Although Hecate's novelty lies in synthesizing potentially better fusion schedules than Grafter using an SMT solver, Orchard is a complementary approach to automatically parallelize traversals without any manual analysis. Hecate uses symbolic evaluation to generate fusion schedules. The symbolic evaluation is greedy, as it fuses functions whenever possible whenever it visits nodes in the tree. The heuristic that Grafter uses is also greedy, but it is deterministic with one unique solution. Hecate's synthesis technique is non-deterministic as it

[‡]Since Hecate was not evaluated on piecewise functions in the original paper, we did not show it in our evaluation.

[§]We used a 20 thread Intel Core i7-12700K machine with a L2 cache size of 12 MiB and L3 cache size of 25MiB. We had root access on this machine which allowed us to install all the dependencies required for Hecate



(a) Performance comparison for AST. The y axis shows the runtime (μs) and x axis shows the tree-size.

(b) Performance comparison for RenderTree. The y axis shows the runtime (μs) and x axis shows the tree-size.

Fig. 17. Performance comparison amongst Hecate-L, Hecate-V, Hecate-P, Grafter and Orchard

allows for different design choices, such as linked lists vs vectors for traversals, etc. This increases the space of potential schedules at the expense of compile time.

Hecate-P relies on the programmer to expose the parallelism using a vector-based data structure for tree-traversals; this limits parallelism opportunities and puts the burden of finding parallelism hot spots on the programmer. Orchard on the other hand, automatically parallelizes tightly interdependent recursive tree traversals and realizes an efficient runtime performance using Opencilk.

4.5 Discussion

Across these case studies, we can draw several conclusions. Fusion decreases the overall work and simultaneously increases work per function. The former directly reduces the runtime of the program, and the latter is indirectly beneficial since it makes parallelism more efficient. In addition, fusion reduces cache misses by increasing data locality. However, too much fusion may reduce parallelism by increasing the span of the program, which is not ideal. In view of parallelism, the amount of fusion that should be allowed depends on the hardware platform and source code properties. Greedy fusion offers the best performance if the programmer has a single thread. Hardware with a lesser number of threads should allow more fusion before extracting parallelism, as fusion will make parallelism more efficient, and any reduction in parallelism will not be consequential to runtime performance. On the other hand, machines with a greater number of threads should control the amount of fusion to get the best parallel performance from their machine. Similarly, source code with greater parallelism will benefit by allowing more fusion and vice-versa.

On the whole, our benchmarks have shown that partial fusion is better for parallelism, and supporting it can enable the programmer to get greater speedups for their applications. By automating the process of fine-grained fusion and heterogeneous parallelism, Orchard provides the programmer with a novel, sound, and easy way to make these choices. The programmer can control the amount of fusion to get a better parallel run-time for their system.

To test the effectiveness of Orchard's automated approach, we manually fused and parallelized a simple tree traversal program. The original program is written in C++ with ~120 lines of code. The traversals in the program include adding one to values stored in the tree node, adding a constant, and exponentiation on the integer values. It took a proficient C++ programmer ~30 minutes to fuse and parallelize the source code, whereas Orchard accomplished this almost instantly. The original code took ~5 seconds to run, the manually optimized code took ~0.97 seconds, and the Orchard generated code took ~0.96 seconds. This shows that Orchard is not only fast but also efficient at extracting fusion and parallelism. As the complexity (loc, number of functions, etc.) in the source code increases, the delta between Orchard and manual effort will increase drastically. It may also introduce unwanted bugs in the program.

5 RELATED WORK

This paper presents a novel framework that allows programmers to write general tree traversals in an elegant way and generate code that provides fast and efficient performance. Prior works aiming to achieve the goal of making tree traversals more efficient have shortcomings when it comes to extracting the full speedup potential.

Meyerovich *et al.* use attribute grammars to represent tree traversals [Meyerovich and Bodik 2010; Meyerovich et al. 2013]. The attribute grammar representation is used to perform a dependence analysis which is then used to generate a parallel schedule. Their work supports both parallelism and fusion but is incomplete. They do not support partial fusion. Moreover, the parallel schedules are designed to be a homogeneous pre-order or post-order traversal, which is not a fine-grained approach to extracting parallelism. Other works use attribute grammars to specify tree transducers for expressing tree traversals [Doner 1970; Engelfriet and Maneth 2002; Maletti 2008]. However, to our knowledge, we do not know of any tree transducer representation that supports partial fusion and heterogeneous parallelism.

Rajbhandari *et al.* present a compiler framework to automatically enable fusion and parallelism in kd-tree traversals [Rajbhandari et al. 2016; Rajbhandari et al. 2016]. Their fusion approach is limited because they require all traversals to be either pre-order or post-order and not a combination of both. Correspondingly, they only extract parallelism in a homogeneous pre-order or post-order manner; the traversals are either fully parallelized otherwise not.

Chen *et al.* [Chen et al. 2022] develop a synthesizer (Hecate) for tree traversals that is able to exploit serial and parallel traversals. However, they rely on the programmer to identify the inherent parallelism in the source code which is then verified by Hecate. In addition, they do not explore fusion and parallelism as mutually beneficial transformations and how certain fusion choices can make parallelism more efficient. In contrast, Orchard is fully automated.

Functional programming languages use deforestation techniques to make traversals more efficient [Wadler 1990]. Other works use the stream fusion approach to perform fusion transformations on data structures such as arrays, lists, etc. [Coutts et al. 2007]. As functional languages are considered easy to parallelize, functional approaches that enable fusion also inherently enable parallelism. However, both the fusion and parallelism problem in functional languages are fundamentally different than in imperative languages, as they do not have to deal with the state.

Other works target specific areas in imperative programs such as loop fusion and parallelism operating on data structures such as arrays, lists, and matrices [Bondhugula et al. 2008; Darte 1999; Kennedy and McKinley 1993; Qasem and Kennedy 2006]. While they often support fusion and parallelism, they do not apply to tree traversals.

Orchard is a fully automated framework that extends Grafter [Sakka et al. 2019] with its ability to efficiently parallelize tightly inter-dependent recursive functions that traverse trees. In order to parallelize function calls, Orchard inserts Cilk directives using a dependence graph representation of the source code and a static analysis over that dependence graph to analyze dependences. The parallelism extracted by Orchard is more efficient as fusion coarsens the granularity of functions, which makes parallelism more efficient. In addition, fusion may expose more opportunities for parallelism in the face of dependences and it reduces cache misses.

6 FUTURE-WORK

Fusion can make parallelism more efficient. However, fusion prior to parallelization on the dependence graph is not always beneficial as shown in Section 4.2. In future work, we aim to generate an optimal fusion schedule that is optimized for efficient parallelism. To achieve this we look back at Equation 1 and realize that we can optimize the program for the lowest runtime by tuning the work and span parameters. This can be done either statically or dynamic approaches like profiling can be used. For the static technique, we would like to come up with a good cost model that gives a good estimate of the work and span from the dependence graph statically. We can use the

amount of fusion, i.e., the number of functions to fuse and the fusion choices (i.e, which functions to fuse or not to fuse) to transform the work and span of the dependence graph and come up with a near-optimal runtime for the program. In fact, Blumofe et al. [Blumofe et al. 1995] encourage programmers to think in terms of work and span as tunable parameters for optimizing the performance of programs. If we have a principled way to analyze work and span from the dependence graph, the performance of an algorithm can be characterized independent of the machine configuration [Blumofe et al. 1995]. This will make our technique general enough to apply in the case of different machine configurations.

7 CONCLUSIONS

Orchard automates the complex and time-consuming task of extracting fusion and parallelism from recursive tree traversals, including those that traverse heterogeneous trees. Orchard enables both fine-grained fusion and heterogeneous parallelism, allowing it to extract more opportunities for parallelism and fusion than prior work. These two features complement each other, as fine-grained fusion produces the necessary coarser-grained tasks that make parallelism more effective. In addition, fusion reduces overall work and decreases cache misses which reduces the overall runtime of the program. We demonstrated through case studies on ASTs, render trees, and piece-wise functions that Orchard's fine-grained mechanism to extract fusion and parallelism delivers substantially better performance than prior work. As multi-core hardware systems are increasing, the burden of making programs adhere and perform to such systems is directly falling on the programmer. Orchard provides a novel, sound, and efficient approach that ensures opportunities for fusion and parallelism in tree traversal programs are not missed, automating the process of applying these crucial transformations.

ACKNOWLEDGMENTS

We would like to show our gratitude to the anonymous reviewers for providing constructive feedback, which helped steer this paper to its current form. We would also like to thank the authors of Hecate for providing the necessary material for evaluating Orchard against Hecate. This work was supported in part by NSF grants CCF-1725672, CCF-1725679, CCF-1150013, CCF-1439126, CCF-1908504, CCF-1919197, CCF-2216978 and DOE DE-SC0010295.

REFERENCES

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '95)*. Association for Computing Machinery, New York, NY, USA, 207–216. https://doi.org/10.1145/209936.209958

Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 101–113. https://doi.org/10.1145/1375581.1375595

Yanju Chen, Junrui Liu, Yu Feng, and Rastislav Bodik. 2022. Tree Traversal Synthesis Using Domain-Specific Symbolic Compilation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 1030–1042. https://doi.org/10.1145/3503222.3507751

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream Fusion: From Lists to Streams to Nothing at All. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07)*. Association for Computing Machinery, New York, NY, USA, 315–326. https://doi.org/10.1145/1291151.1291199

Alain Darte. 1999. On the Complexity of Loop Fusion. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*. IEEE Computer Society, USA, 149.

John Doner. 1970. Tree acceptors and some of their applications. *J. Comput. System Sci.* 4, 5 (1970), 406 – 451. https://doi.org/10.1016/S0022-0000(70)80041-1

Joost Engelfriet and Sebastian Maneth. 2002. Output String Languages of Compositions of Deterministic Macro Tree Transducers. *J. Comput. System Sci.* 64, 2 (2002), 350 – 395. https://doi.org/10.1006/jcss.2001.1816

Robert J. Harrison, Gregory Beylkin, Florian A. Bischoff, Justus A. Calvin, George I. Fann, Jacob Fosso-Tande, Diego Galindo, Jeff R. Hammond, Rebecca Hartman-Baker, Judith C. Hill, Jun Jia, Jakob S. Kottmann, M-J. Yvonne Ou, Junchen Pei, Laura E. Ratcliff, Matthew G. Reuter,

Adam C. Richie-Halford, Nichols A. Romero, Hideo Sekino, William A. Shelton, Bryan E. Sundahl, W. Scott Thornton, Edward F. Valeev, Álvaro Vázquez-Mayagoitia, Nicholas Vence, Takeshi Yanai, and Yukina Yokoi. 2016. MADNESS: A Multiresolution, Adaptive Numerical Environment for Scientific Simulation. *SIAM Journal on Scientific Computing* 38, 5 (2016), S123–S142. https://doi.org/10.1137/15M1026171 arXiv:https://doi.org/10.1137/15M1026171

Ken Kennedy and Kathryn S. McKinley. 1993. Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, Berlin, Heidelberg, 301–320.

J. R. Larus and P. N. Hilfinger. 1988. Detecting Conflicts Between Structure Accesses. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI '88)*. ACM, New York, NY, USA, 24–31. https://doi.org/10.1145/53990.53993

Andreas Maletti. 2008. Compositions of extended top-down tree transducers. *Information and Computation* 206, 9 (2008), 1187 – 1196. https://doi.org/10.1016/j.ic.2008.03.019 Special Issue: 1st International Conference on Language and Automata Theory and Applications (LATA 2007).

Leo A. Meyerovich and Rastislav Bodik. 2010. Fast and Parallel Webpage Layout. In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*. Association for Computing Machinery, New York, NY, USA, 711–720. https://doi.org/10.1145/1772690.1772763

Leo A. Meyerovich, Matthew E. Torok, Eric Atkinson, and Rastislav Bodik. 2013. Parallel Schedule Synthesis for Attribute Grammars. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 187–196. https://doi.org/10.1145/2442516.2442535

Dmitry Petrashko, Ondřej Lhoták, and Martin Odersky. 2017. Miniphases: Compilation Using Modular and Efficient Tree Transformations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 201–216. https://doi.org/10.1145/3062341.3062346

Apan Qasem and Ken Kennedy. 2006. Profitable Loop Fusion and Tiling Using Model-Driven Empirical Search. In *Proceedings of the 20th Annual International Conference on Supercomputing (ICS '06)*. Association for Computing Machinery, New York, NY, USA, 249–258. https://doi.org/10.1145/1183401.1183437

S. Rajbhandari, J. Kim, S. Krishnamoorthy, L. Pouchet, F. Rastello, R. J. Harrison, and P. Sadayappan. 2016. A Domain-Specific Compiler for a Parallel Multiresolution Adaptive Numerical Simulation Environment. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 468–479.

Samyam Rajbhandari, Jinsung Kim, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, Robert J. Harrison, and P. Sadayappan. 2016. On Fusing Recursive Traversals of K-d Trees. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*. Association for Computing Machinery, New York, NY, USA, 152–162. https://doi.org/10.1145/2892208.2892228

Laith Sakka, Kirshanthan Sundararajah, and Milind Kulkarni. 2017. TreeFuser: A Framework for Analyzing and Fusing General Recursive Tree Traversals. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 76 (Oct. 2017), 30 pages. https://doi.org/10.1145/3133900

Laith Sakka, Kirshanthan Sundararajah, Ryan R. Newton, and Milind Kulkarni. 2019. Sound, Fine-Grained Traversal Fusion for Heterogeneous Trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 830–844. https://doi.org/10.1145/3314221.3314626

Dipanwita Sarkar and Oscar Waddell. 2005. A Nanopass Framework for Compiler Education.

Tao B. Schardl, William S. Moses, and Charles E. Leiserson. 2017. Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. Association for Computing Machinery, New York, NY, USA, 249–265. https://doi.org/10.1145/3018743.3018758

Philip Wadler. 1990. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science* 73, 2 (1990), 231 – 248. https://doi.org/10.1016/0304-3975(90)90147-A

Yusheng Weijiang, Shruthi Balakrishna, Jianqiao Liu, and Milind Kulkarni. 2015. Tree Dependence Analysis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 314–325. https://doi.org/10.1145/2737924.2737972

Ben Wiedermann and William R. Cook. 2007. Extracting Queries by Static Analysis of Transparent Persistence. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. ACM, New York, NY, USA, 199–210. https://doi.org/10.1145/1190216.1190248