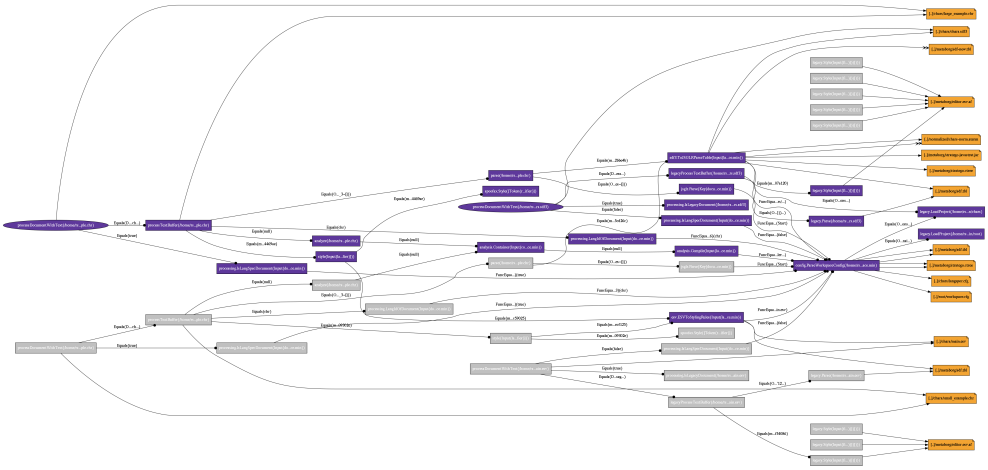
Task Observability in change driven incremental build systems with dynamic dependencies

Version of August 21, 2019



Roelof Anton Sol

Task Observability in change driven incremental build systems with dynamic dependencies

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Roelof Anton Sol
born in Eindhoven, the Netherlands

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Cover picture: Spoofax Build pipeline for the 'char' language.

Task Observability in change driven incremental build systems with dynamic dependencies

Author: Roelof Anton Sol
Student id: 4012194
Email: r.a.sol@student.tudelft.nl

Abstract

Abstract

Context Updating an old result by selective re-execution of the inconsistent parts of some computation is usually faster than recomputing everything. Incremental build systems and interactive development pipelines use this technique to speed up feedback. They consist of different tasks. These tasks form a graph by depending on the environment and the result of other tasks. To re-execute a build requires an incremental build algorithm to find and re-execute inconsistent tasks. This can be done by traversing the dependency graph top down. When the mutations to the input environment are known in advance a build algorithm can avoid graph traversal. Thereby scaling with the size of a change instead of the size of the graph. Another important distinction between incremental build systems is their ability to handle dynamic dependencies. That is, dependencies that are discovered during a build. PIE is a bottom-up build algorithm that supports dynamic task dependencies. It schedules and executes inconsistent tasks without traversing the entire dependency graph. The current PIE algorithm is capable of adding dynamic task dependencies. However, it does not process removing dynamic task dependencies. This preserves consistency, but limits scalability over time. Each detached task is still scheduled and executed by the bottom up algorithm.

Inquiry PIE is inefficient when it executes tasks that are no longer a transitive dependency. In this paper we introduce task observability to solve this. This problem is unique to bottom up scheduling build systems. However, we are able to re-use techniques from incremental build systems and garbage collections to implement our solution.

Approach We split the problem into three parts, determining task observability, scheduling, and re-observing.

Knowledge With the new algorithm we are able to improve the efficiency of PIE and its scalability over time.

Grounding We verify and benchmark our changes with two artificial and one real world use case in the Spoofax Workbench.

Importance The PIE runtime is able to continue operating efficiently even when tasks are detached. In some situations the Spoofax PIE pipeline reduces the feedback time with 1800 ms when results are not observed by the user. Additionally, new design patterns are made possible. For Spoofax,

toggling observability creates the opportunity to implement various quality of life features such as file and project renaming.

As a secondary contribution we implement a garbage collector for detached tasks and create a visualization tool for displaying the dependency graphs stored in PIE.

Thesis Committee:

Chair:	Prof. dr. E. Visser, Faculty EEMCS, TU Delft
Committee Member:	Dr. R. Krebbers, Faculty EEMCS, TU Delft
Committee Member:	Dr. A. Katsifodimos, Faculty EEMCS, TU Delft

Preface

A special thanks to Gabriël Konat for his guidance, weekly discussions, proofreading, and help with the implementation and benchmarks of the improved algorithm. Thanks to Sebastian Erdweg for proposing this topic. Thanks to Eelco visser for his feedback and all the lunches. At time of writing a part of this thesis is setup to be submitted in article form for review to the 2020 <Programming> language conference.

Roelof Anton Sol
Delft, the Netherlands
August 21, 2019

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software projects are composed of smaller pieces that must be combined to form a whole. This process includes compilation of multiple source files and linking them into an executable. A system used for automating these steps is called a *build system*. However, build systems are not limited to compilation; they can combine any automated tasks.

Development of a small piece is done in isolation before it is used in another task. For example a 'publish' task requires building the program, building documentation, and sending the result somewhere. The developer builds the program any number of times before publishing it. When the developer decides to run the publish task, the build system checks if it needs to rebuild the program or if the old result is still up to date. A build system that reuses the results of intermediate steps is an *incremental build system*. Make is a well-known example(Richard M. Stallman 2016).

Integrated Development Environments (IDEs) use incremental build systems to power their *interactive software development pipelines* (Konat, Erdweg, and Visser 2018), These pipelines are used in software such as spreadsheets, rendering pipelines, and compilers. The term build system is used throughout this thesis to refer to both incremental build systems and interactive software development pipelines.

In the case of a Software development IDE, the intermediate results created during compilation are of value to the user. Compilation might be split up into parsing, type-checking, and code generation. An incremental build system allows the IDE to reuse these intermediate results as shown in Figure 1.1. The result of the parser is reused for syntax highlighting, the result of the type checker is reused to provide type hints to the user. These uniquely addressable and reusable steps are referred to as *tasks*. Tasks return a result and have other tasks and files as *dependencies*.

Figure 1.1 is a dependency Directed Acyclic Graph (DAG) representation of the execution of a build. Code Highlight is a task and depends on the Parse task and the content of a color file. A result of a task is reused while its dependencies are *consistent*. When a dependency of a task becomes inconsistent, the task itself becomes inconsistent. A dependency must first be made consistent and the task re-executed to produce a new result.

In general, build systems should be fast and easy to use. Konat, Erdweg, and Visser (2018) identified three specific properties that an incremental build system should have:

- **Scalability**[1]: The time it takes to bring the system into a consistent state is proportional to the size of the change, and not proportional to the number of tasks. A change that affects only a few things should take a short time, even in a large project
- **Precision**: An incremental build system is only effective if its dependencies are precise. When dependencies are under-defined, the system might not pick up on a change

[1] Referred to as Efficiency in (Konat, Erdweg, and Visser 2018)

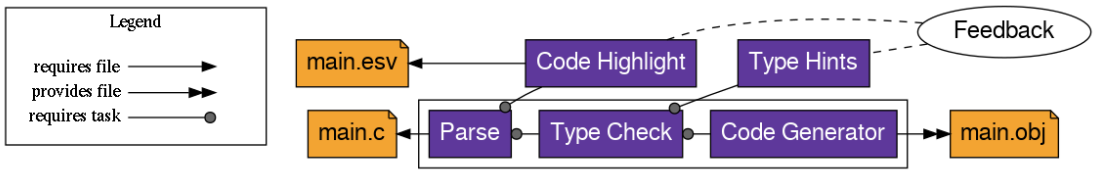


Figure 1.1: Dependency graph of an execution trace for an IDE compilation build script. A C file is compiled into an object file, while providing syntax highlighting and type hint for to the developer.

which produces an incorrect result compared to a non-incremental build. At the other end, when dependencies are overly broad a build is slower than necessary

- **Expressiveness**: The method by which a task is specified should be simple and powerful. Build systems grow along with their projects (McIntosh, Adams, and Hassan 2010) and require maintenance (Epperly 2002). Therefore, a build script must be simple to read and write. Accidental complexities, such as forcing the order of recursive Makefiles by some implicit mechanism Miller n.d., should not be an idiomatic solution.

Expressiveness has not been a focus for most incremental build systems. Specifically, two common patterns in build scripts are hard to express in most incremental build systems. A *conditional build* where the task might be included based on the result of another task, and an *iterative build* where a task is executed for different inputs and the result is processed within the loop. These patterns create *dynamic dependencies*.

Consider a hypothetical ***checkProjectStyleWarnings*** task, which must read a configuration file which defines the maximum of number style warnings allowed in the code. In addition, the configuration file contains a set of files that must be excluded from the total. This specification requires dynamic dependencies and it is not obvious how to express it in Make. Patterns such as recursive Makefiles or additional rule syntax fail at our expectation of expressiveness when implementing these requirements; consequently, it is not uncommon to sacrifice precision by changing the requirement, such as deciding to fail on any style warning.

The straightforward solution to expressiveness and precision is to use a full fledged programming language. PIE (Konat, Steindorfer, et al. 2018) is a framework for developing incremental build systems, which includes such a programming language. The PIE language has a focus on expressiveness that supports dynamic dependencies with familiar constructs such as 'if', 'for', local variables, and special syntax to access the file-system. Listing 1.1 shows an implementation of ***checkProjectStyleWarnings*** in the PIE language. The code precisely fits the requirements and is familiar to most programmers. In this example the iterative building of the checkCodeStyle for each file and the condition that a file is not in the ignoreStyleWarnings variable create dynamic dependencies.

PIE code is compiled and executed in the PIE runtime. The PIE runtime maintains a dependency DAG similar to Figure 1.1 in which it stores the latest results and dependencies of tasks. The runtime is responsible for providing the result of a task. Either from (persistent) memory if it is consistent, or through re-execution of inconsistent tasks. This process requires a build algorithm.

A straightforward *top down* incremental build algorithm works as follows: Initially the dependency graph is empty. The task must be executed. When it requests a dependency the execution is suspended and the dependency is first executed. The results and the dependency relation are saved. When a task is requested for a second time, the runtime first checks if any of its direct or transitive dependencies have has been modified. If no dependency has been modified it returns the stored result, otherwise it must be re-executed.

Listing 1.1: A snippet of PIE code with dynamic dependencies.

```
taskdef checkProjectStyleWarnings() -> String  {
  val config = parseYaml('./config.yaml')
  val files = './src/**.c'
  for (file <- files) {
      if ( !config.ignoreStyleWarnings.contains(file) ) {
        val warnings = checkCodeStyle(file)
        if ( warnings > config.maxStyleWarnings ) {
          return "Too many errors"
        }
      }
   }
   return "Ok"
}

taskdef checkCodeStyle(p: Path) -> Int {...}
taskdef parseYaml(p: Path) -> Yaml {...}
```

This strategy requires checking the entire dependency graph for inconsistent tasks. Konat, Erdweg, and Visser (2018) described and implemented an alternative *bottom up* build algorithm for incremental build systems with improved scalability. The algorithm takes as input a dependency graph and a set of modified files and returns a graph with all affected tasks in a consistent state. By only processing affected tasks its execution time is independent of the number of tasks in the system.

The bottom up algorithm proceeds as follows. It creates a set to hold the inconsistent tasks. It fills it with the tasks directly affected by a changed file. While this set is not empty, it removes a task based on the topological order. That is, a task that has no transitive dependencies to any task in the set of inconsistent tasks.

During execution the task may dynamically require the result of other tasks. The runtime provides these through a mix of bottom up and top down execution. Finally, after the task has been executed, its result and dependencies are saved and affected tasks which depend on it are added to the set. This process continues until the set is empty. A more in depth explanation and example is described in chapter 2.

The bottom up algorithm is an improvement in general, however it slows down after continuous use. It does not take into account when a task is no longer a dependency of any other task or requested by the user. That is, when it becomes unobserved. Instead it considers all tasks observed and thus schedules and executes them. This produces correct results, but is inefficient.

Consider Listing 1.1. When a file is added to the `ignoreStyleWarnings` list it is removed as a dependency. In the bottom up algorithm, this dependency is not removed and a change in that file will schedule `checkProjectStyleWarnings` even though it can not be affected.

Another case can be demonstrated with Figure 1.1. Suppose we are temporally only interested in Code Highlight. The tasks for Type Check, Type Hints, and Code Generator are unobserved and executing them is unnecessary. The current algorithm has no mechanism to suspend execution for these tasks while the user is not interested.

A naive solution for detecting unobservable tasks starts by traversing the dependency graph top-down after a rebuild and marking all the reachable tasks. The unmarked tasks are then no longer in the transitive closure. However, this approach requires visiting every node in the graph and undermines the scalability improvements of the bottom up algorithm over

the top down algorithm.

This paper improves the bottom up algorithm in terms of speed and memory efficiency. Its main contribution is an improvement to the bottom up algorithm that tracks the observability state of a task during execution such that unobserved tasks do not have to be executed, and a method to re-use old results when tasks become observed again (Section 4). The key idea is the following: A user sets a number of explicitly observed tasks. Observability of other tasks is efficiently updated when a dependencies are added and removed. When a task is executed it is marked observed. When a dependency is removed, the dependency and its sub-graph is processed to mark newly unobserved tasks. Next, unobserved tasks are not scheduled. Finally, when an unobserved task is required again, the current state is considered inconsistent. The task is then rebuild in a similar manner to the incremental top down algorithm.

First, we track the transitive closure of a set of explicitly observed tasks. When a dependency is removed, the observability state of the tasks in its sub-graph is updated by counting the number of times the task is referenced as a dependency of an observed task. Secondly we do not schedule tasks that are detached from all observed tasks. First, we track the transitive closure of explicitly observed tasks during execution. Secondly, any task not observed is not scheduled. Finally, when a task is re-observed it is considered inconsistent. It is then rebuild similar to the incremental top down algorithm.

We benchmark it in two artificial scenarios and one real world case study (Section 5). The two artificial scenarios allow us to evaluate our modifications with respect to correctness and efficiency. We run these scenario's twice, once such that task execution dominates the build time[2], and once such that the runtime and build algorithm dominates the build time. The real world case study is the Spoofax Langauge Workbench using the PIE build script. Our tests show that it for some situations the improved algorithm saves 1800 ms. Additionally, the improved algorithm is able to keep operating efficiently when tasks are detached. This allows for various new designs, such as file renaming in the Spoofax pipeline.

During our research we also implemented a garbage collector and a dependency graph visualization tool (Section 6). The garbage collector works by combining the observability state with an extension to the task definitions indicating if the task is suitable to be cleaned up. This avoids cleaning unobserved tasks which are expensive to compute. The dependency graph visualization tool allows us to inspect the dependency graphs of the PIE runtime during execution. It has been invaluable when discussing solutions and it has been successfully used to find bugs in earlier implementations and our benchmarks.

In summary, We review PIE and show the effects of unobservable tasks and the scalability issue with the bottom up algorithm (Section 2). We describe our key ideas for tracking observability and building unobserved tasks (Section 4). We create two types of artificial dependency graphs. We validate and evaluate our new algorithm using these graphs. Next we evaluate our improvements with a case study of the Spoofax PIE build system (Section 5). We show two secondary contributions. A simple garbage collection algorithm using the new observability state, and our setup and experience with a dependency graph visualizer (Section 6). We discuss the related work with respect to build systems and Incremental Computation (IC) in general (Section 7). Finally, we propose future improvements for the execution algorithms and potential improvements to the PIE language and runtime (Section 8).

[2]This is the usual case for build systems

Chapter 2

Background

PIE is an incremental build system developed at the TU Delft. We will use its terminology throughout this paper. PIE consists of the PIE language (Konat, Steindorfer, et al. (2018)) and the PIE runtime. The PIE language is used to create build scripts. Build scripts are executed by the runtime. There are multiple methods to execute an incremental build. The PIE runtime implements two of these incremental build algorithms. A top down algorithm modeled after PLUTO (Erdweg, Lichter, and Weiel 2015), and a bottom up algorithm (Konat, Erdweg, and Visser 2018). This paper improves on the scalability of the bottom up algorithm.

In this chapter we start by introducing PIE with an example program and explain the terminology we will use throughout the rest of this paper. We explain the two build algorithm and discuss the bottom up strategy in depth with an example execution.

2.1 PIE Build Script by Example

To explain the core concepts of PIE we define a single recursive task definition `calcSum` (Listing 2.1). The goal of this task definition is to calculate the sum of the entries in a list. Each entry is a number or a filename. In case it is a number, its value is used. In case of a filename, the task will require the result of `calcSum` on this file. In our example we will use four input files. Their content and the result of `calcSum` is given in Table 2.1. The dependency graph generated for `calcSum(delta)` is shown in Figure 2.2.

Listing 2.1: The `calcSum` task definition. It reads a file and sums up every entry. An entry is either a number or a reference to another file

```
1  taskdef calcSum(sourcePath: Path) -> Integer {
2    val text = require(sourcePath)
3    val sum = 0
4    for (entry in text.lines() ){
5      try {
6        sum = sum + parseInt(entry);
7      }
8      catch ( e : ParseError ){
9        sum = sum + calcSum(entry)
10     }
11   }
12   return sum;
13 }
```

Table 2.1: An example of four files; Their names, the content set by the user, and the result of calling `calcSum` (Listing 2.1) with the file name as argument.

file name	alpha	beta	gamma	delta
content	1	2 alpha	3 alpha	beta gamma
`calcSum`	1	3	4	7

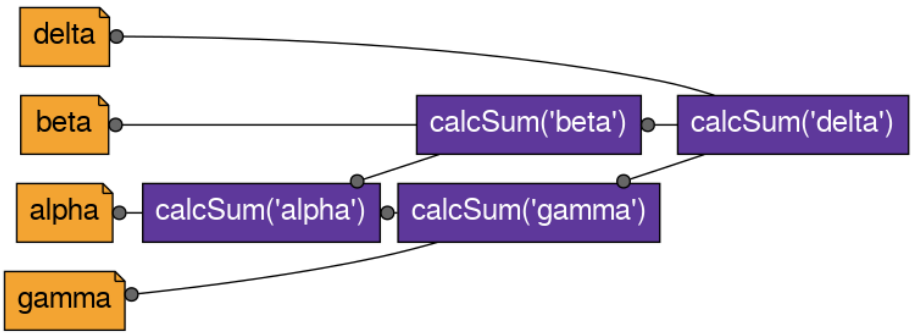


Figure 2.2: The dependency graph created by calling calcSum('delta') (Listing 2.1) on the files described in Table 2.1.

The user of PIE defines one or more *task definitions* in the PIE Language such as `calcSum`. This definition is compiled and registered with the runtime. The task definition together with an input creates a task. A task represents the application of a (impure) function. They are executed by the runtime and return a result. The result is (re)used by other tasks. In Listing 2.1 the *task definition* `calcSum` creates four distinct *tasks*, each with a different argument. The runtime keeps track of each task by its task-key. This key uniquely identifies the task. By default, the key is a combination of the task definition and its input. Alternatively, a custom task-key can be set to avoid using large inputs as keys.

The dependency graph starts empty and is modified during execution. The runtime records when a task *requires* the result of a task or content of a file. Specifically, the call `calcSum(entry)` (Line 9) in our example is transformed in to the call `require(calcSum,[entry])`. The `require` function is provided by the build algorithm. Furthermore, the runtime records when a task *provides* a file as shown in the introduction in Figure 1.1.

Additionally, a task may also execute an external tool. In this case it must explicitly notify the runtime about the files it requires and provides. When one task provides a file and another task requires it; there must exist a dependency from the requiring task to the providing task.

A build algorithm is responsible for creating a *consistent* dependency graph. The dependency graph is consistent if the result of each task is the same as a *non-incremental* execution would produce. That is, straightforward evaluation without reusing results. Consistency is a transitive relation. All tasks and files in a transitive closure of a task must be consistent before the task itself can be made consistent. This relation forms the basis of incrementality. An execution algorithm can reuse a result if it is able to determine that all the dependencies are consistent. Generally, the difference between build algorithms are: the input it requires, the method and order by which it determines consistency, and the order in which tasks are scheduled and executed.

To determine consistency with a previous execution the runtime can compare an entire file or result. As an optimization, the runtime creates and compares stamps. Commonly this is the file access time or the a hash. Alternately, a user can provide a custom method. For

instance a stamp method that reads a single line from a file. When a task is executed, the runtime stores the stamps together with the dependency edge.

For sound execution, the following must hold:

- First, the graph must remain acyclic. A cycle in the graph is the equivalence of an infinite loop during execution.
- A file can only be provided by a single unique task.
- When a file is provided by a task, any other task that requires the file must also require the providing task.

These invariants are checked during execution and a descriptive error message is returned when one is violated. Similar to previous work, we address the mutable environment as 'files'. However, this generalizes to other resources, such as folders, network resources, and user supplied variables.

2.2 Top Down Execution

The first build algorithm used by PIE was a modified version of the PLUTO algorithm (Erdweg, Lichter, and Weiel 2015). We refer to this as the Top Down algorithm. In essence, it starts a depth first search for inconsistent tasks and rebuilds them. The Top Down algorithm provides a `require` function that works as follows: If the task doesn't exist or its result has been removed, it is executed. Otherwise, `require` determines if the task is consistent with the current state of its dependencies. To determine if the files it requires and provides are consistent, it computes a new stamp of the current file and compares it to the old. To determine if the task dependencies are consistent, it invokes `require` to get the latest consistent result and then computes a stamp for it. Note that `require` is invoked recursively for tasks. If any stamp is inconsistent, the task is re-executed. When a task is re-executed, the stamps for its dependencies are updated with their new result. Finally, `require` adds the task to the set of visited tasks. Other tasks will first check this set when they `require` a result. Therefore, task consistency is only checked once per build. Top down building is initiated by the user. The user invokes a function that clears the visited set and then invokes the top down `require` function.

2.3 Top Down Scalability

This method allows for incremental execution and dynamic dependencies but the algorithm must call `require` for every task in the transitive closure of a dependency to determine if the previous result is consistent. In other words, *the algorithm scales with the size of the graph, not the size of the change* As an example, consider Figure 2.3. The file `alpha` contains a long list of filenames. When only file `b` changes, the top down algorithm must validate every task and file. It is clear that executing task `calcSum(b)` and then task `calcSum('alpha')` using the old graph creates a consistent result as well. Although the order of re-execution seems predetermined in this case, in general due to dynamic dependencies the correct execution order is unknown.

2.4 Bottom-Up Execution

The bottom up algorithm is a build algorithm created to overcome the top down scalability issue. It requires the set of changed files and an existing dependency graph to bring the system in to a consistent state. With this information it executes tasks (transitively) affected

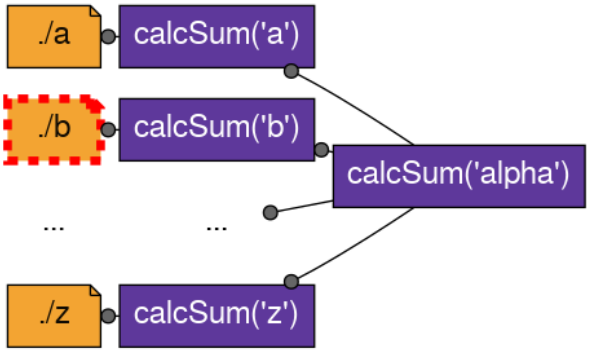


Figure 2.3: The top down algorithm will visit every task and file. Ideally, the build algorithm scales with the size of a change. That is, after the file b has changed, only task `calcSum('b')` and `calcSum('alpha')` are inconsistent and must be visited.

Listing 2.2: `require` and `exec` for the bottom up algorithm.

```
function exec(t)
  if t ∈ T_e then abort;
  T_e := T_e ∪ t; valr := t.run(); T_e := T_e\t
  DG := DG ∪ r ; validate(t, r); notify( t, r .output )
  O_c[t] := r.output ; return r.output

function require(t)
if t ∈ O_c then return O_c[t]
  if ¬DG.hasOutputOf(t) then return exec(t)
  return requireScheduleNow(t)

function requireScheduledNow(t)
  while val t_min := T.q.leastDepFromOrEq(t) do
    T_q := T_q\t_min
    val o := execAndSchedule(t_min)
    if t = t_min then return o
  val o := DG.outputOf(t)
  O_c[t] := o
  return o
```

by the changes in the environment. Because unaffected tasks are never considered, the bottom up algorithm scales with the number of tasks affected by a change instead of the size of the dependency graph. We explain the algorithm and show how this would operate on our example in chapter 2.5.

The bottom up algorithm consists of two public functions shown in Listing 2.4. The first, `buildNewTask`, is required to build an initial dependency graph. The second, `buildWithChangedFiles`, is used in conjuncture with a set of changed files to bring the dependency graph up to date.

Both initialize the variables T_e and O_c for a build. The variable T_e is a set of tasks that are currently being executed, the variable O_c is the results of tasks already executed during this build. Finally, the variable DG contains stamps and dependencies of the tasks and files forming the dependency graph.

Both methods will `execute` a task. When a task is executing it retrieves the result of another task or file with the `require` function (Listing 2.2). That is, in our example `calcSum(line)`

Listing 2.3: Functions used to Schedule tasks that are inconsistent.

```
function schedAffByFiles(F)
  for f ← F do
    for (stamp, t) ← DG.requireesOf(f) do
      if ¬stamp.isConsistent(o) then
        T_q := T_q ∪ t
    for (stamp, t) ← DG.generatorOf(f) do
      if ¬stamp.isConsistent(o) then
        T_q := T_q ∪ t

function schedAffCallersOf(t, o)
  for (stamp, t_call) ← DG.callersOf(t) do
    if ¬stamp.isConsistent(o) then
      T_q := T_q ∪ t_call

function execAndSchedule(t)
  val r := exec(t)
  schedAffByFiles(t,r.genFiles)
  schedAffCallersOf(t,r.output)
  return r.output
```

Listing 2.4: The entry points to initiate bottom up execution.

```
var T_q; var T_e; var O_c; var DG

function buildNewTask(t)
T_q := ∅; O_c := ∅;
exec(t)

function buildWithChangedFiles(F)
T_e := ∅; O_c := ∅;
T_q := new PriorityQueue(DG.depOrder())
schedAffByFiles(F)
while T_q ≠ ∅ do
  execAndSchedule(T_q.poll(), DG)
```

will invoke `require(calcSum, [line])`. In an initial build, require recursively executes a task or retrieves its value from O_c.

The function `buildNewTask` which initializes a graph is similar to the top down algorithm. Executing a task will recursively invoke invokes `require` and `exec` in order to run the build and register stamps and dependencies in the dependency graph DG.

For bottom up building the process is more involved. `buildWithChangedFiles` is its entry point. This function adds all tasks affected by a set of changed files to T_q.

Next it continuously processes tasks from T_q. The queue is topologically ordered with respect to the *current* dependency graph. The first task is removed and made consistent with `execAndSchedule`. `execAndSchedule` will execute the task and then schedule all tasks affected by its result or files that it has written. This scheduling is done with `schedAffCallersOf` and `schedAffByFiles` respectively (Listing 2.3).

When the task is executing, it will `require` other tasks and files. If dynamic dependencies are not supported this process is straight forward. Because all inconsistent tasks are in T_q and the current task has no transitive dependencies on T_q, all its dependencies are currently up to date and can be used. However, in the context of dynamic dependencies the `require` function is more complex. When a task executes and adds a new dependency, it might create a transitive dependency on a task scheduled in T_q. To deal with this situation `require` uses `requireScheduledNow`. This method will find and execute tasks in the transitive closure of the required task. That is, it will only execute tasks in T_q that might affect the required tasks. One interpretation is that `requireScheduledNow` is a bottom up build, similar to `buildWithChangedFiles`, that only considers the transitive closure of the required task in question.

The user of PIE can add callback functions to notify a GUI or other program when the result of a task changes. This is done by the `notify` function in our code.

2.5 Bottom Up by example

We continue the `calcSum` example where we left off in Figure 2.2. The files for alpha and gamma are modified so that `calcSum(gamma)` creates a new dependency on `calcSum(beta)` (Table 2.4). The example demonstrates how `requireScheduledNow` takes a value from T_q.

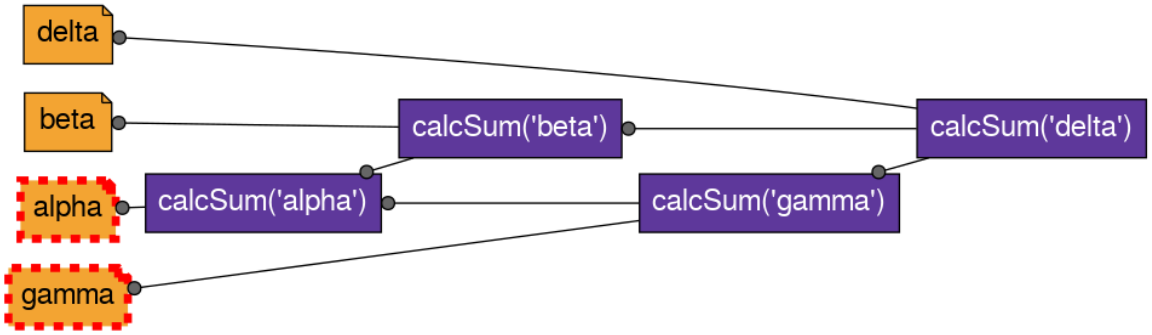
- Figure 2.5a) : T_q=[]
  The example begins by the user invoking `buildWithChangedFiles({alpha,gamma})`.
- Figure 2.5b : T_q=[`calcSum(alpha)`,`calcSum(gamma)`]
  `schedAffByFiles({alpha, gamma})` adds `calcSum(alpha)` and `calcSum(beta)` to T_q .
- Figure 2.5c : T_Q=[`calcSum(gamma)`,`calcSum(beeta)`]
  `calcSum(alpha)` is in front of the queue because of its topological order w.r.t. `calcSum(gamma)`.
  `schedAffCallersOf(calcSum(alpha))` schedules `calcSum(beta)`.
- Figure 2.5d : T_Q= [`calcSum(gamma)`,`calcSum(beta)`]
  No order exist between `calcSum(beta)` and `calcSum(gamma)` yet.
  First `calcSum(gamma)` is executed. During execution, it requires `calcSum(beta)`.
- Figure 2.5e : T_Q =[`calcSum(gamma)`,`calcSum(delta)`]
  This invokes `requireScheduledNow(calcSum(beta))`. `calcSum(beta)` is in its own transitive closure so it is removed from T_q and `execAndSched(calcSum(beta))` schedules task `calcSum(delta)` .
- Figure 2.5f T_Q = [`calcSum(delta)`]
  The execution of `calcSum(gamma)` finishes and only `calcSum(delta)` is left in T_q . The modification to `gamma` adds a dependency to `calcSum(beta)`.

Table 2.4: The content of each file and the result of `calcSum`. Changes compared to Table 2.1 are underlined.

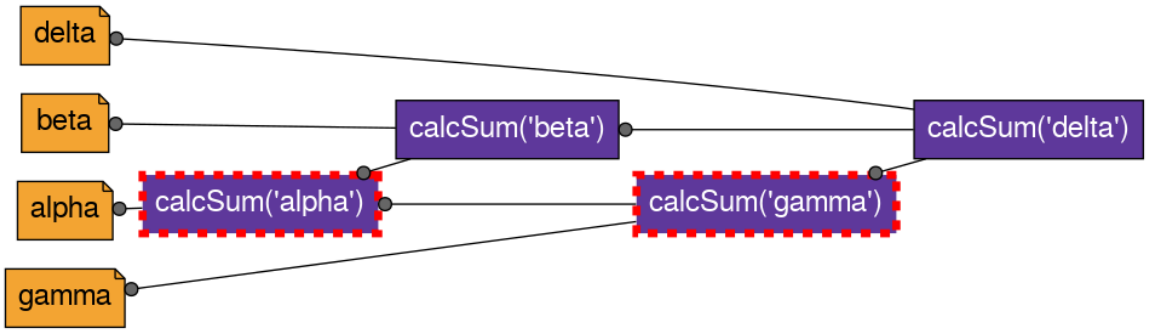
file	alpha	beta	gamma	delta
content	2	2 alpha	3 alpha beta	beta gamma
`calcSum`	2	4	9	13

After this task is executed the entire graph is consistent.

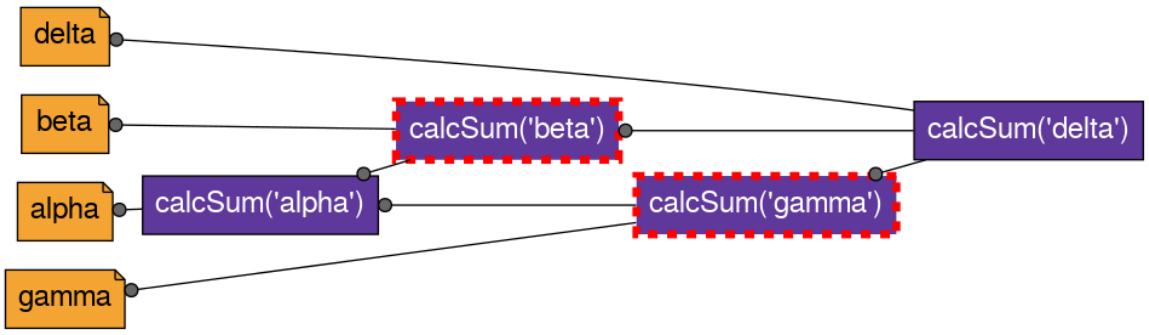
(a) T_q = []
Files alpha and gamma are modified.



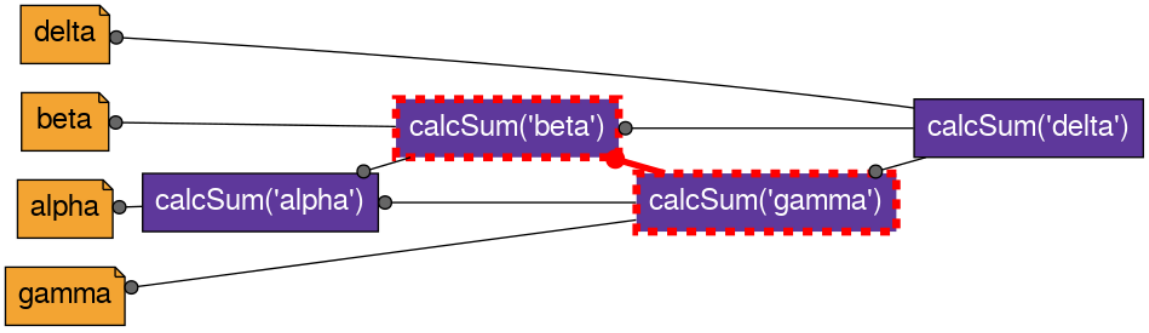
(b) T_q = [`calcSum(alpha)`,`calcSum(gamma)`]
The affected tasks are scheduled.



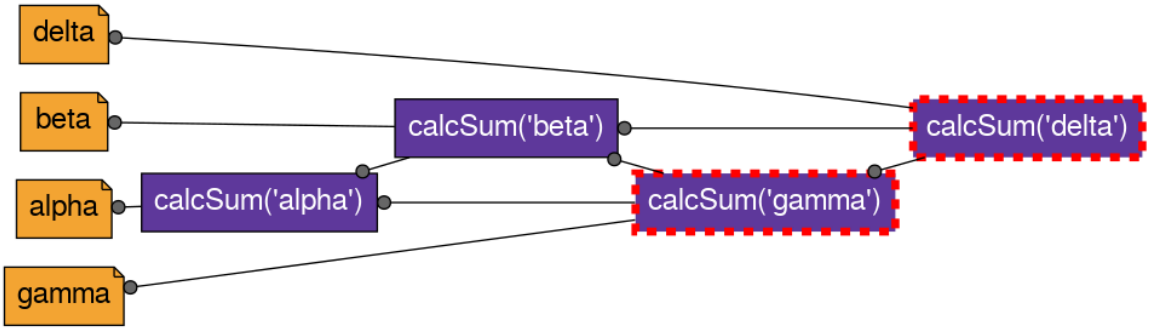
(c) T_q = [`calcSum(gamma)`,`calcSum(beta)`]
`execAndSched(calcSum(alpha))` has scheduled `calcSum(beta)`



(d) T_q := [`calcSum(gamma)`,`calcSum(beta)`]
`calcSum(gamma)` adds a dependency. `requireScheduledNow(calcSum(beta))` determines that `calcSum(beta)` is in T_q and must be executed.



(e) T_q : [`calcSum(gamma)`,`calcSum(delta)`]
`execAndSchedule(calcSum(beta))` has scheduled `calcSum(delta)`.



(f) T_q : [`calcSum(delta)`]
`requireScheduledNow(calcSum(beta)` returns with the result. `calcSum(gamma)` finishes.

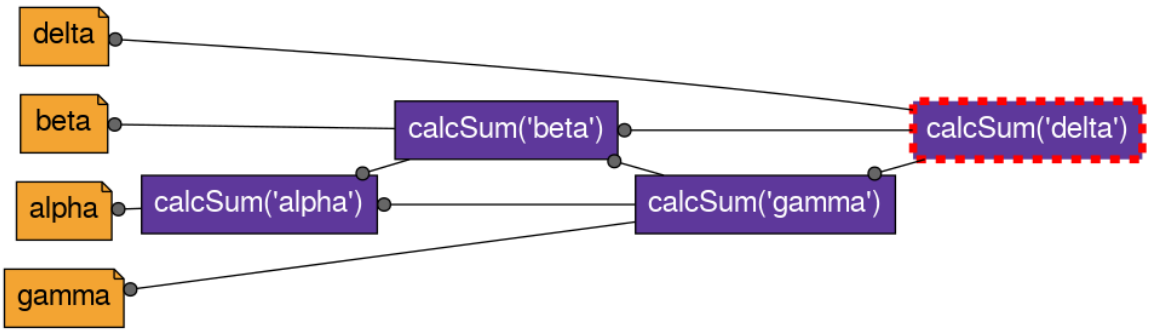


Figure 2.5: A bottom up build with changed files alpha and gamma (Table 2.4) showing affected tasks being scheduled and a dependency added.

Chapter 3

Problem Statement

When a task is no longer dependent upon, it still persists in the dependency graph. During top down execution these tasks without incoming dependencies are never visited. However, the bottom up algorithm will continue to schedule and execute the task.

To demonstrate this, suppose the user requires only the result of `calcSum(delta)`. Next we remove the dependency to `calcSum(gamma)` from `delta` (Figure 3.1). Now, if we modify alpha or `gamma` the function `execAndSchedule` schedules `calcSum(gamma)`.

That is, tasks are executed even when they no longer contribute a result that the user is (transitively) interested in. The time spent building these results is wasted. The severity of the inefficiency ranges from benign up to destructive Out of Memory Errors depending on the task definitions in use.

To determine which tasks are irrelevant, the user must indicate a set of tasks that are relevant. We define the *RootObserved* state (Indicated by the oval shape of `calcSum(delta)` in Figure (3.1)). This state indicates that a task was directly requested by the user. When a dependency is required it is *Observed*. When a task is completely removed from the transitive closure of a *RootObserved* task we define it to be *Detached*. With this information, the build algorithm can skip scheduling Detached task for execution. However, this creates a new problem. A Detached task that has not been executed is not known to be consistent.

Our goal, is to design, realize, and evaluate a method to efficiently track observability, avoid executing Detached tasks, and efficiently build re-observed tasks.

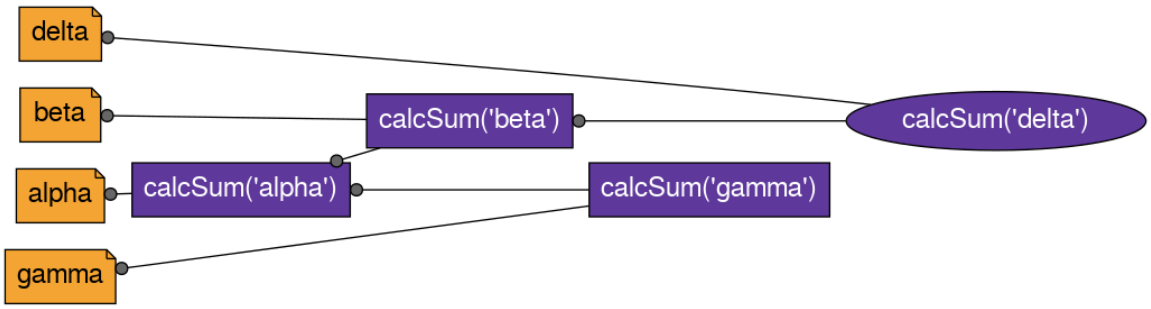


Figure 3.1: Removing the dependency from `calcSum(delta)` to `calcSum(gamma)`. If the user only needs the result of `calcSum(delta)`, scheduling and executing `calcSum(gamma)` is wasted time.

Chapter 4

Key Idea and Solution

This chapter describes our key idea and solution. We divide our solution in three parts. The first deals with setting tasks to *Detached*. Next we modify the algorithm to avoid executing the task. Finally, we describe a method to efficiently re-observe the *Detached* tasks.

The general outline of our solution is as follows: We update the observability state of tasks during execution when a dependency is removed. We recursively set the task to Detached if all tasks depending on it are Detached. During bottom up execution, Detached tasks are not scheduled. When a Detached task is required, we execute a top down incremental build of the task while ensuring tasks already scheduled are taken into account.

4.1 Detaching

An inefficient solution to determine if a task is Observed or Detached is to do a simple traversal algorithm after each rebuild in which the dependency graph has changed. First we mark all non Observed task as Detached. Next we traverse the graph starting at the RootObserved and mark every visited note as Observed. However, this defeats the purpose of bottom up execution. It requires the traversal of the entire graph, making the algorithm scale with the graph size instead of the size of the change. Instead we modify the algorithm to track observability during execution when the dependencies are modified.

Our goal is to uphold the following two invariant for a sub-graph after a task is required:

- A task in the transitive closure of a RootObserved task is either RootObserved or Observed.
- A task not in the transitive closure of a RootObserved task is Detached

Initially, when the user invokes `buildNewTask` (Line 4) and after the task is made consistent with `require` its observability state is set to RootObserved. The interface is expanded with `dropRootObserved` (Line 14) for the user to remove the RootObserved status of a task.

To uphold the first invariant we modify the `exec` function (Line 21). If the task is new or it is Detached it is set to Observed.

To ensure the second part of the invariant, we determine the set of tasks which have been removed as dependencies. For each we call `propagateDetachment` (Line 32). `propagateDetachment` sets the transitive closure of the task to Detached, but not in the case that a task is RootObserved or that another non-Detached task requires this task.

4.2 Scheduling

The modifications required to avoid scheduling are minimal (Listing 4.2). If a task is Detached we do not schedule it (Line 5).

Listing 4.1: Modifications to the bottom-up algorithm to track the `ObservableState` of each task. Original code (Listing 2.2) is shown in grey.

```
type ObservableState = RootObserved | Observed | Detached }

function buildNewTask(t)
  T_e := ∅; O_c := ∅;
  require(t)
  DG.setObservableState(t, RootObserved)

function addRootObserved(t)
  if DG.observability(t) == Detached then
    buildNewTask(t)
  DG.setObservableState(t,RootObserved)

function dropRootObserved(t)
  if DG.observability(t) != RootObserved
    return
  DG.setObservableState(t,Observed)
  propagateDetachment(t)


function exec(t)
  if t ∈ T_e then abort;
  if DG.observability(t) == Detached ∨¬ DG.observability(t) then
    DG.setObservableState(t, Observed)
  val dep_old = DG.taskRequires(t)
  T_e := T_e ∪ t; valr := t.run(); T_e := T_e\t
  removed := dep_old\ DG.requiredBy(t)
  for d ← removed do propagateDetachment(d)
  DG := DG ∪ r ; validate(t, r); notify( t, r .output )
   O_c[t] := r.output ; return r.output

function propagateDetachment(t)
  if DG.observability(t) == RootObserved then return
  for (_, t_call) ← DG.callersOf(t) do
    if DG.isObserved(t) then return
  DG_new.setObservability(t, Detached)
  for (_, t_req) ← DG.tasksRequired(t) do
    propagateDetachment(t)
```

Listing 4.2: Modifications such that Detached tasks are not scheduled. Original code (Listing 2.3) is shown in grey.

```
function schedAffCallersOf(t, o )
  for (stamp, t_call) ← DG.callersOf(t) do
    if ¬stamp.isConsistent(o) then
      if DG.isObserved(t) then
        T_q := T_q ∪ t_call
```

Listing 4.3: Adding and integrating `requireDetached`. Detached tasks can be re-used without re-executing when they are consistent. Original code (Listing 8 and Listing 2.3) is shown in grey.

```
function require(t )
  if t ∈ O_c then return O_c[t]
  if ¬DG.hasOutputOf(t) then return exec(t)
  if ¬ DG.isObservable(t) then return requireDetached(t)
  return requireScheduleNow(t)

function requireScheduledNow(t)
  while val t_min := T.q.leastDepFromOrEq(t) do
    T_q := T_q\t_min
    val o := execAndSchedule(t_min)
    if t = t_min then return o
  val o := DG.outputOf(t)
  O_c[t] := o
  return o

function requireDetached(t)
  for (stamp, f_provide) ← DG.filesProvided(t)
    if ¬stamp.isConsistent(f_provide) then return exec(t)
  for (stamp, f_require) ← DG.filesRequired(t)
    if ¬stamp.isConsistent(f_require) then return exec(t)
  for (stamp, t_require) ← DG.tasksRequired(t)
    val r_req;
    if DG.observability(t_require) == Detached then
      r_req := requireDetached(t_require)
    else
      r_req := require(t_require)
    if ¬stamp.isConsistent(r_req) then return exec(t)
  DG.setObservableState(t,Observed)
  O_c[t] = t.output
  return t.output
```

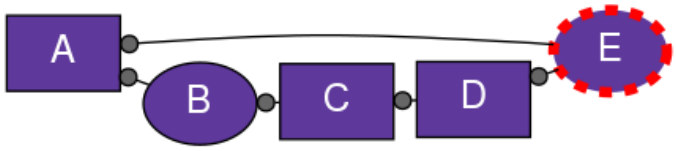
4.3 Require Detached tasks

The final challenge is to modify the algorithm to deal with a *Detached* dependency that is required again (Listing 4.3).

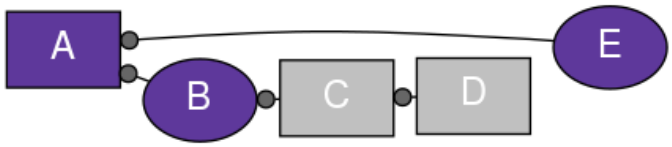
The function `require` is modified to handle the case where a task is Detached (Line 5). `requireDetached` (Line 17) is similar to an incremental top down function.

In order to determine consistency the function first compares the stamps for the files that the task provides and generates. These stamps were created before the task was detached. Next it iterates over the tasks it requires in order. For each it compares the stamp of its last execution and the current state of the task.

The consistency of a task is determined. For Detached tasks this is done through depth first invocation of `requireDetached` (Line 25). If the task is not Detached its result is determined with `require` (Line 27). Each result of the dependency is compared to its previous stamp. If any dependency has changed the task is re-executed, otherwise the task is consis-



(a) E is modified to remove D as a dependency.



(b) During `exec(E)`, D $\in$ *removed*, and thus `propagateDetachment(D)` sets D and E to Detached

Figure 4.1: Two snapshots of the dependency graph (without files) during a rebuild showing the process of `propagateDetachment`.

tent and added to the set of visited tasks O_c. [1]

Properties

We now argue that after running `requireDetached` the task is consistent if the task definitions are sound.

First, consider the case that all tasks in the transitive closure of t_d are Detached. In this case, the correctness is analogous to that of a top down incremental build. If a task t_o in the transitive closure of t_d is Observed, `require` is invoked. t_o may be in a number of states. If it is in the visited set, t_o must be consistent. If the data related to the task has been removed it must be re-executed, otherwise `requireScheduledNow` is invoked.

`requireScheduledNow` ensures that all tasks potentially affecting t_o are executed before t_o is executed or its old result returned as is argued in Konat, Erdweg, and Visser 2018. In short, it returns a consistent result if the invariants hold. That is, no cycles are formed (anything requiring t_d or t_o), and no file is edited which was required by a task in the visited set.

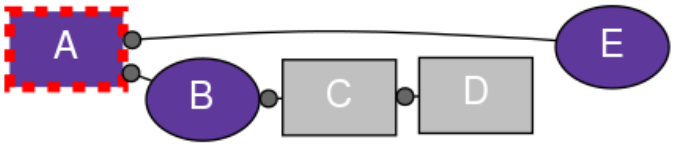
A new variable to consider in `requireScheduledNow` is the impact that Detached tasks have on correctness. This is resolved by the observability invariant. The method `requireScheduledNow` is only ever invoked for Observed or RootObserved tasks. Our invariant guarantees that there are never Detached tasks in the transitive closure.

4.4 Example Execution

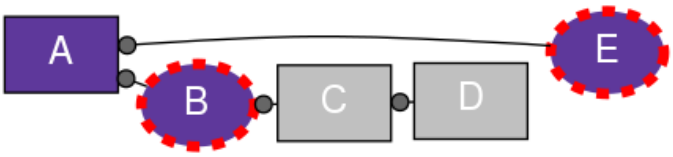
We step through an example. In order to cover most of the algorithm conceptually we do not provide concrete task definitions. Instead, note the following: A task may change its dependencies by the result of a file, or the result of another task. Therefore, we remove file dependencies and schedule tasks at will during our example execution. This provides clarity in the visualization by removing the file nodes,

The graph is initiated with `buildTask` for task B and task D. These have become RootObserved and have become oval shaped in the Figures. First we change E to demonstrate `propagateDetachment` (Figure 4.1). Next we change A to demonstrate `requireDetached` (Figure 4.2)

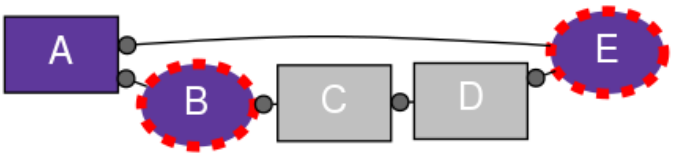
[1] Alternatively, $r_{req} := require(t_{require})$ can replace the if-else, `require` checks for Detached tasks and invokes `requireDetached`. However, we find this formulation easier to reason about.



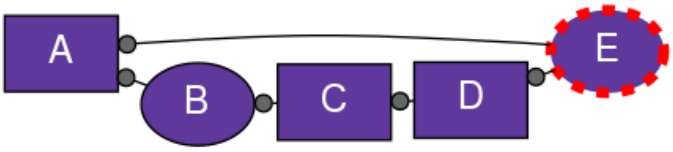
(a) A. The process starts with `execAndSchedule(A)`. The change will eventually make D a dependency of E.



(b) `scheduleAffectedCallersOf(A)` schedules B and E. We arbitrarily pick E to execute first



(c) `exec(E)` invokes `require(D)`. This sets up the chain of `requireDetached` for D and then C. C will `requireScheduledNow(B)`.



(d) E can continue execution with the result of D.

Figure 4.2: Four snapshots of the dependency graph (without files) during a rebuild showing the process of `requireDetached` and how a dependency is made consistent before it is used.

- Figure 4.1a)
  Task E is modified. It removes its dependency on task D.

  After execution, `propagateDetachment(D)` is invoked.

- Figure 4.1b)
  `propagateDetachment(C)` and `propagateDetachment(B)` are invoked. The latter doesn't take effect because B is RootObserved.

Continuing from Figure 4.1b we consider `requireDetached` in Figure 4.2. Task A is modified. Its result will add a dependency from task E to task D.

- Figure 4.1a)
  First A is executed.

- Figure 4.1b)
  After execution both task B and E are scheduled because they require A.

- Figure 4.2a)
  First E is executed[2]. During execution, it invokes `require(D)` and consequently `requireDetached(D)`.

[2]There is no transitive relation between E and B, so either may go first.

To determine if D is consistent, it will `requireDetached(C)`. To determine if C is consistent, it must get B. Because B is Observed `require(B)` is invoked. This in turn invokes `requireScheduledNow`. Task B is scheduled and in its own transitive closure. C finishes execution and finally D.

- Figure 4.2b
  Task E can now determine if it must be executed.

4.5 Implementation

We have implemented observability as part of the PIE runtime (Konat, Erdweg, and Visser 2018). PIE and our modifications are written in Kotlin and runs on the JVM. The code published online [3]. For our research we created a third build algorithm with the same interface as the bottom up executor[4]. This allows us to compare the old and new algorithm with minimal changes. However, the official PIE release has merged the ideas developed into the bottom up algorithm entirely.

[3] https://github.com/metaborg/pie
[4] https://github.com/RoelofSol/pie/tree/observability

Chapter 5

Evaluation

In this chapter we evaluate the new algorithm compared to the old. Specifically, we try to answer the following three questions:

1. How does the new algorithm perform compared to the old?
    a) Have we solved the scalability issue of executing detached tasks (Section 3)?
    b) What are the costs of validating and reusing tasks that have been detached?
2. Is the new algorithm suitable for a real-world application?
3. Does the new algorithm produce the correct results?

We approach the question 5.3 by creating two artificial scenarios. To answer 1a we create a diamond shape graph and have a number of detached tasks. For question 1b we detach, build, and require a task in a number of different ways.

The performance characteristics changes depending on the tasks in question. So in order to be thorough we evaluate two situations. Either execution time is dominated by tasks execution, or it is dominated by the PIE algorithm. In our experience, task execution time is the dominating factor in build scripts. Especially if they invoke an external program. However, we are also interested in what happens when tasks have a negligible execution time and the PIE algorithm becomes an important factor.

To answer question 5.3 we extend the Spoofax benchmark created in Konat, Erdweg, and Visser (2018). To resolve question 5.3 we compare the results we generate between the two algorithm while running these benchmarks.

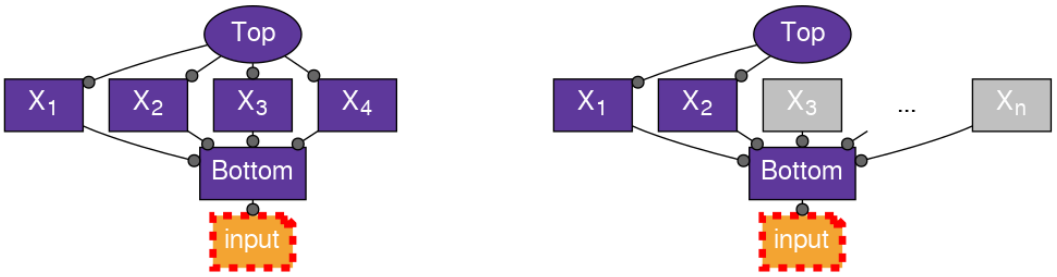
Experimental Setup We use OpenJDK version 1.8.0_202. Our benchmarks are executed on a Ryzen 7 1700 processor with hyper-threading disabled and 16 GB of DDR4 memory. Each core has a 512 KB cache size. The disk used is a 120 GB SSD with a 600 MBps transfer rate.[1]

5.1 Artificial Test Cases

First we attempt to answer question 5.3. We created two types of scenario's. First we describe the commonality in their setup. Next, for each benchmark we explain their specific setup and we interpret their results.

In both scenario's we create a build script that constructs a dependency graph of a specific shape and size. We simulate actions by the user and track the execution times of rebuilding the graph for different sized graphs. Both benchmarks are executed twice. First we add an artificial delay of 10ms. This allows us to validate our expected time savings when we avoid executing a detached tasks, and simulates the common scenario where rebuild time

[1] Although we suspect no disk IO was performed after the initial warm-up.



(a) Phase 1: the width is increased every round. Next round will add X_5, then X_6 etc.

(b) In phase 2 the width is decreased every round leaving unobserved tasks. In phase 3 the detached tasks are required again.

Figure 5.1: The shape of the dependency graph used in the diamond benchmark.

is dominated by the build steps. Next we remove the delay and create a task that does a single integer addition. This gives insight into the cost of the new `propegateDetachment` and `requireDetached` functions that were introduced in chapter 4. We observe the outputs of each task and compare them to the old in order to answer question 5.3.

The artificially slow executions start with a single warm-up and record 3 iterations. The benchmarks without a delay start with 15 warm-up rounds and record 30 iterations. An average is taken and the standard deviation is plotted as the error[2].

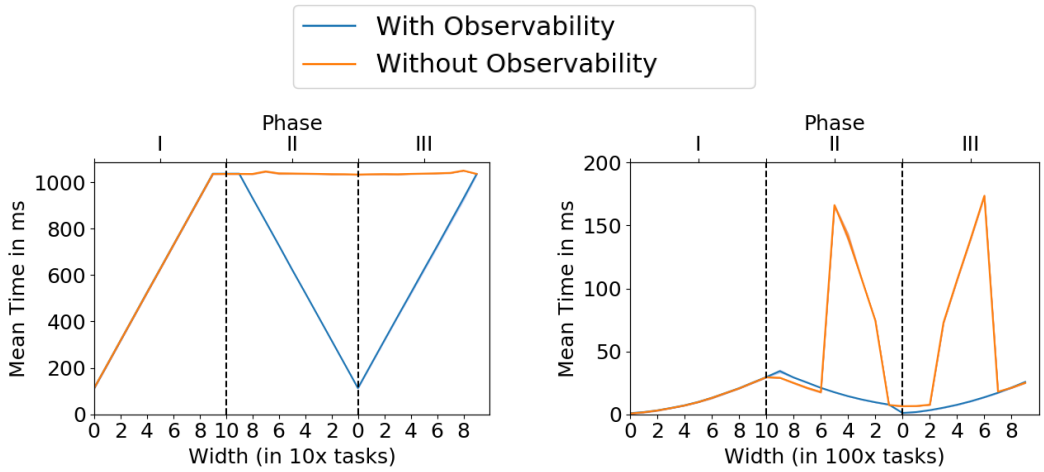
Finally, we avoid using system-calls for file-system access even though system calls would likely be significant. There are two reasons for this. First, they would cloud our measurements by adding (semi) random noise. Secondly, with the help of a watchdog process or the certainty that no external program is editing the files, the number of file system access can be equivalent to the old algorithm.

Diamond

For our first benchmark we wish to answer question 1a: Do we avoid executing detached tasks? To do this, we create a build script that generates dependency graphs in the shape of a diamond (Figure 5.1a). Every round the bottom task is modified and all tasks must be re-executed. The modification of the bottom task schedules every edge task. After these are done, the top task is scheduled and adds or removes a number of task dependencies. We divide the state of the graph in three phases, with every phase containing 10 rounds in which the graph is increased or decreased in size each round. In phase 1 (Figure 5.1a), the top task increases the number of dependencies. That is, the graph is widened with new tasks each round. This process is equivalent for the old and new algorithm. Phase 2 (Figure 5.1b) is the process of removing dependencies from the top task. The old algorithm schedules every task, even those detached from the top task. The new algorithm has to invoke `propagateDetachment` for each removed dependency, but subsequently avoids scheduling and execution. In phase 3 (Figure 5.1b) the new algorithm has to re-observed tasks through `requireDetached`. Their previous result is invalid and the tasks are executed.

When task execution time is the dominating factor we expect the new algorithm to outperform the old in phase 2 and 3 proportional to the number of unobserved task. Throughout execution we validate consistency by comparing the results of each observed task between the old and new algorithm.

[2] After ~10 warm-up iterations the error becomes insignificant.



(a) Build times when each task takes 10ms to complete.

(b) Build times when each task is only a single addition operation.

Figure 5.2: Diamond benchmark results. The build times for the old and new algorithm building a variable sized diamond shaped graph.

Interpretation

The diamond benchmark with artificial delay performed as expected (Figure 5.2a) and answers question 1a. We have solved the scalability issue described in Chapter 3. During the second and third phase, the old algorithm schedules and executes every task detached form the Top task (Figure 5.2b).

It should be noted that the transition between phase one and two, the new algorithm lags before decreasing, and it doesn't reach zero. This is expected but not immediately obvious. The reason is as follows.

The increase in width is done in batches of a 100 tasks. The graph reaches its maximum width of a 1000 and it ends phase 1. When it enters phase 2, the end result will be that a 100 task are detached from the top task. However, these not-yet-detached tasks must be executed before the top task can execute and remove them as dependencies.

In the benchmark without delay (Figure 5.2b) the results are strange but consistent across multiple benchmarks. That is, the errors are plotted in the graph, but are indistinguishable because of their insignificance. First, there is an unexplained spike during execution of the old algorithm. Another interesting result is the dip in execution time just after entering phase 2 for the old algorithm. Potential culprits include JIT re-calibration, CPU pipe-lining & branch-prediction, and measurement error.

We expected `propegateDetachment` and `requireDetached` to be significant in the new algorithm when tasks execution time was insignificant. However, this benchmark did not confirm this.

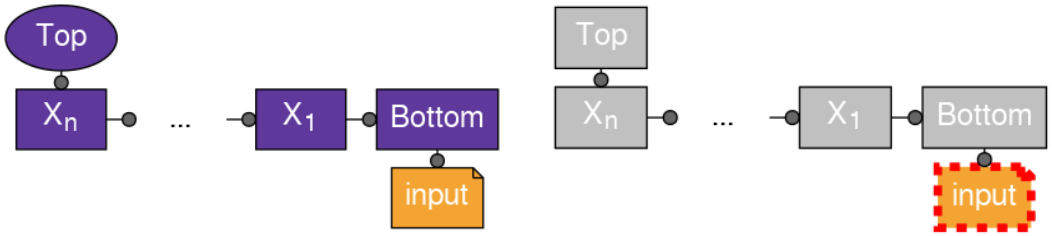
Tube

In our second benchmark we answer question 1b: What are the cost of validating detached tasks.

To do this we created a build script that generates dependency graphs in the shape of a tube (Figure 5.4a). We create a tube of a given length and run three tests on them. First, we drop the observability (Figure 5.4b) and then re-observe. Next, we drop the observability, and invoke a rebuild once with a modification, then we re-observe. Finally, we drop the observability, and invoke a rebuild twice with a modification, then we re-observing. In this sequence we expect the old algorithm to be infinitely faster in the first case. Dropping

Table 5.3: The three scenario's for the Tube Benchmark. A sequence of `dropRootObserved`, `buildWithChangedFiles` and `addRootObserved`. Observability has no effect in the old algorithm, but it must continuously exec unobserved tasks.

Scenario	[drop,add]	[drop,change,add]	[drop,change,change,add]
Old	-	exec	exec X2
New	requireDetached	exec	exec



(a) The tube is Observed.

(b) The tube is Detached.

Figure 5.4: The dependency graph in the Tube benchmark for the new algorithm. We create a long chain of dependencies and switch between observed and unobserved.

and re-observing is a no-op for the old algorithm. For the second case we expect the algorithms to be comparable. With the new algorithm at a slight disadvantage because it must `propegateDetachment` and `requireDetached`. And finally, we expect the new algorithm to outperform the old when a Detached task must be rebuild (Table 5.3). We do this for 10 different length tubes. Again, similarly to the previous benchmark, we compare the result of each task execution between the old and the new in order to validate the correctness of our solution.

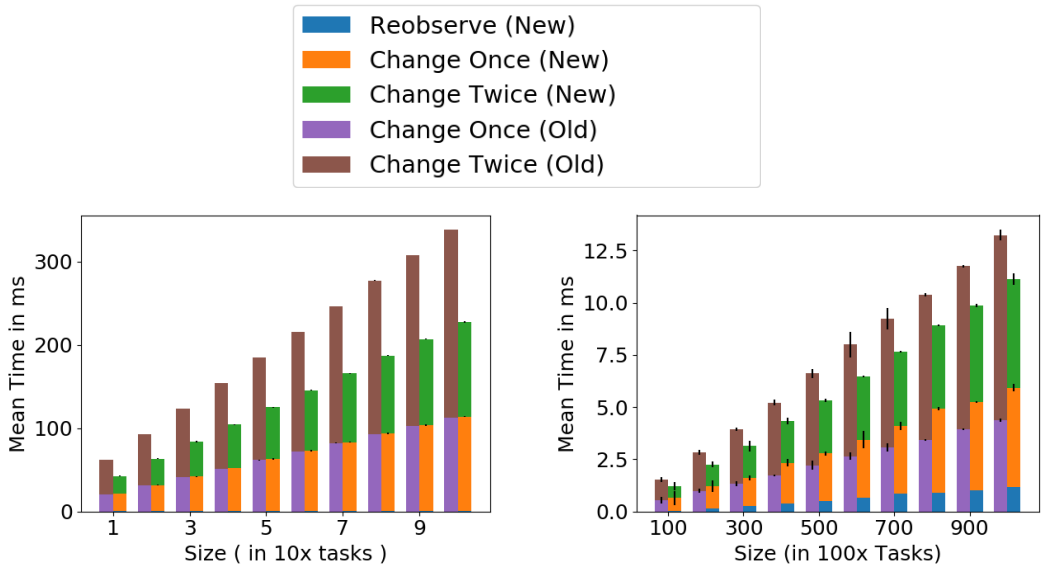
Interpretation

The benchmark performed better then expected and answered question 1b. The cost of validating a large sequence of detached tasks is insignificant in practice. Even when tasks themselves are trivial, the cost of validating a very long graph is low. Suggesting that `requireDetached` for a 1000 tasks with a single dependency is approximately as fast as running 200 trivial tasks. However, it should be noted, that by the nature of our artificially shaped graphs, the JVM and CPU will likely perform optimizations that would not be applicable in the real world use cases.

5.2 Real World Test Case: Spoofax Build Script

In this chapter we attempt to answer question 5.3: Is our algorithm suitable for real-world application?. To test this we take a real world PIE build script. We set observability and measure the build times after modifying a file. The build script implements a pipeline for the Spoofax Language Workbench [3]. Spoofax (Kats and Visser 2010) is a Language Workbench (Erdweg, Storm, et al. 2013) for the development of Domain Specific Languages (DSLs). This is a class of IDE used for developing programming languages and their tools such as syntax highlighters, type checkers, and compilers. The code for our benchmark can be found online at [4].

[3] https://github.com/metaborg/spoofax-pie
[4] https://github.com/RoelofSol/spoofax-pie/tree/observability



(a) Execution time in ms after adding a 10ms delay to each task

(b) Execution times without delay

Figure 5.5: The Tube Benchmark. A comparison of time spend building and time spent validating consistency in the old and new algorithm. A dependency graph (Figure 5.4 is tested in three scenarios (Table 5.3) for the old (Bars on the left) and the new (Bars on the right). The cost of `requireDetached` is low compared to executing tasks (Bottom right bars in Figure 5.5b)

For our purpose we consider a simple tutorial language 'Calc' and simulate a user that is editing the language, a syntax highlighter and two example files. We measure every change in every state for this project and extrapolate cost of using this in production.

Experimental Setup

Spoofax requires a large and intricate pipeline of different tasks with overlapping dependencies. For the purposes of our benchmark we only consider a single language definition and limit our tasks to parsing and styling.

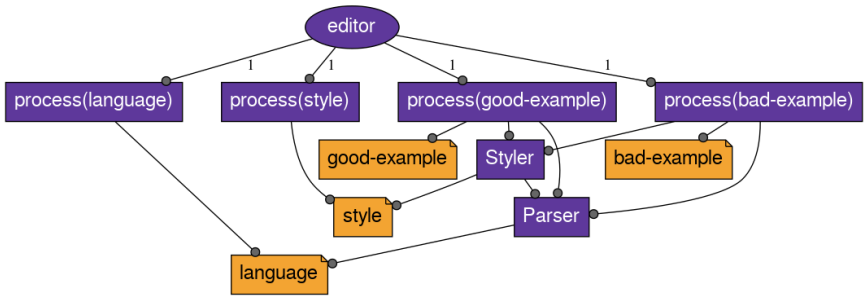
The PIE pipeline for Spoofax defines a function `processDocument`. This function generates feedback for the user. Our modified pipeline consist of a language editor that is processing four files.

- The `language` file contains the definition of the parse table.
- The `style` contains information on what color highlights to apply to specific keywords.
- `good-example` contains 500 lines of valid source code in the target language.
- `bad-example` contains invalid source code.

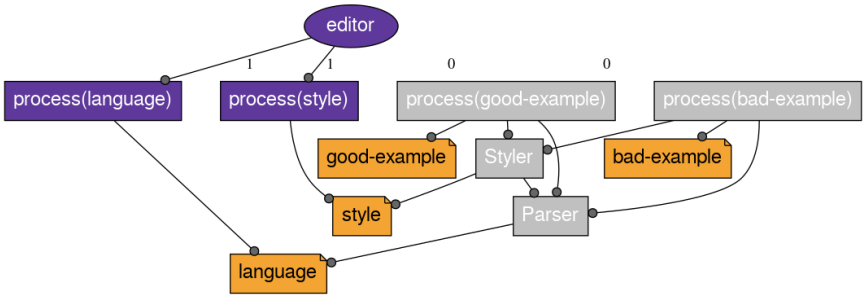
The Spoofax IDE might detach `processDocument` for a number of reasons. Similarly, files can be modified for a number of reasons. Scenario's include: Renaming a file, closing a file, temporarily disable highlighting, opening a new language project, git checkout, and others.

We do not define scenario's explicitly, instead we measure every build time after changing a file in every possible state of observed `processDocuments`.

The four `processDocument` tasks represent a total of $2^4 = 16$ observable states. That is, 16 unique combinations of observed and detached tasks. We edit the content of each file for



(a) State: 1111 - Every task is observed.



(b) State: 1100 - Both example programs are unobserved, nothing is parsed or styled.

Figure 5.6: Expected dependency graphs representing a observed state in the Spoofax Benchmark.

every combination of observed tasks. In addition we benchmark the Setup Observably time, when we modify the observability state. This creates $16 * 5 = 70$ measurements.

We execute the entire benchmark 10 times.

Table 5.7 shows the difference between the new and old algorithm for each state. A negative value means an improvement to the modified algorithm. The column 'Observed' represents the RootObserved tasks in the build. A 1 means the task is Observed and a 0 means the `processDocument` task is Detached.

Results and Interpretation

The results show improvements when documents are not processed. This follows our expectation that not observing any example program would detach the Parser (Figure 5.6b).

The initial setup for state 1111 sets all `processDocument` as RootObserved. This took approximately 30,000 ms and executes 42 tasks and requires 196 files. Our table shows an increase in build time for the old algorithm of 133 ms on average with a standard deviation of 486 ms. This is to much noise to draw any conclusions.

State 1110 represents the invalid source code file not being processed. There are no significant changes in build time. Parsing fails so it has no other dependencies.

State 1101 shows the build times when the good example file is modified. It shows a significant improvement when editing the language. A similar pattern exists in state 1001, 0101, and 0001. The logs indicate the new algorithm is is able to skip building the task `sdf3.toStrategoSignatures`.

The state 1100, 1000, 0100, and 0000 all show the same improvement of a reduced build time of approximately 1800 ms when editing the language. The logs indicate that the new algorithm is able to skip the task `sdf3.ToJSGLRParseTable`

The increase in setup for states 1011, 0111, and 0011 is due to the fact that in the previous state, the language was not observed and thus not build.

Observed	Setup	Edit lang	Edit style	Edit Good	Edit Bad
1111	133 ± 486	0 ± 0	0 ± 1	0 ± 0	0 ± 0
1110	7 ± 25	1 ± 50	-2 ± 3	-1 ± 4	-1 ± 0
1101	40 ± 10	-567 ± 44	-20 ± 2	-13 ± 1	0 ± 0
1100	6 ± 5	-1852 ± 31	-17 ± 1	-15 ± 3	0 ± 0
1011	1417 ± 19	-2 ± 54	22 ± 4	0 ± 1	0 ± 0
1010	3 ± 2	-10 ± 56	18 ± 4	0 ± 1	0 ± 0
1001	15 ± 2	-599 ± 44	-16 ± 1	-12 ± 1	0 ± 0
1000	5 ± 3	-1787 ± 29	-16 ± 4	-11 ± 0	0 ± 0
0111	1377 ± 18	-29 ± 55	23 ± 0	0 ± 2	0 ± 0
0110	1 ± 1	-5 ± 49	16 ± 3	-3 ± 5	0 ± 0
0101	10 ± 1	-588 ± 45	-15 ± 2	-11 ± 0	0 ± 0
0100	0 ± 1	-1774 ± 30	-14 ± 0	-13 ± 3	0 ± 0
0011	1345 ± 13	-23 ± 52	23 ± 3	1 ± 1	0 ± 0
0010	0 ± 0	-3 ± 55	17 ± 2	1 ± 1	0 ± 0
0001	10 ± 0	-596 ± 46	-14 ± 0	-11 ± 0	0 ± 0
0000	0 ± 0	-1763 ± 32	-14 ± 0	-11 ± 0	0 ± 0

Table 5.7: The average difference (New - Old) and the average standard deviation, for `buildWithChangedFiles` after an action, for every possible combination of Detached tasks. Negative values mean an improvement in speed. 'Edit Lang' updates the language definition. 'Edit Style' modifies the style. 'Edit Good' makes a change to valid source code. 'Edit Bad' makes a change to invalid source code. 'Setup' sets tasks to RootObserved according to 'State'.

State 0000 shows the effects over all. Editing a language file is vastly faster when no documents require it. Editing a style and editing the good file is faster because they do not get processed. However, only parsing and highlighting is applied. When advanced type checking is added, this is likely to increase significantly.

Overall, observability can realize significant efficiency improvements when tasks are detached. This means faster feedback time, and also the ability to add features such as renaming a file.

5.3 Conclusion

We set out to evaluate our new algorithm by attempting to answer three questions.

How does the new algorithm perform compared to the old?

We demonstrated that our new algorithm is an improvement of the old. The algorithm is scalable over time when tasks become detached. Furthermore, the execution time of our new functions is insignificant compared to executing a task.

Is the new algorithm suitable for a real-world application?

The new algorithm improves the feedback time of the Spoofax pipeline in some situations. The Spoofax pipeline no longer processes files after closing them. However, the design of the build script is the determining factor in the benefits gained from the new algorithm. It

is possible to write a script that performs poorly. Another important consequence of our improved algorithm is the new design patterns enabled by observability. Such as renaming files in the Spoofax IDE.

Does the new algorithm produce the correct results?

In all our tests we compared the results of the new algorithm with the old. All observed tasks produced the same results. This however does not prove the absence of bugs. In chapter 8 we mention further work that can be done with respect to correctness.

Chapter 6

Miscellaneous improvements

During development we developed a useful tool to visualize our problem and PIE scripts in general. Additionally we experimented with subsequent improvements that observability enables. In this chapter we briefly document our experience.

6.1 Dependency Graph Visualizer

We have implemented a tool to visualize the state of a DAG. The DAG at its core is a key-value mapping containing the nodes, their state, and in and outgoing edges. Our tool transforms this representation into valid `dot`Ellson et al. 2001 syntax with appropriate shapes and colors similar to the ones used in this paper. This is then passed to the dot program to be processed into an image. We have included an example of a graph created with our Spoofax pipeline in the appendix (Figure A.1).

The use of this tool has been invaluable in analyzing and debugging PIE itself and build scripts written in PIE. However it has limitations. First, its layout is inconsistent. Graphviz optimizes the layout with respect to the dependency edges. Consequently, nodes in a graph jump around when dependencies are removed or added. Secondly, we hit the limit of graphviz when we attempted to visualize the graph from our Spoofax benchmark. Most states simply crashed dot. For those that were small enough the result is a ~26MB png file at about ~230,000 by 1,000 pixels.

6.2 Garbage Collection

With our modified algorithm we are able to implement a simple form of garbage collection. Unobserved tasks waste time and space if they are never required again.

We have implemented a simple garbage collection algorithm (Listing 6.1). In our solution each class of task definitions implements an optional `removeIfUnobserved` function. This function returns a bool indicating if it is acceptable to remove the task from the dependency graph. If a task is both Detached and `removeIfDetached` returns true, the task is removed from the graph. If the function is not present the system defaults to false in order to make this a backwards compatible change.

Listing 6.1: A garbage collection algorithm for unobserved and user-defined-collectible tasks

```
function gc(DG) {
  U = DG.unreferenced()
  while (U ≠ ∅)
    t = U.pop()
    if (¬DG.isObserved(t) ∧ t.removeIfUnobserved())
      DG.dropTask(t)
      U ← DG.requiredBy(t)
}
```

Chapter 7

Related work

In this chapter we discuss related work with respect to PIE as a build system, PIE's relation to other methods for incremental computation, and our method of observability.

In so far as we know, there exists no other bottom up execution method for general incremental computation similar to PIE's. As a consequence, observability has no direct relation to other systems in general. Most other incremental systems start execution at the top or do not allow dynamic dependencies. When executing top down, an algorithm doesn't risk scheduling unrelated tasks in the way PIE did.

7.1 Build Systems

Make (Richard M. Stallman 2016) operates by building and traversing a dependency graph in a depth first search; executing any inconsistent rules. It has no first class support for dynamic file dependencies or dynamic task dependencies. It does not have a proper programming model. That is, its model is a global namespace of mutable string variables. In addition, it doesn't support custom file stamps or detection of hidden dependencies. As a result, Makefiles are commonly generated and dependencies have to be over approximated or the user risks creating inconsistent results. Many other build systems follow a similar approach to Make. Because Make always builds the entire dependency graph from scratch, there is never an issue with observability.

Shake (Mokhov, Mitchell, Jones, and Marlow 2016) is an incremental build system written in Haskell. It offers solutions to many of Make's problems such as proper namespaces and variable types, use of higher order functions, the use of libraries written in Haskell, build rules with multiple outputs and a form of dynamic dependencies. However, compared to PIE it has limited support for dynamic file dependencies and generated files are statically determined. In addition, the dynamic dependencies are referred by 'keys'. These are not parameterized and return no value. Unreferenced tasks are garbage collected after execution.

A categorization for build systems is given in (Mokhov, Mitchell, and Jones 2018). It does not contain a Scheduling abstraction that incorporates the set of changed files in a rebuild. However, simply adding such an abstraction is insufficient. Specifically, when the set of changed files is known, new and optimized top down methods become available. We discuss such a method in chapter 8.1.

7.2 Incremental Computation

PIE is a form of incremental computation (IC). IC refers to any attempt to save time on computation by reusing an old result. This definition is applicable to many systems. Starting with dynamic programming Bellman 1954, IC has been studied continuously Ramalingam and Reps 1993, including more recent work such as Nominal AdaptonHammer, Dunfield, et

al. 2015. These systems usually develop domain specific techniques that exploit some invariant reducing the need to store traces or graphs and reducing the overhead in determining what must be done. For example the View Maintenance problem in databases Jain and Gosain 2012, or Make's assumption that all builds only require a static dependency graph to traverse. As such, they do not implement general purpose IC.

A relatively recent general purpose IC language is Adapton Hammer, Khoo, et al. 2014 Hammer, Dunfield, et al. 2015. Adapton is a formal operational semantics and general purpose programming language. PIE and Adapton both implement incrementality through partial re-execution of a dependency graph. On the surface they differ in that Adapton does not have custom task-keys or stamps, and PIE has a focus on files instead of mutable variables.

With respect to their change driven build algorithm, the difference is as follows: Adapton achieves incrementality by flagging the tasks transitively dependent on a change as dirty. When a dirty expression is demanded, it recursively checks, to either compute or reuse a result. On the other hand, PIE starts scheduling and execution at the point of change.

The PIE approach has down sides. First, there is the additional complexity introduced in this work to avoid executing detached tasks. Secondly, in order to determine if required result has a transitive dependency on tasks that are scheduled, the function is `requireScheduledNow` must traverse the dependency sub-graph. Third, Tasks that are Detached *during* a build might be executed before being detached (See the evaluation for the diamond benchmark chapter 5.1). The upside for PIE is its early cut-off if a result has not changed. This avoids traversing upwards in the dependency graph and is optimal if a change has no effect.

Additionally, Adapton does not have a mechanism for granular garbage collection and does not persist on disk.

Chapter 8

Future Work

8.1 Build Algorithms

The top down algorithm can be improved. A relatively simple addition would be top down execution with changed files in order to avoid checking stamps on every file. On top of this, we believe it is possible to add a Bloom filter to each task to indicate a set of files required by its transitive closure. This would allow top down execution to skip graph traversal with some probability. Furthermore, the Adapton algorithm would make an interesting addition to the set of build algorithms in the PIE runtime.

When invoking `propagateDetachment` the algorithm has to determine if its transitive closure is consistent. This process scales by the size of the sub-graph. This is analogous to the top down scalability issue. Additionally, on first inspection it appears that checking stamps during `requireScheduledNow` is expensive. In order to avoid checking stamps, a set of changed files could be tracked for the duration that the task is Detached. `propagateDetachment` might then use any strategy able to handle a set of changed files.

Finally, the re-appearance of the mentioned top down scalability issue suggest a further abstraction can be created to incrementally Observe a detached tasks by an executor.

8.2 The PIE Language & The Runtime

We have used 'file' to describe variables in our mutable environment. Work is under way to expand this into the generalized notion of 'resource'. In general, any combination of getter and setter should be adoptable.

We have shown a simple manually configured garbage collector that the user can invoke. However, there are automatic policies such as Least Recently Used that might be easier for the user. It is also possible for garbage collection to be triggered immediately after execution for all Detached task. Both of these methods of freeing memory are aided by expanding the PIE language to include additional keywords. First we propose to introduce the `function` keyword to indicate a task can be removed if Detached. Next we propose to make garbage collection opt-out per task definition instead of the current opt-in system. Opt out of garbage collection can be indicated by `persistent taskdef .`

The next major milestone for PIE should be to support concurrent execution. Inspecting the dependency graph allows some analysis of which tasks can be executed concurrently. However, the 'internal' dependency graph of each task would provide additional information. PIE could analyze the task definition and statically group tasks which are required together unconditionally [1].

[1] similarly to dependency analysis used in compilers

As noted in (Konat, Steindorfer, et al. 2018), PIE currently doesn't support dynamic task definitions. As an alternative, PIE could inline the runtime into the generated objects creating a self-contained algorithm without dependencies (similar to the technique used in SvelteJS[2])

8.3 Analysis

Future work may focus on a formal semantics or correctness proof in a tool such as Coq. A challenge would be to model a 'dependency flip'. That is, a dependency from A to B changing into a dependency from B to A. [3]

In our evaluation we created two types of graphs to use in our benchmark, a 'diamond and 'tube. There are however an infinite number of graphs and graph transformations. In order to further compare build algorithms, similar synthetic scalable graphs should be collected. An interesting addition would be hash algorithms. They are the antithesis of incremental computation. This collection is valuable to other incremental systems as well. In addition, this might prove to be an efficient method to categorize incremental systems with respect to their capabilities and runtime.

In an ideal world these build algorithms are compatible and reusable *inside* one another. A 'meta' build algorithm could then focus on applying the best one for the build at hand. Especially task definitions without dynamic dependencies can be aggressively optimized.

[2] https://svelte.dev/

[3] In fact, this might be impossible for the general case.

Chapter 9

Conclusion

We have demonstrated the need for PIE's bottom up build algorithm to avoid executing tasks which are no longer required. We have designed an improved algorithm such that, a detached task is correctly labeled, the detached tasks are not executed, and re-observing tasks is done efficiently in order to improve scalability over time. To demonstrate the effectiveness and correctness of our improvements we have compared the old and new algorithm with two types of artificial dependency graphs. These benchmarks have shown that the new algorithm avoids scheduling detached tasks. Furthermore, we have shown that the overhead for our improvements is insignificant compared to task execution. Our improvements effective in the Spoofax PIE build script. In test the feedback time was reduced by 1800 ms in certain situations when compared to the old algorithm. The new algorithm avoids processing tasks that the user has detached. This allows the PIE runtime to keep operating efficiently over time. In the context of the Spoofax IDE, toggling observability creates the opportunity to implement various quality of life features such as renaming file and projects.

Further more, we have expanded PIE with a dependency graph visualizer and a prototype for garbage collection. We have related our work to Adapton and Incremental computation in general. Finally, we have suggested future work with respect to execution algorithms, the PIE language, and in approaching the analysis of incremental build algorithms.

Bibliography

Bellman, Richard (1954). "Some Applications of the Theory of Dynamic Programming - A Review". In: *ior* 2.3, pp. 275–288. DOI: http://dx.doi.org/10.1287/opre.2.3.275.

Ellson, John et al. (2001). "Graphviz - Open Source Graph Drawing Tools". In: *gd*, pp. 483–484. DOI: http://link.springer.de/link/service/series/0558/bibs/2265/22650483.htm.

Epperly, G. K. Kumfert T. G. W. (2002). *Software in the DOE: The Hidden Overhead of "TheBuild"*. Tech. rep. Lawrence Livermore National Laboratory.

Erdweg, Sebastian, Moritz Lichter, and Manuel Weiel (2015). "A sound and optimal incremental build system with dynamic dependencies". In: *OOPSLA*, pp. 89–106. DOI: http://doi.acm.org/10.1145/2814270.2814316.

Erdweg, Sebastian, Tijs van der Storm, et al. (2013). "The State of the Art in Language Workbenches - Conclusions from the Language Workbench Challenge". In: *SLE*, pp. 197–217. DOI: http://dx.doi.org/10.1007/978-3-319-02654-1_11.

Hammer, Matthew A., Joshua Dunfield, et al. (2015). "Incremental computation with names". In: *OOPSLA*, pp. 748–766. DOI: http://doi.acm.org/10.1145/2814270.2814305.

Hammer, Matthew A., Yit Phang Khoo, et al. (2014). "Adapton: composable, demand-driven incremental computation". In: *PLDI*, p. 18. DOI: http://doi.acm.org/10.1145/2594291.2594324.

Jain, Hemant K. and Anjana Gosain (2012). "A comprehensive study of view maintenance approaches in data warehousing evolution". In: *sigsoft* 37.5, pp. 1–8. DOI: http://doi.acm.org/10.1145/2347696.2347705.

Kats, Lennart C. L. and Eelco Visser (2010). "The Spoofax language workbench: rules for declarative specification of languages and IDEs". In: *OOPSLA*, pp. 444–463. DOI: 10.1145/1869459.1869497.

Konat, Gabriël, Sebastian Erdweg, and Eelco Visser (2018). "Scalable incremental building with dynamic task dependencies". In: *kbse*, pp. 76–86. DOI: https://doi.org/10.1145/3238147.3238196.

Konat, Gabriël, Michael J. Steindorfer, et al. (2018). "PIE: A Domain-Specific Language for Interactive Software Development Pipelines". In: *programming* 2.3, p. 9. DOI: 10.22152/programming-journal.org/2018/2/9.

McIntosh, Shane, Bram Adams, and Ahmed E. Hassan (2010). "The evolution of ANT build systems". In: *msr*, pp. 42–51. DOI: http://dx.doi.org/10.1109/MSR.2010.5463341.

Miller, Peter (n.d.). *Recursive Make Considered Harmful*.

Mokhov, Andrey, Neil Mitchell, and Simon L. Peyton Jones (2018). "Build systems à la carte". In: *PACMPL* 2.ICFP. DOI: https://doi.org/10.1145/3236774.

Mokhov, Andrey, Neil Mitchell, Simon L. Peyton Jones, and Simon Marlow (2016). "Non-recursive make considered harmful: build systems at scale". In: *haskell*, pp. 170–181. DOI: http://doi.acm.org/10.1145/2976002.2976011.

Ramalingam, Ganesan and Thomas W. Reps (1993). "A Categorized Bibliography on Incremental Computation". In: *POPL*, pp. 502–510.

Richard M. Stallman Roland McGrath, Paul D. Smith (2016). *GNU Make. A Program for Directing Recompilation*. URL: https://www.gnu.org/software/make/manual/make.pdf.

AST abstract syntax tree

DSL Domain Specific Language

IDE Integrated Development Environment

IC Incremental Computation

DAG Directed Acyclic Graph

PIE Pipeline for Interactive Environments

JVM Java Virtual Machine

JMH Java Microbenchmark Harness

JIT Just-In-Time

Appendix A

Appendix

A.1 Dependency graph of the Spoofax Pipeline for a small language

Figure A.1 is a dependency graph created during the execution of our spoofax pipeline at state 0110. This is a smaller language then the one used for our benchmark. The language used in our benchmark generates graphs to big to generate.

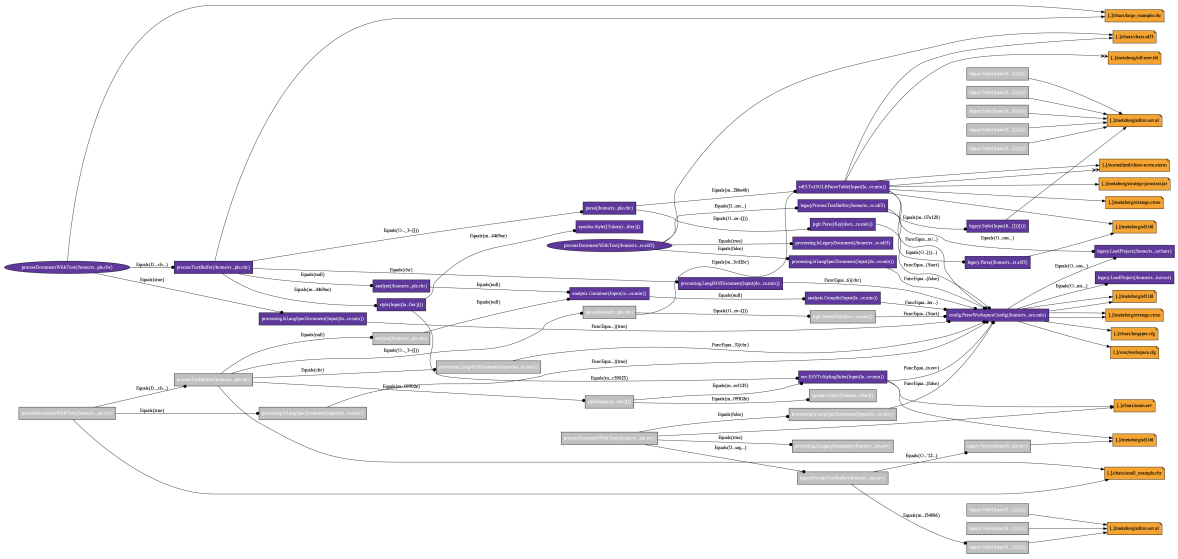


Figure A.1: A Dependency graph for the 'char' language of the pipeline generated by the visualizer. It is similar but smaller to the language used in our Spoofax Benchmark.