Level Graphs

Generating Benchmarks for Concurrency Optimizations in Compilers

Andrés Goens, Sebastian Ertel, Justus Adam, Jeronimo Castrillon
Chair for Compiler Construction
Technische Universität Dresden
Dresden, Germany
{first}.{last}@tu-dresden.de

Abstract

Benchmarks are needed in order to test compiler and language-based approaches to optimize concurrency. These have to be varied, yield reproducible results and allow comparison between different approaches. In this paper, we propose a framework for generating synthetic benchmarks that aims at attaining these goals. Based on generating code from random graphs, our framework operates at a high level of abstraction. We test our benchmarking framework with a usecase, where we compare three state-of-the-art systems to optimize I/O concurrency in microservice-based software architectures. We show how using our benchmarks we can reliably compare between approaches, and even between the same approach using different coding styles.

1 Introduction

Modern computing systems, especially heterogeneous ones, are parallel and rely on concurrency for ensuring an efficient execution. As such, both industry and academia spend a considerable effort in tools that identify, extract and optimize concurrency in applications. In order to asses the efficacy of a tool or framework, however, we have to evaluate it using benchmarks. Unfortunately, to the best of our knowledge, there are no satisfactory benchmarking options that are varied, yield reproducible results and allow a fair comparison between approaches, all at the same time.

When testing a tool or approach, we should ideally use ample benchmarks, spanning a wide range of typical usecases. Without variety in the benchmarks, we cannot guarantee the applicability of a tool being evaluated. Similarly, results obtained through a benchmark should be reproducible. Computer science should be treated as an empirical science, and without reproducibility, the scientific rigor of a test is lost.

MULTIPROG'18, January 24, 2018, Manchester, UK

Optimizing concurrency is a very important topic. As such, there is a myriad of different approaches and solutions to problems of optimizing concurrency. Thus, it is imperative that we are able to compare fairly between the different approaches. For benchmarks this means that they need to be portable and allow comparison between such approaches.

Existing benchmarks, like PARSEC [1] or SPLASH-2 [10], are limited to a small set of problems and inputs which can quickly become dated [1]. Furthermore, they are limited to an implementation of concurrency, e.g. a specific threading library, making it difficult to compare to other approaches. On the other hand, companies commonly use priopietary code and input as a benchmark (e.g. [6]). While these benchmarks are representative of a problem domain and usually up-to-date, they make results impossible to reproduce for other researchers, and much less to compare to other approaches. All of this is not surprising, since companies usually spend several person-decades to produce a codebase that efficently leverages the concurrency in their problems. Any good benchmark that is varied and up-to-date would need to reflect this effort.

In this paper we propose an alternative solution to the problem of missing benchmarks. Our proposed solution involves generating synthetic benchmarks from random graphs, in a way that we believe captures the general structure of complex applications with considerable amounts of concurrency. We do this at a high level of abstraction, which we believe is enough to evaluate the efficacy of general approaches. By releasing the tool to the public domain and using deterministic pseudo-random-number-generation, we can achieve reproducible results. Furthermore, by using a two-tiered approach, generating a graph first and then code from it, we can produce comparable code for different frameworks, allowing us to compare between them. This is the other advantage of our graph-based approach, besides capturing the structure of the applications.

To evaluate our benchmarks, we look at a usecase from the domain of microservices. In this domain, performance is heavily influenced by the concurrent execution of I/O calls. We generate code to evaluate and compare three frameworks for extracting and exploiting this concurrency in I/O from microservice-based applications. Additionally, using the high level of abstraction of our tool, we show how we can produce

code with the same structure using different abstractions for concurrency. Experiments show how this can affect the optimization potential from these frameworks.

The rest of the paper is structured as follows: Section 2 introduces the ideas and foundations of our code-generating framework. Section 3 introduces the usecase and its evaluation. Finally, Sections 4 and 5 discuss related work and conclude the paper, respectively.

2 Random Code Generation

In this section we explain the design of a tool that generates random code following a specific structure. In our benchmark tool, we generate programs that aim to resemble the typical structure of applications with complex concurrent interactions, as motivated by the usecases. While we based our tool on concurrent applications as ocurring in microservice-based architectures, we believe the concepts are more general and apply to most types of concurrent applications. We hope that companies building such architectures will find our tool appealing for their testing and enrich it with their insights to make the generated programs capture more real-world characteristics, even in different domains. For this reason, the code described in this section has been released[1] under an open-source license.

For this tool we derive the structure of the target programs from applications which process data from different sources in multiple layers. To build such benchmark programs, our tool generates source code out of random graphs. We do this in two steps, first generating the graph and then, the source code out of it. This allows us to generate code variants in different languages with the same structure.

For the purpose of code generation, we first introduce a special type of graph, a *Level Graph*. We believe this reflects the layered structure of our target applications. Afterwards, we explain its concept and present an algorithm to generate random Level Graphs.

2.1 Level Graphs

Generic graphs are too general to describe the call graphs of the typical concurrent applications. In order to better describe their structure we need a concept that is more specific. By analyzing call-graphs in usecases from the microservice domain, we noticed that most applications can usually be described by trees with a small amount of edges, compared to the size of the graphs themselves. In particular, edges are local in a sense that we will try to formalize. We call our approach to formalizing graphs with this structure *Level Graphs*. Since not all call-graphs are trees, we take rooted directed acyclic graphs as a base instead. A Level Graph is thus a rooted, directed acyclic graph with integer labels in the vertices. These labels are called levels, and we require that edges only go from a lower to a higher level. In particular, there can be no edges between two nodes that are at the same level. The levels in a Level Graph are a means of formalizing the data-locality observed in call-graphs, e.g. by bounding the level-difference between two nodes where an edge is present. They also impose constraints on the scheduling of the nodes, representing those imposed by the way the code is written.

2.2 Random Graph Generation

To generate random Level Graphs, we base our algorithm on a basic graph construction, the join of two graphs, usually denoted by $G_1 + G_2$. If $G_i = (V_i, E_i), i = 1, 2$, then it is defined as:

$$G_1 + G_2 := (V_1 \dot\cup V_2, E_1 \cup E_2 \cup \{(v, w) \mid v \in V_1, w \in V_2\} \cup \{(v, w) \mid v \in V_2, w \in V_1\}),$$

where $V_1 \dot\cup V_2$ denotes the *disjoint union* of V_1 and V_2.

Similarly, we can define the graph join of two Level Graphs by changing the additional edges included in the join so that they respect the Level Graph structure. Let $L_i = (V_i, E_i), l_i : V_i \to \mathbb{N}, i = 1, 2$ be two Level Graphs, where l_i denotes the level label function, and $L_1 + L_2 = (V_{1+2}, E_{1+2})$ be the (conventional) graph join of them. Then we define the Level Graph join as:

$$L_1 \oplus L_2 := (V_{1+2}, \{e = (v, w) \in E_{1+2} \mid l(v) < l(w)\})$$

To generate random Level Graphs we create levels and join them together using a construction similar to the graph join construction. The graph join construction gives us all possible edges between two Level Graphs. However, we do not want to have all possible edges. Instead, we want to have only a random subset of them.

Our basic approach for random Level Graphs follows the philosophy of the Gilbert approach [4]. This means that for a given probability $p \in [0, 1]$ every (possible) edge is present independently with probability p. If our random Level Graphs are to represent the call graphs of typical applications processing data, our model needs to consider that data is used much more in a local fashion. Thus, instead of a fixed probability p, we assign a probability function based on the levels. More precisely, we have a set $p_{i,j} \in [0, 1]$ for $i, j \in \mathbb{N}$ and the independent probability of the edge $e = (v, w)$ in the Level Graph join is given as:

$$p((v, w)) = p_{l(v),l(w)}.$$

This means, in particular, that $p_{i,j} = 0$ for $i \geq j$. For example, for the tests in this paper we used the probabilities $p_{i,j} = 2^{i-j}$ for $i < j$. This makes sure that edges spanning several levels become increasingly less likely the more levels they span.

Using the random Level Graph join construction we can combine null Level Graphs (a Level Graph with one level,

[1] https://github.com/goens/rand-code-graph

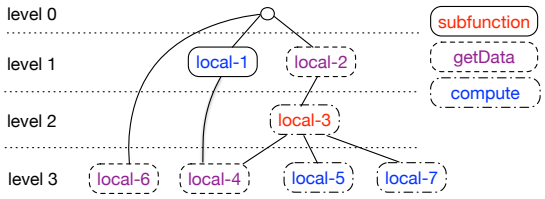

Figure 1. An example of a Level Graph.

and thus, no edges) multiple times to obtain a Level Graph with multiple levels.

Note that this graph is not guaranteed to be connected. It might also have several "root-like" nodes, i.e. nodes without predecessors. We fix these two issues by making the graph rooted. We do this by creating a root node one level below all current existing levels and connecting all nodes without predecessors to this root node.

Figure 1 illustrates a Level Graph with four levels. The nodes have been labeled with code types, which will be introduced in the next subsection.

2.3 Code Generation

To generate code, we first extend Level Graphs to have labels for the type of code that each node will represent and randomly assign labels to a generated Level Graph. The annotations depend on the usecase, of course. For the purposes of illustration, consider the following five different annotation types: computation, data-source, data-fetch (side-effect), subfunction and map. The first three generate code that simulates external function calls, involving I/O and pure computations. Subfunction/map nodes generate a call to/a map over a local function, for which a new random graph is generated in turn. The choice of these annotation types is, to some extent, arbitrary. They could be replaced by a similar set of code annotations in a different setting.

In order to generate code from a rooted and code-annotated Level Graph, we only need to traverse the nodes and generate functions with dependencies in accordance to the graph. If there are subfunctions labeled as such in the graph, we need to do this recursively for all corresponding subgraphs as well.

To traverse a graph, we sort the nodes in it by levels. This sorting is also a topological sort, by the defining properties of Level Graphs. We traverse the graph bottom-up, starting at the highest level, i.e. in reverse-topological order. At every level, we serialize the nodes in that level as function calls, as well as all predecessors of a node as the arguments of the function call.

The example of Figure 1 can be converted into the following Clojure code. It shows only the code for the graph depicted, omitting the subfunction generated for `subfun-3`.

```
(let [local-4 (get-data "source" 100)
      local-5 (compute 100)
      local-6 (get-data "source" 100)
      local-7 (compute 100)]
  (let [local-3
        (subfun-3 local-4 local-5 local-7)]
    (let [local-1 (compute 100 local-4)
          local-2 (get-data "source" 100 local-3)]
      (get-data "source"
                100 local-1 local-2 local-6))))
```

This approach is not limited to Clojure nor to functional languages. For example, the same graph could generate Java code similar to the following snippet:

```
{
  Object loc4 = get_data("source",100);
  Object loc5 = compute(100);
  Object loc6 = get_data("source",100);
  Object loc7 = compute(100);
  {
    Object loc3 = subfun_3(loc4,loc5,loc7);
    {
      Object loc1 = compute(100,loc4);
      Object loc1 =
          get_data("source",100,loc1,loc2,loc6);
    }
  }
}
```

3 Evaluation

In order to evaluate our code-generation framework, we generate code for usecases from the microservice domain. Using this code we evaluate and compare three frameworks with similar aims: To improve I/O in microservice architectures by leveraging concurrency in I/O calls and batching them together. The frameworks we evaluate are Haxl, by Facebook [6], which is an EDSL in Haxl, and two open-source academic frameworks inspired by Haxl, Muse [5] and Ÿauhau [3], which are both based on Clojure. Thus, we need to generate Haskell code for Haxl, and Clojure code for Muse and Ÿauhau.

Since our Level Graphs have a high level of abstraction, there are different ways we can serialize them as code for Haskell and Clojure. In particular, we can use a style using monads, or use a more direct approach with applicative functors. In the following we discuss these different ways, which we will call code-style variants.

3.1 Code-Style Variants

In the following we use the example of Figure 1 to depict the code that we generate for all different frameworks. Again, we omit the subfunction generated for `subfun-3`.

3.1.1 Applicative Code Style

The applicative code style makes explicit use of the applicative functor or the respective concept thereof.

Haxl The applicative version of the Haxl code executes the functions of a single level using the applicative functor and

uses a single do block to connect the levels with each other via variables. This is also the version of code favored by the authors of the Haxl paper, since at the time there was no support for desugaring do into applicative functors [6]. This only became available with GHC version 8 [7].

```
do
  -- level 3
 (local4, local5, local6, local7) <- (,,,)
   <$> getData "source" [100] <*> compute [100]
   <*> getData "source" [100] <*> compute [100]
 -- level 2
 local3 <- subfun3 [100, local4, local5, local7]
 -- level 1
 (local1, local2) <- (,)
    <$> compute [100, local4]
    <*> getData "source" [100, local3]
 -- level 0
 getData "source" [100,local1,local2,local6]
```

Note that the parameters to the function calls are also controlled by the code generation framework. For the `compute` function the parameter controls the execution time and in the case of the `getData` function the parameter is the ID of the data source to be queried. The choice of 100 here is arbitrary. In future work we plan to find accurate models for timings, including timing variations. However, this bears no consequence in the evaluation of this paper, since we only care about the number of I/O operations, not the total execution time.

Muse The syntactically equivalent Muse code uses a single `mlet`-block of the Cats library which is equivalent to Haskell's do block[2].

```
(cats/mlet
   [; level 3
    [local-4 local-5 local-6 local-7]
      (cats/<$>
        clojure.core/vector
        (get-data "source" 100)
        (cats/<$> (compute (cats/return 100)))
        (get-data "source" 100)
        (cats/<$> (compute (cats/return 100))))
    ; level 2
    local-3
      (subfun-3 local-4 local-5 local-7)
    ; level 1
    [local-1 local-2]
      (cats/<$>
           clojure.core/vector
           (cats/<$>
             (compute (cats/return 100)
                      (cats/return local-4)))
           (get-data "source" 100 local-3))]
   ; level 0
   (get-data "source" 100 local-1 local-2
             local-6))))
```

[2]https://funcool.github.io/cats

The implications to the programming style are quite impacting because the programmer has to make use of monadic concepts which is uncommon in Clojure.

Ÿauhau In contrast to Muse, Ÿauhau uses Clojure code with the normal `let` construct.

```
(let
   [; level 3
    [local-4 local-5 local-6 local-7]
      (vector (get-data "source" 100)
              (compute 100)
              (get-data "source" 100)
              (compute 100))
    ; level 2
    local-3 (subfun-3 local-4 local-5 local-7)
    ; level 1
    [local-1 local-2]
      (vector (compute 100 local-4)
              (get-data "source" 100 local-3))]
   ; level 0
   (get-data "source" 100 local-1 local-2
             local-6))))
```

3.1.2 Monadic Code Style

The monadic version of the code uses no applicative functor or any other similar abstractions. It is, among those supported by the frameworks, the code style that is most natural to the developer.

Haxl The monadic Haxl code binds the result of every node in the graph in a single large do block.

```
do
 -- level 3
 local7 <- compute [100]
 local6 <- getData "source" [100]
 local5 <- compute [100]
 local4 <- getData "source" [100]
 -- level 2
 local3 <- subfun3 [100, local4, local5,
                    local7]
 -- level 1
 local2 <- getData "source" [100, local3]
 local1 <- compute [100, local4]
 -- level 0
 getData "source" [100,local1,local2,local6]
```

Note that the order of serialization still remains the same as in the case of the applicative code style. Therefore, the code preserves the concept of variable locality.

Muse The syntactically equivalent Muse code uses nested `mlet`-bindings, one for each level.

```
(cats/mlet [; level 3
      local-4 (get-data "source" 100)
      local-5 (compute 100)
      local-6 (get-data "source" 100)
      local-7 (compute 100)]
 (cats/mlet [; level 2
      local-3 (subfun-3 local-4 local-5 local-7)]
```

```
   (cats/mlet [; level 1
      local-1 (compute 100 local-4)
      local-2 (get-data "source" 100 local-3)]
 ; level 0
(get-data "source" 100 local-1 local-2 local-6))
```

We generate nested `mlet` blocks because it is a more natural "Clojure-resque" style of programming that one would typically find. The runtime effect is the same as defining all variables in a single `mlet` block. The code shows that a lot of the monadic aspects are gone from the code and it is much more readable than its applicative counterpart.

Ÿauhau Once more, the code for Ÿauhau is the same as for Muse but with a plain Clojure style using nested `let` expressions instead of `mlet`.

3.2 Experimental Setup

To compare the frameworks, we execute comparable code generated for all three frameworks and measure their performance. In order to do so, out of the same graph we produce code for all six presented combinations of framework and code variant. Our experiment framework provides implementations for the functions `getData` and `compute`. These functions are no-ops for the experiments that count the total number of performed I/O calls. We use this as a measure of the efficacy of the framework, as was established by the evaluation in [3, 6] (lower is better). In particular, the total execution time is irrelevant.

In our experiments, we vary the number of levels in a graph from 1 to 20. For each fixed number of levels we generate 20 different random graphs. Thus, for each fixed number of levels, we report the average number of I/O calls executed by each framework in the corresponding 20 runs, which we denote as #I/O calls (avg.).

3.3 Results

Figure 2 shows the difference in batching between the different frameworks, and for each framework, between the monadic and the applicative versions. Neither Haxl (pre GHC 8) nor Muse have support for batching requests in the monadic code style. Therefore, we observe the same performance for both frameworks when using a monadic code style. Code written in this form does not benefit from request batching and display the efficiency of a sequential execution (which we omitted from the plot for readability).

We also ran both frameworks with the same code graphs transformed into code which heavily favors the applicative code style.

In terms of batching, Ÿauhau is superior than Haxl and Muse even when following the applicative style. In this set of experiments, Ÿauhau outperforms the monadic versions by 49% for programs with more than a single level. This is due to the fact that finding the minimal number of rounds with the maximum number of fetches per round requires non-trivial code analysis and changes. Ÿauhau provides such a data dependency analysis at compile time, which enables it to find the most efficient solution, whereas Haxl and Muse only provide a mechanism for optimizing this at runtime that is less efficient.

The difference in performance, i.e., number of I/O calls, for both implementations in Haxl and in Muse display a clear tradeoff. Code executes most efficiently when written in an applicative form, but the monadic code style is necessary to make the code understandable, i.e., concise and maintainable. In Ÿauhau, in turn, the algorithms are independent of the structure of the code itself. For that reason, the new Haskell compiler (GHC version 8) desugars monadic binds into applicative functor. Therefore, we used the monadic variant of the created code and compiled it with GHC 8.0.1. The plot in Figure 2 shows that this increases performance but only up to the level of applicative Haxl code, still below the performance of Ÿauhau. This is due to the structure of the source graph (Level Graph). As a direct result, when we apply applicative do to the monadic version it has to insert at least one bind per level. The resulting code is essentially the same as the applicative style code that we generate because of a reordering limitation that comes about by conservative optimizations not knowing the exact dependencies. Therefore, applicative-do code inherits the same performance as the applicative style. We argue that our level graph encapsulates the notion of variable locality, i.e., a variable is used in the code close to where it is was created, as a natural way of writing code. Reordering the statements such that Haxl performs as good as Ÿauhau would entirely break this notion and produce code that is hard to maintain.

We see how our level graphs framework provides means of comparing different frameworks based on different languages in a meaningful way. In particular, it allowed us in this usecase to determine how the code style affects performance in the frameworks. This is a virtue of the high level of abstraction in the graphs.

4 Related Work

As explained in the introduction Benchmarks, like PARSEC [1] or SPLASH-2 [10] are limited to a small set of problems and inputs which can quickly become dated [1]. Furthermore, they are limited to an implementation of concurrency, making it difficult to compare to other approaches.

However, several approaches also exist to generate synthetic benchmarks through graphs with the typical structure of software applications. The most prominent ones include TGFF [2] or SDF^3 [9]. These tools are designed to test methods for scheduling these graphs, but do not generate code from the graphs. Similarly, there is previous work on generating random programs for particular languages [11] or from grammars [8]. However, these tools are very usecase specific. In praticular, they operate at a much lower level of

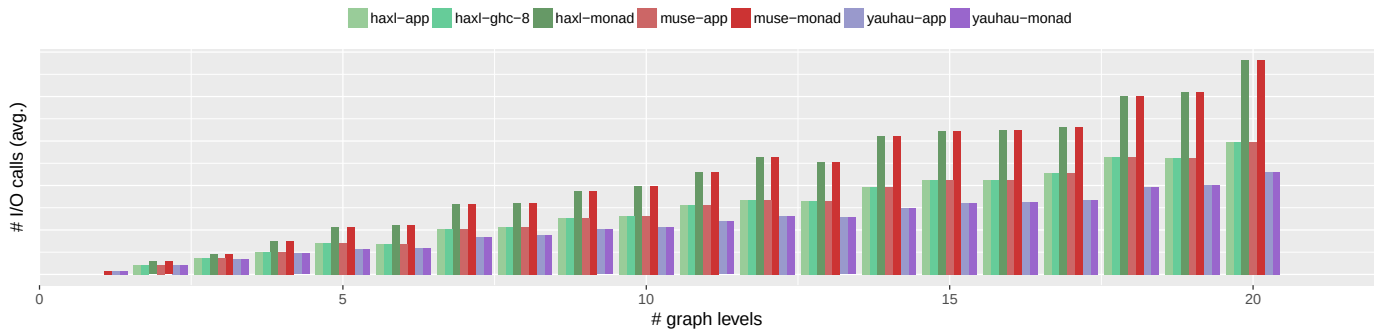

Figure 2. Monadic vs. Applicative Code Style

abstraction, which allows only to generate code for a specific language, and not benchmarks that can be used to compare approaches implemented in different languages. Additionally, the low level of abstraction makes it difficult to adapt the approach to a particular problem domain, like I/O in microservice architectures. To the best of our knowledge, our Level Graphs approach is the first high-level framework that can be adapted to a particular problem, while producing actual code for different languages, which can be used to compare different approaches.

5 Conclusion

In this paper, we introduced a methodology to generate code from random graphs at a high level of abstraction. We argued why our Level Graph approach captures concepts like data locality and allows to extract the structure of the use case. We then also show with an experiment, how our method provides means of comparing different frameworks based on different languages in a meaningful way. In particular, the high level of abstraction allowed us to produce code in different styles. This in turn permitted us to determine how the code style affects performance in different frameworks for optimizing I/O calls in microservices. We see thus that using our approach with Level Graphs we can generate synthetic benchmarks that are varied and yield reproducible results, while allowing different tools based on different languages to be compared.

5.1 Future Work

An issue that is still open is how closely the graphs resemble actual applications, as well as how many different types of applications can be covered by such an approach. To deal with this, we plan to extend our work by validating the structure of the graphs. We plan to mine from different codebases and abstract the high-level information in an automated fashion. This way, we can use a learning approach to find typical Level Graph structures in real code from real benchmarks.

Acknowledgments

This work was supported in part by the German Research Foundation (DFG) within the Collaborative Research Center HAEC and the Center for Advancing Electronics Dresden (cfaed).

References

[1] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[2] R. P. Dick, D. L. Rhodes, and W. Wolf. Tgff: task graphs for free. In *Proceedings of the 6th international workshop on Hardware/software codesign*, pages 97–101. IEEE Computer Society, 1998.

[3] S. Ertel, A. Goens, J. Adam, and J. Castrillon. Compiling for concise code and efficient i/o. In *Proc. of the 27th Int. Conf. on Compiler Construction (CC 2018)*, CC 2018, New York, NY, USA, Feb. 2018. ACM.

[4] E. N. Gilbert. Random graphs. *The Annals of Mathematical Statistics*, 30(4):1141–1144, 1959.

[5] A. Kachayev. Reinventing haxl: Efficient, concurrent and concise data access. Technical report, 2015. URL https://www.youtube.com/watch?v=T-oekV8Pwv8. [Online; accessed 4-May-2017].

[6] S. Marlow, L. Brandy, J. Coens, and J. Purdy. There is no fork: An abstraction for efficient, concurrent, and concise data access. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 325–337, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2873-9. doi: 10.1145/2628136.2628144. URL http://doi.acm.org/10.1145/2628136.2628144.

[7] S. Marlow, S. Peyton Jones, E. Kmett, and A. Mokhov. Desugaring haskell's do-notation into applicative operations. In *Proceedings of the 9th International Symposium on Haskell*, Haskell 2016, pages 92–104, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4434-0. doi: 10.1145/2976002.2976007. URL http://doi.acm.org/10.1145/2976002.2976007.

[8] B. McKenzie. Generating strings at random from a context free grammar. 1997.

[9] S. Stuijk, M. Geilen, and T. Basten. SDF^3: Sdf for free. *Application of Concurrency to System Design, International Conference on*, 0:276–278, 2006. ISSN 1550-4808. doi: 10.1109/acsd.2006.23. URL http://dx.doi.org/10.1109/acsd.2006.23.

[10] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, pages 24–36. IEEE, 1995.

[11] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. In *ACM SIGPLAN Notices*, volume 46, pages 283–294. ACM, 2011.