Accelerating Static Taint Analysis with CUDA Toolkit

Project Report

Jonas Svenningsen, Nicklas Hugöy, Thorulf Neustrup

Aalborg University

AALBORG UNIVERSITY
STUDENT REPORT

Electronics and IT
Aalborg University
http://www.aau.dk

Title:
Accelerating Static Taint Analysis with CUDA Toolkit

Theme:
Distributed Systems

Project Period:
Spring Semester 2022

Project Group:
cs-22-ds-10-02

Participant(s):
Jonas Askløf Svenningsen
Nicklas Emil Hjortshøj Hugöy
Thorulf Neustrup

Supervisor(s):
René R. Hansen
Danny B. Poulsen
Anton Christensen

Copies: 1

Page Numbers: 80

Date of Completion:
June 17, 2022

Abstract:

This paper continues the work of a previous paper [16] in off-loading static taint analysis to a GPU. As the previous attempt lacked performance gains from a matrix-based algorithm, a new attempt makes use of CUDA, a GPU toolkit by Nvidia. To better utilize the computational pover of GPUs, the CFG is designed in such a way that all the different types of CFG nodes are represented with the same type of node. Two CUDA implementations were made, bit-cuda and cuda work-list. The CPU counterpart was optimized to increase the fairness of comparing performance between CPU and the two GPU implementations The performed benchmarks shows that the CPU outperforms the GPU on smaller programs up until 700.000 CFG nodes where the GPU starts to perform slightly better than the CPU.

Summary

Static program analysis is built into most modern day compilers. Some analyses are used to perform program transformations in order to optimize the programs during compilation, while other types of analysis looks at potential errors and security issues within the program. The analysis which is used in this paper is static taint analysis. This analysis is performed on a small programming language SC which was developed in the previous work of off-loading static taint analysis to the GPU. The previous work attempted to transform the taint analysis problem into matrices and use the GPU to perform matrix operations in order to reach a least fix-point. However, the matrix approach performed many times slower than the CPU baseline which leads to this paper attempting a more direct way in controlling the GPU with use of CUDA, a GPU toolkit by Nvidia.

The language SC were extended to increase the complexity of programs able to be written. In order to transfer the CFG nodes to the GPU another way of storing the data was needed since the initial CFG creation was done with individual nodes being heap allocated. This was changed such that it is stored contiguously in heap memory and is then simple to copy to the GPU device memory. Since the GPU operates differently compared to CPU, the CFG was modified to remove all polymorphic data to better take advantage of the GPU's Single Instruction Multiple Threads(SIMT) execution model.

Two different approaches were implemented on the GPU, bit-cuda and cuda work-list. The bit-cuda implementation assigns one CFG node per thread which makes it possible to propagate taint for all CFG nodes simultaneously. This method however have many threads not performing any actions as not all CFG nodes have taint to propagate for each iteration, therefore cuda work-list analysis was implemented as well. This approach makes use of a work-list which have threads operating on data in a single index of the work-list. This limits the number of threads used for the analysis to the size of the work-list. However, the work-list data structure were challenging to implement as it is accessed by multiple threads at once. These threads would all try and add new nodes to the work-list resulting in race conditions. Since many threads tries to add elements, a queue structure would be difficult to use. Therefore, adding elements to the work-list was implemented using CUDA's atomic operations. Before benchmarking and comparing performance of CPU and GPU, it is necessary to optimize the CPU algorithm to make the comparison more fair. This optimization revamped the entire CFG used on the CPU. With the new optimized CPU algorithm, benchmarks were performance on different programs which attempts to be best case and worst case programs to analyse for the GPU. The CPU generally outperforms the GPU on smaller programs but when the size of the program increases the GPU starts to outperform CPU when reaching 700.000 CFG nodes.

These results significantly improved the results from the previous work on a factor of a thousand. The improvement was not enough to beat the CPU baseline consistently, the GPU is only faster on some large programs. Despite this the GPU implementations is still proposed to be useful if applied in a smart manner. As the GPU could run parallel with CPU gaining a higher computational throughput than either could alone. Potentially solving different steps of compilation at the same time.

Contents

Chapter 1

Introduction

Static program analysis is present in most compilers, and can be the first step in optimization or bug prevention. This helps improve code quality significantly, by detecting both small code errors and maintain code properties such as security or performance. As systems grow, reasoning about these properties becomes impractical, making static program analysis an important tool. But with larger systems analysis becomes more time consuming and could become prohibitively slow. Therefore, steps must be taken to take better take advantage of the computational power available. The CPU is usually fully occupied during compilation as compilers takes advantage of parallel computing. But the GPU a massively parallel processing unit sits idle despite boasting higher computational throughput than the CPU.

This paper follows up on an attempt to take advantage of the GPU by running a taint analysis using matrix multiplication. The matrix reduction attempt failed to make good utilization of GPU resources, so instead this paper looks at other ways to create algorithms for GPUs. The most general approach, the CUDA toolkit is used to design and implement taint analysis algorithms for the GPU.

However, it is required to have some prior knowledge of the previous work attempting matrix multiplication as parts of the work from that paper is reused in this paper. Therefore a preliminary section is provided to supply the knowledge of the previous paper and the reused aspects of it.

Chapter 2

Preliminary

In this chapter, the preliminary work and concepts will be introduced. First, the previous work that this report builds on will be introduced. Followed by an introduction of the basics of CUDA, a GPU-toolkit by Nvidia. Then some related works will be researched. Finally optimization approaches are considered for the previous work to evaluate potential for further work.

2.1 Previous Work

This paper continues the work started in "GPU-accelerated Taint Analysis"[16]. The previous work attempted to perform taint analysis on GPU with use of GPU matrix libraries cuBLAS [11] and cuBool [7]. Following the successes of similar works also using matrix approaches like EigenCFA [14]. The approach was to reduce the analysis into a matrix multiplication problem and use the GPU to quickly perform matrix operations. In order to compare the performance of the taint analysis on the GPU, a worklist-based CPU analysis were implemented.

The library cuBLAS was used to create the initial GPU implementation. CuBLAS is a dense basic linear algebra Subroutine library. The library provides helper functions to manage data allocations and transferring between CPU and GPU. This implementation shows that translating taint analysis into a matrix multiplication problem is possible but the performance of this method was slower in comparison to the CPU counterpart. This approach operates on dense matrices but the data in the matrices were sparse. This results in many wasted computations when computing the fix-point. Therefore another implementation was created which operates on sparse matrices instead.

For this approach cuBool was used. CuBool is different as it works on sparse boolean matrices instead of dense floating point matrices which cuBLAS operates on. However it was discovered that cuBool did not fully support the features necessary for performing the needed matrix operation and was therefore required to create a workaround. This additional overhead caused the analysis using cuBool to be significantly slower than the cuBLAS approach. It was not definitive that a sparse matrix approach is implausible due to the overhead. But the work still concluded that transforming taint analysis into a matrix multiplication problem is an infeasible approach for static analysis on GPU.

2.1.1 Improvements for the Analysis

Additional improvements for the different analysis were made to ensure that the process of using matrices were insufficient. Through these improvements a general optimization of generalizing variable names were made. The process of this will be covered here. Lastly the CPU analysis also received improvements making it significantly faster in comparison to the performance of the algorithm of the previous work. This improvement caused there to be an even larger performance gap between the initial GPU analysis implementation and the CPU, showing that another approach is needed in order to receive a speed up in performance with the use of GPU.

Optimizing CuBLAS Solution

Additional work is put into optimizing the CuBLAS implementation from previous work[16]. Some of the most prevalent optimizations that were considered are.

- **Batch scheduling** The current implementation requires many relatively small matrix calculations, for each iteration of the algorithm. Each of these calculations have some scheduling overhead as the host has to schedule the task on the device for each necessary calculation. CUDA and in extension CuBLAS comes with tools to batch schedule these calculations that guarantee parallel execution.
- **Contiguous memory transfers** By allocating all the matrices in a contiguous memory region. The data transfer from host to device can be done with a single call. Which reduces the transfer overhead to a minimum.
- **Using page-locked memory** The type of memory the data is copied from matters. CUDA requires the data to be located in page-locked memory allocation. Page-locked memory is pinned to main memory and can not be paged-out which regular paged memory can. If the data is located in a normal allocation, transfer has additional overhead.

Each of these potential optimizations were tested individually and compared to a control implementation without optimizations. These implementations have multiple measure points to give a better overview of specific speedups and slowdowns of an implementation. These tests were made with five sample runs for each implementation to even out fluctuations. However none of these optimizations resulted in a noticeable speed up.

- **Batch scheduling** was seemingly a good approach as it could improve the main bottleneck of the algorithm, which is the least fixed point algorithm. But this optimization did not improve the runtime. This can happen for two reasons, either the cuBLAS library already provides parallel execution, or each matrix operation can utilize the majority of GPU resources alone.
- **Contiguous memory transfers** failed to provide a speed up, despite improving copying times significantly, as additional overhead were cropped up. More importantly it negatively affected the matrix operations, making them significantly slower. Both of these issues might be solvable.
- **Using page-locked memory** due to longer allocation times, the time to allocate memory on the host memory slowed down the process significantly. Additional attempts to combine this optimizations with **Contiguous memory transfers** solved that problem but seemingly negatively affected matrix math operations. Making the overall results slower.

Despite these results more options for optimizations exist. Currently duplicate transfer function matrices are copied to the GPU. Duplicate matrices could be shared as they are immutable. As briefly mentioned, the matrix math is the primary slowdown on the system. This happens because the dense matrix algorithm keeps updating each node every iteration even if it has reached its local least fixed point. So work should be put into making changes to the least fixed point algorithm, that reduce redundant work. A change could be a work-list type solution, that is already well known to work on CPUs.

Variable reduction

The CuBLAS implementation works on state matrices that scale up depending on program size and number of variables. This means that both dimensions grow depending on the size of the program, as the number of variables is usually tied to the size of the program. Storing the variables this way creates ever larger matrices that take up more and more time to calculate during the least fixed point algorithm. So by reducing the matrix size, the overall algorithm could get a speed up as it directly reduces the required GPU resources per matrix operation.

Two approaches to reducing matrix sizes is considered, first cutting down on program size by **minimizing CFG**, the other approach reduces the number of variables **variable reduction**. **Minimizing CFG** is done by changing the CFG in a way such that it still represents the same behaviour. The simplest solution here is to remove the most common node type which is a no-op (no-operation) node. These nodes will be named propagation nodes, they do not alter program state and can be removed without altering behaviour as long as the control flow is maintained. Removing propagation nodes reduces both state matrices and the successor matrix. The second approach is **variable reduction** which aims to lower the number of variables. This can be done by taking advantage of the nature of variable scopes in C-like languages. If there are two unique variables in two different scopes then these two variables could be reduced into a single variable as the variables cannot interact. A reduction that only looks at function scopes would reduce the number of variables significantly.

When performing these optimizations it is worth to consider how they impact the CPU implementation. The **minimizing CFG** is not just an optimization for GPU implementations, but would also make CPU implementations faster. So this optimization should also be implemented for the CPU algorithm out of fairness.

Additionally considerations are taken whether these optimizations would make sense in the grand scheme of a full C language. **Minizing CFG** would likely work just fine, as it simply removes no-op nodes. However, **variable reduction** would become significantly more problematic due to C's pointers. A similar optimization that reduces variables would probably be possible, however the current approach is incompatible due to side effects in functions.

Optimizing CPU analysis

The previous work had a work-list CPU taint analysis algorithm which the GPU implementations is compared against. This CPU implementation is problematic in use for comparisons, as it is slow which artificially makes the GPU implementations look better. The old implementation runs 21.000 nodes in 6.25 seconds [16]. Compared to the optimized implementation which does the same in 3 milliseconds as seen in figure 6.1.

To reach the optimized CPU implementation, that is compared with in later sections, a few major optimizations were made. The main issue in the old implementation was the use of sets of strings to keep track of tainted variable names in the abstract program state. This would incur expensive string comparisons during look ups and union operations. This was resolved by replacing the sets with bit-vectors. To avoid string operations during analysis variables would be reduced into indexes that are used to index into these bit-vectors. This is done in the variable reduction step see section 5.2.

2.1.2 Programming language: SC

A new language was developed for the taint analysis to be performed on. This language was designed to be simple to implement but be broad enough to make the taint analysis non-trivial. The majority of SC was developed in the previous work, with the intention of copying the C syntax [16]. SC is a simple imperative programming language containing constructs such as functions, variables, assignments and while loops. The language only has integer variables and by extension only supports arithmetic expressions. Functions in the SC have no side effects, uses pass-by-value and does not support recursion. Since taint originates from external inputs to the program, a special symbol is used to indicate a taint source in SC programs. The grammar for SC can be seen on figure 2.1 and the semantics can be seen in appendix A.

2.1.3 Extension of SC

The language SC was developed and used as the language for the taint analysis of the previous work[16]. The language constructs of SC have been extended to include statically sized arrays. Arrays were added to increase the complexity of programs which can be written in SC. The extension of SC's grammar is highlighted in figure 2.1, this extension handles initialization of arrays and assigning values to an array index.

$$\begin{array}{lll}
\langle \mathit{prog} \rangle & ::= & \langle \mathit{funcDef} \rangle + \\[6pt]
\langle \mathit{funcDef} \rangle & ::= & \text{int } \mathbf{ID} \text{ (} \langle \mathit{param} \rangle\text{?) \{ } \langle \mathit{stmt} \rangle + \text{ \}} \\[6pt]
\langle \mathit{stmt} \rangle & ::= & \mathbf{ID} = \langle \mathit{expr} \rangle; \\
 & \mid & \text{while (}\langle \mathit{expr} \rangle\text{) \{ } \langle \mathit{stmt} \rangle + \text{ \}} \\
 & \mid & \text{if (}\langle \mathit{expr} \rangle\text{) \{ } \langle \mathit{stmt} \rangle + \text{ \} else \{ } \langle \mathit{stmt} \rangle + \text{ \}} \\
 & \mid & \text{return } \langle \mathit{expr} \rangle \text{ ;} \\
 & \mid & \text{int}[\mathbf{Integer}]\ \mathbf{ID} = \{\langle \mathit{args} \rangle\}; \\
 & \mid & \mathbf{ID}[\langle \mathit{expr} \rangle] = \langle \mathit{expr} \rangle; \\[6pt]
\langle \mathit{param} \rangle & ::= & \text{int } \mathbf{ID} \\
 & \mid & \langle \mathit{param} \rangle, \langle \mathit{param} \rangle \\[6pt]
\langle \mathit{args} \rangle & ::= & \langle \mathit{expr} \rangle, \langle \mathit{args} \rangle \\
 & \mid & \langle \mathit{expr} \rangle \\[6pt]
\langle \mathit{expr} \rangle & ::= & (\langle \mathit{expr} \rangle) \\
 & \mid & \langle \mathit{expr} \rangle\ \langle \mathit{op} \rangle\ \langle \mathit{expr} \rangle \\
 & \mid & \mathbf{ID} \text{ (} \langle \mathit{args} \rangle\text{?)} \\
 & \mid & \mathbf{Integer} \\[6pt]
\langle \mathit{op} \rangle & ::= & + \mid - \mid / \mid *
\end{array}$$

Figure 2.1: Grammar of SC[16] with array-extension highlighted with blue text

To convey how the arrays are handled in SC, an extension of the SC semantics is presented. First off, another environment is required in order to store variable names of arrays, along with a mapping of indexes to values. This environment env_A will be included in the other constructs of SC. The complete semantics of SC can be seen in appendix A where the extension to the original semantics are highlighted.

The first rule `arrayInit` handles the array initialisation. Here an array is constructed with a variable name, a size, and a number of array elements as expressions. First, each expression $e_1, ..., e_n$ is evaluated to an integer values $v_1, ..., v_n$. Then the array environment is updated with the array variable name a and the mapping is assigned with the values. An integer from zero to the number of values n are each mapped to one of the evaluated expressions, in the same order as they are defined. It is required that the number of expressions n is equal to the array size $\mathcal{Z}(l)$.

The `arrayAssign` rule handles the assignment of an array element to a new value. First, the array index expression e_1 is evaluated to the integer `i` and the expression e_2 to be assigned is evaluated to the integer v. It is required that i is a valid index in the array and should be a value between zero and the size in the array environment for a. Finally, the array environment is updated for array variable a where the value for the mapping for index i is changed to v.

The final rule `arrayExpr` handles when an array is used in an expression. The index expression e is evaluated to the integer i. The array variable name is used to access the array environment with the values of the array *vals* and the size of the array n. Again, it is a requirement that the index i is between zero and the array size n. Finally the expression return the value contained in the *vals* map for the value i.

$$env_A \in Env_A = Vars \rightharpoonup \mathbb{N} \times (\mathbb{Z} \rightharpoonup \mathbb{Z})$$

$$(arrayInit)\frac{\begin{array}{c} env_P, env_A, env_V \vdash e_1 \rightarrow_e v_1 \\ .... \\ env_P, env_A, env_V \vdash e_n \rightarrow_e v_n \end{array}}{\begin{array}{c} env_P, env_V \vdash \langle int\ a[l] = \{e_1, ..., e_n\}, env_A \rangle \rightarrow_s \\ \langle env_A[a \mapsto (n, [0 \mapsto v_1, ..., n-1 \mapsto v_n])], env_V, \bot \rangle \end{array}} \text{ where } \mathcal{Z}(l) = n$$

$$(arrayAssign)\frac{\begin{array}{c} env_P, env_A, env_V \vdash e_1 \rightarrow_e i \\ env_P, env_A, env_V \vdash e_2 \rightarrow_e v \end{array}}{\begin{array}{c} env_P, env_V \vdash \langle a[e_1] = e_2, env_A \rangle \rightarrow_s \\ \langle env_A[a \mapsto (n, vals[i \mapsto v])], env_V, \bot \rangle \end{array}} \text{ where } \begin{array}{c} env_A(a) = (n, vals) \\ 0 \leq i < n \end{array}$$

$$(arrayExpr)\frac{env_P, env_A, env_V \vdash e \rightarrow_e i}{env_P, env_A, env_V \vdash a[e] \rightarrow_e vals(i)} \text{ where } \begin{array}{c} env_A(a) = (n, vals) \\ 0 \leq i < n \end{array}$$

Abstract semantics

To create an analysis that will eventually terminate, the concrete semantics have to be abstracted in a monotonic fashion. To do this the abstract program state is modelled using a lattice. For this taint analysis the abstract state should keep track of whether each variable is tainted or not. For each variable $v \in Vars$ a lattice with a $\bot$ and $\top$ element is created, the $\bot$ element means the variable is not tainted and $\top$ means it is tainted. this is combined into a *State* lattice.

$$\begin{array}{ll} L_{taint} & = (\{\bot, \top\}, \sqsubseteq_{taint}) \\ \sqsubseteq_{taint} & = \{(\bot, \top), (\top, \top), (\bot, \bot)\} \\ State & = Vars \longrightarrow L_{taint} \end{array}$$

To learn the most as possible from the analysis a *State* will be stored for each node in the CFG, that way taint information can be extracted from any part of the program. Given a CFG (N, R, syn) the full abstract program state is defined as $PState = N \longrightarrow State$. With this a shorthand to get a node's state is denoted $n \in N \quad [\![n]\!]$ and is equivalent to $PState(n)$. With this the abstract semantics are defined as a function $t \in (N \times PState) \longrightarrow PState$ that takes in a CFG node and the node's current state returning a modified state. The t takes in a node and depending on the syntax of said node it differs on the semantics. All the rules are made to only change the *State* that corresponds to the current node. With this it is easier to disconnect each of the rules, and check whether they are monotonic and in turn if t is monotonic. However, despite not changing other node's state they do read their predecessors state using the $Join \in N \longrightarrow State$ function that gets the least upper bounded state of all predecessors for a given node.

$$Join(v) = \bigsqcup_{w \in pred(v)} [\![w]\!]$$

With this each rule of t can be defined, they all follow the same form first is the syntax that the rule can match. Then followed by the expression of a given rule that calculates the new state of the given node called $v \in N$. Many of these rules depend on an *eval* function that is explained at the end but intuition is that it evaluates whether an expression is tainted or not. The rules of t are as follows:

$$\begin{array}{ll} x = e : & [\![v]\!] = in[x \mapsto eval(in, e)] \\ \quad where & in = Join(v) \end{array}$$

$$\begin{array}{ll} return\ e : & [\![v]\!] = \bot\ [\tau\text{—}return \mapsto eval(Join(v), e)] \end{array}$$

$$x = \tau{-}return: \qquad [\![v]\!] = [\![v']\!][x \mapsto [\![w]\!](\tau{-}return)] \quad where \begin{cases} v', w \in pred(v) \\ syn(v') = f(e_1, ...e_n) \\ syn(w) = Exit \end{cases}$$

$$Entry: \qquad [\![v]\!] = \bot[p_1 \mapsto eval([\![v']\!], e_1)...p_n \mapsto eval([\![v']\!], e_n)]$$
$$where \begin{cases} v' \in pred(v), w \in succ(v) \\ syn(v') = f(e_1, ..., e_n) \\ syn(w) = int\ f(p_1, ...p_n) \end{cases}$$

$$\begin{array}{ll} int\ x[n] = \{e_1, ..., e_n\}: & [\![v]\!] = in[x \mapsto \bigsqcup_{i=1}^{n} eval(in, e_i)] \\ \quad where & in = Join(v) \end{array}$$

$$\begin{array}{ll} x[e_i] = e: & [\![v]\!] = in[x \mapsto in(x) \sqcup eval(in, e)] \\ \quad where & in = Join(v) \end{array}$$

Some of the available syntax options in the CFG are not covered by these rules, that is by design. As any rule that was not covered above are considered to be no-op nodes. The semantics for these nodes are to just to continue propagation.

$[\![v]\!] = Join(v)$

The eval function takes in an abstract state and an expression. Depending what type of expression we have different outcomes. Here we handle the cases: binary operations, $x \in Vars$, τ as the taint constant and $l \in Literal$.

$eval : (State, Expr) \longrightarrow L_{taint}$

$$\begin{array}{ll} eval(\sigma, e_1 \bigoplus e_2) & = eval(\sigma, e_1) \sqcup eval(\sigma, e_2) \\ eval(\sigma, x) & = \sigma(x) \\ eval(\sigma, \tau) & = \top \\ eval(\sigma, l) & = \bot \end{array}$$

2.2 Lattices

Because programming languages have constructs that make can make a program run forever. An important property of static program analysis is that it should terminate eventually such that even if the analysis takes a long time it must stop eventually. This is useful because it lets us learn about programs that will not necessarily terminate in a static environment.

The intuition behind proving that an analysis eventually terminates is that the abstract program state must be partially ordered, the ordering must have a finite height and the algorithm needs to be monotonic. That way each iteration of the algorithm either moves up the ordering of states or terminates. This makes a max bound for run time as the algorithm will eventually run out of states. With this idea we can prove the algorithm will eventually terminate, this algorithm will be called a least fixed point algorithm.

To properly describe the partially ordered state space lattices are be used. A lattice L is a tuple $L = (A, \sqsubseteq)$ where A is a set of elements in the lattice e.g. the state space, and a set of binary relations $\sqsubseteq \in \mathcal{P}(A \times A)$. As a shorthand to describe one of these relations we write $x, y \in A, x \sqsubseteq y$ which means $(x, y) \in \sqsubseteq$. The shorthand also applies over an element $x \in A$ and a set $A' \subseteq A$ such that $x \sqsubseteq A'$ denotes $\forall y \in A' : x \sqsubseteq y$. The binary relation should uphold these properties:

- **Reflexivity**
  $\forall x \in A : x \sqsubseteq x$
- **Transitivity**
  $\forall x, y, z \in A : x \sqsubseteq y \wedge y \sqsubseteq z \implies x \sqsubseteq z$
- **Anti symmetry**
  $\forall x, y \in A : x \sqsubseteq y \wedge y \sqsubseteq x \implies x = y$

When working with lattices the least upper bound operator $\sqcup$ is used to find the lowest ordered element that is also ordered higher than both operands. $x, y, z \in A$ a least upper bound operation has the following properties $x \sqcup y = z \implies \{x, y\} \subseteq z$. This operator is generalised to as a summation $\bigsqcup A'$ on $A' \subseteq A$ which returns element $z \in A$ such that $\bigsqcup A' \implies A' \sqsubseteq z$. The least upper bound is introduced because it offers a monotonic way to combine lattices, which is used to define the monotonic function in the abstract semantics.

Any lattice from this point forward is a complete lattice unless otherwise stated. A complete lattice is a lattice with a finite height and any subset must have some least upper bound $A' \subseteq A \implies \exists z \in A : A' \sqsubseteq z$. To ensure this holds all future lattices have two special elements $\top, \bot \in A$ these are respectively the greatest and lowest ordered elements in the lattice L, that is $A \sqsubseteq \top$ and $\bot \sqsubseteq A$ always holds. As the intuition eluded to the height of a lattice is important. The height of a lattice is the longest chain $C \subseteq \sqsubseteq$ that contains both the greatest and lowest elements $\top$ and $\bot$. The height is considered finite if this chain is not infinitely long that is $|C| \neq \infty$.

2.3 Control Flow Graph

A Control flow graph (CFG) is a directed graph which contains nodes and edges. Each node describes a part of a program and edges describes the control flow throughout a program. More formally a CFG is a tuple with a set of nodes N, a binary relation of directed edges $R \subseteq \{(u, v) \mid u, v \in N\}$ and a syntax function syn for a final tuple (N, R, syn). The addition of a syntax functions comes from [16]. An edge between two nodes $u, v \in N$ is represented with $(u, v) \in R$ saying v is the successor to u. For use in later sections two functions are defined $pred(v) = \{u \mid (u, v) \in R\}$ to get predecessors of v and $succ(u) = \{v \mid (u, v) \in R\}$ to get successors of u. The syntax function is used to get the syntax from some node such that $syn \in (N \longrightarrow S)$ where S is the set of possible statements. These statements are defined in the grammar of SC more specifically the $\langle stmt \rangle$ rule together with three special node types used to denote the entry, exit and function definition $S = \langle stmt \rangle \cup \langle funcDef \rangle \cup \{"Entry", "Exit"\}$. There exist exceptions to that rule: while, if and function definition, the grammars for these three constructs recursively contain $\langle stmt \rangle$ rules themselves. These inner statements are ignored in the resulting syntax and are instead converted to new nodes to represent the body of a if or while statement.

2.3.1 CFG Construction

The CFG is the result of parsing and for most cases one node is created for each $\langle stmt \rangle$. However, there is an edge case as seen in figure 2.2. When a function call statement $f_c \in N$ is reached the called function is inlined by copying the function body's CFG. The inlined function is surrounded by an Entry and Exit node respectively named $f_{entry}, f_{exit} \in N$. The entry node is always guaranteed to have exactly one predecessor $pred(f_{entry}) = \{f_c\}$ and exactly one successor $succ(f_{entry}) = \{f_{def}\}$ and $pred(f_{def}) = \{f_{entry}\}$, the node f_{def} will always be the function definition node such that $syn(f_{def}) \in \langle funcDef \rangle$. The successor predecessor structure of the Entry node f_{entry} is important for the abstract semantics.

The call site also has a special construction to handle returning values properly. After function inlining is done another node is added $f_r \in N$ which is the return assignment node. It assigns the returned value from the function to some variable. These nodes also have a specific structure regarding to their edges. The call node has the successors $succ(f_c) = \{f_r, f_{entry}\}$, the control flows splits here to propagate the program state from f_c to f_r. The return assignment node must have the following predecessors $pred(f_r) = \{f_c, f_{exit}\}$. It has the Exit node as a predecessor so it has access to the returned value.

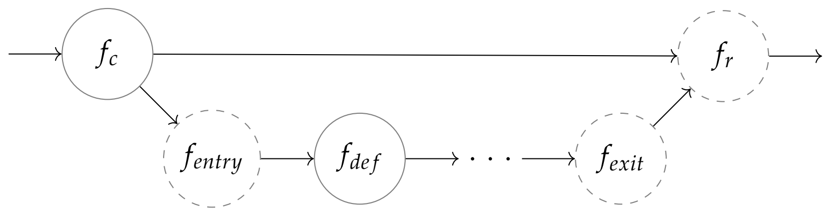

Figure 2.2: CFG example of a function call where dashed nodes are nodes created by special rules instead being of from some source code

2.4 CUDA

The CUDA-toolkit is used to write GPU programs which are to be executed on Nvidia GPUs. CUDA allows use of high-level languages to be used for developing GPU programs known as kernels [12]. A kernel is transferred from host to device where it is executed. A host is the CPU that establishes a connection with the GPU and allocates the resources needed for the kernel to be executed. The device is the GPU, it executes the kernel that have been transferred and informs the host once the kernel terminates. Transferring data to and from the device is handled by the host. The kernel is a highly parallel process, we reason about its concurrency and threads using different layers which can be seen on figure 2.3.

The threads executing in a kernel are structured in a grid of thread blocks. This grid can have up to three dimensions, the dimensions of the grid is a software abstraction over a one-dimensional array. Each thread block holds the same amount of allocated threads defined for when the kernel is launched. When a block is scheduled for execution, its threads are grouped into warps of 32 threads. Scheduling of these warps are done with use of streaming multiprocessors(SM). A single SM can schedule multiple warps and the warps can be from different thread blocks. These SMs are part of the Nvidia GPU architecture and is not controlled by the programmer. Threads within the same warp are executed in a lock-step fashion. This execution is called Single Instruction Multiple Threads(SIMT). Meaning that for each thread in the same warp, the same instruction is executed in parallel. The downside with this execution method is that if some threads in a warp are to branch their execution path from the rest of the threads, it will make threads inactive which are to execute different instructions. This means that branching limits the parallel execution of warps until threads can reconverge. This is only a downside for individual warps since execution of warps are independent of each other.

To fully utilize the parallel performance of GPUs it necessary to reduce the amount of inactive threads, which can occurs from divergence, threads exiting early and unused threads in warps. Since warps are always a size of 32 threads, it would mean that if the number of threads, in a thread block, is not a multiple of 32, the remaining threads assigned in a warp would be inactive. The way a thread block is grouped into warps is always the same. Assigning threads to a warp is done by having threads of incremental thread ids added to the warps until 32 threads have been added. This makes it possible to design code around these warps even though it is not possible to alter warps using CUDA.

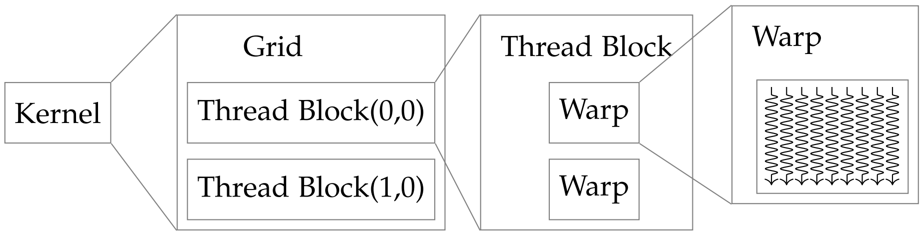

Figure 2.3: Structure of a GPU kernel

2.4.1 CUDA Memory Management

The most common performance bottleneck in CUDA-based applications is memory bandwidth [15]. Compared to the amount of instructions a GPU can execute per second, the memory bandwidth is very limited. Comparing the newest high-end consumer-grade CPU and GPU, which at the time of writing is the Intel Core i9-9900K and the NVIDIA RTX 3090 TI. The CPU is capable of about 922 GFLOPS (giga floating point operations per second) at base frequency and 1.28 TFLOPS at max turbo frequency [6, 17]. The maximum memory bandwidth of the CPU is 41.6 GB/s. In comparison the GPU is capable of a peak TFLOPS of 35.6 and a maximum memory bandwidth of 672 GB/s [13]. The ratio between memory bandwidth and operations per second is more than double on the GPU compared to the CPU. Here the bottleneck is the memory in case that more than one of the operations tries to access data which are not located in the cache. So making efficient use of the correct memory types and caching in CUDA is important.

Device memory is the main memory in the GPU device and is the staging area for all the data that gets copied from the CPU. As seen in figure 2.4, device memory is logically divided into several types of memory. These types of memory include local, global, constant and texture memory. Local memory resides in the global memory but is local because it is private to each thread. Local memory used when there is a register spill and the data spills to device memory. Global memory can be seen as the main part of device memory, this is the memory where data is copied from the host. Constant memory and texture memory are both global caches in the device memory. Constant memory is optimized for repeated access to the same values whereas the texture memory is optimized for spatial locality.

Concurrent accesses from a warp can be optimized as a single coalesced memory access if the memory is sequentially read, also known as coalesced memory access. This can reduce the amount of time needed to access the global memory because because caching can be used and it is possible to retrieve 32 values, one for each thread in a warp, in a single read.

Next up the memory hierarchy is shared memory. Shared memory is shared between threads in the same block, and is not accessible for threads in other blocks. Shared memory has lower latency and higher bandwidth than global memory. It can be advantageous to keep data that is reused many times, in the shared memory.

To increase the bandwidth of shared memory, it is divided into equally sized memory chunks called banks[5]. As an example the Volta architecture has the shared memory of each thread block split into 32 banks, where each bank is 4 bytes wide. This allows simultaneous access to memory for all threads that accesses different banks. A bank conflict a is where threads in the same block access memory in the same bank. When a bank conflict occurs, the access of the involved threads will be serialized. A way to minimize this is by padding the data so the threads will access different banks.

Compared to CPUs, GPUs have significantly more registers with up to 64,000 registers pr SM [12, Appendix K.1]. This makes context switching faster, as multiple contexts are stored within the registers so the core does not have to save or load any data to context switch. Local variables defined in the kernel will be stored in the registers. Each individual SM have a L1 cache, that caches data when accessing local and global memory[12]. The SMs of a device shares a constant cache along with a L2 cache. The constant cache stores data when accessing constant memory space. The L2 cache is used to cache accesses to local and global memory. If it is discovered during the compilation that there is not enough registers then the data will get pushed to the L2 cache or even the global memory.

Copying data between CPU and GPU can have a huge impact on the execution time of CUDA kernels [4]. Two primary issues when copying data is the amount of individual transfers for the kernel and how the data is stored on host. A way to reduce data copying is to group data and perform a single contiguous copy, reducing the total execution time of the kernel. The other issue is how the data is stored in memory on the host. The CUDA driver will not directly operate on page-able memory so additional overhead occurs in order for CUDA to perform a transfer for data in page-able memory[4]. CUDA performs a copy of the data into pinned memory and afterwards transfers the data to the GPU. This overhead reduces the speed of which data is transferred from CPU to GPU. In order to avoid this, the host have to allocate the data directly as pinned memory.

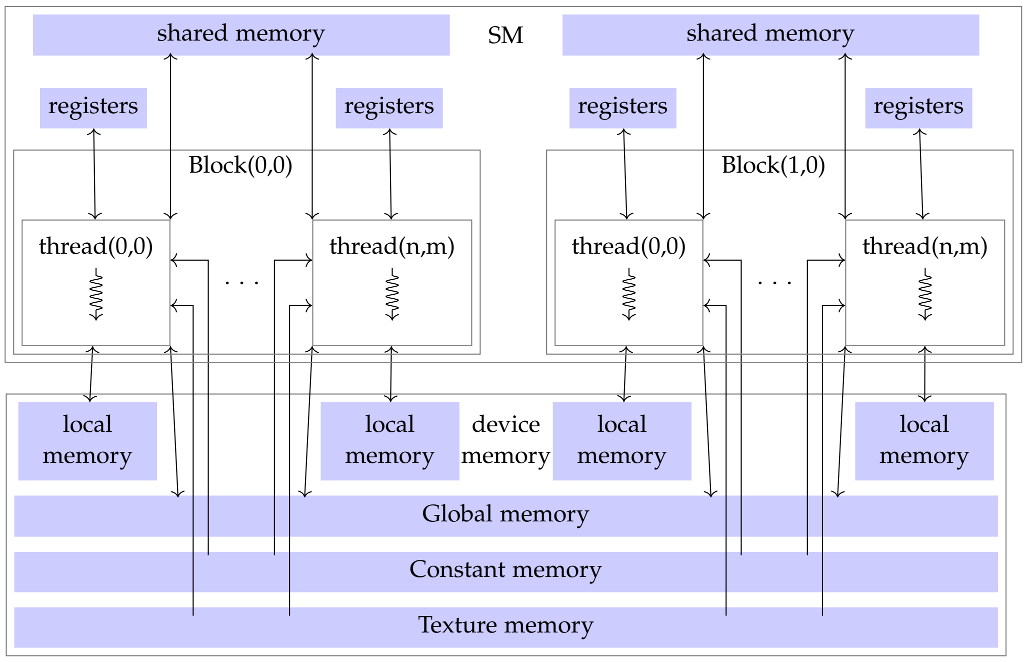

Figure 2.4: Memory access of threads

2.5 Related Work

It have been attempted to create a dynamic data race detection tool GUARD(GPU Accelerated Data Race Detector) which makes use of on-chip GPU[8]. It utilizes the GPU cores to process memory traces of CPU programs to locate data races. These memory traces are created from the CPU program which is being executed in parallel with GUARD. GUARD makes use of a Happened-Before algorithm to find data races in the memory traces. They show that there is 1.8% performance overhead and produce results with 18.8% false positive rate.

Another use of the GPU to accelerate analysis is an inclusion-based Points-to Analysis [9]. This paper shows that it is possible to implement irregular algorithms on GPU. They implement Andersen-style inclusion-based points-to analysis and receive a speedup in performance of 7 times compared to the CPU sequential implementation. They inform that porting CPU algorithms to GPU results in poor performance and it is therefore require to heavily modify the algorithm before it is possible to utilize the computation power of the GPU. They inform that this process requires a larger time investment to implement the algorithm, however the final code of the algorithm requires fewer lines compared to the CPU.

An attempt to off-load graph algorithms have been done with use of CUDA[3]. They create implementations in CUDA for three different algorithms and compare their performance with the CPU counterpart. These comparisons show a significant improvement in performance, however this improvement is lost if the graph being worked on have low edge to node ratio. These low degree graphs makes the GPU algorithms slower then the CPU as the linearity of the graph makes the parallel execution of the GPU ineffective. They show that it is possible to have GPU operate on graph data, which proves that operating on a CFG would also be feasible.

These work shows that attempts to utilize GPUs to accelerate already known CPU algorithm have been made and shows it is feasible to do. A major issue with different CPU algorithms is that they can become slow with large amounts of data to operate on. To the best of our knowledge no other works attempt to parallelize a taint analysis on a GPU using the CUDA toolkit.

2.6 Research Direction

After the additional optimizations that was made to CUBLAS did not show promising results compared to the CPU analysis, another approach is needed to be able to move the static program analysis to the GPU. Dropping the additional CUBLAS library and creating the analysis directly with CUDA could give more performance and control.

Chapter 3

Design

The previous taint analysis work on SC showed that using matrices and the CUDA library CuBLAS was not ideal for a GPU approach for static taint analysis. Therefore, a more direct approach can be taken by programming CUDA kernels which performs the analysis on the GPU. Two algorithms are designed the bit-cuda and cuda work-list, the design of these algorithms will also be described in this chapter.

3.1 Abstract semantics reduction

During parsing and CFG creation a naive approach is to take advantage of polymorphism as every statement has different data and semantics tied to them. As example the assignment "$x = e$" has a variable and an expression compared to a function call "$x = f(e_1, ..., e_n)$" which also has a function identifier and more expressions. Polymorphic CFG nodes solve this issue but this leads to branching code either through virtual functions or if statements. Which is problematic when writing CUDA kernels as it causes warps to diverge, reducing computational throughput.

To avoid this issue a non-polymorphic solution that can model the abstract semantics defined in section 2.1.3 is needed. The previous work already achieved something similar by reducing the abstract semantics into matrix multiplication problem. So there is a precedent that it is possible. A new reduction that changes it into a non-polymorphic bit-vector problem is proposed.

First the observation is made that every rule of the abstract semantics joins on the predecessor data and uses it on some level. Therefore the first step for any node v in the reduction is performing a join. The join operation is divergence-safe to use because the predecessor data is present on every CFG node.

$$in = Join(v)$$

From here most of the rules contain two patterns. The first rules evaluate some expression and assign the result to a variable. This rule models the data flow from an expression to variable, for example in an assignment. The second rule places some restrictions on what data can be propagated. This is used to model constructs such as *return*, where there should be a restriction on what can propagate back to the call-site.

To model these propagation restrictions each node is given a join mask such that a node v has a join mask $J_v \subseteq \mathit{Vars}$ that contains what variables cannot be propagated. The $j(v)$ function returns the joined state of the predecessors of v but where all the variables contained in the join mask J_v is reset.

$$j(v) = in[x \mapsto \bot \mid x \in J_v]$$

To evaluate expressions and assign them to some variable, transfers are introduced. A transfer is a tuple of some variable that will be assigned to and a set of variables which used in an expression $(\mathit{Vars}, \mathcal{P}(\mathit{Vars} \cup \tau))$. So for the example of an assignment "$x = y + \tau$" is modelled with the transfer $(x, \{y, \tau\})$. This tuple models the following semantic $\bot[x \mapsto eval(in, E)]$. This new *eval* is equivalent to the ones used in the abstract semantics. It takes in a set of variables and the taint constant that is present in an expression and calculates whether the expression is tainted:

$$eval(\sigma, E) = \begin{cases} \top & if\ \tau \in E \\ \bigsqcup_{v \in E}\ \sigma(v) & else \end{cases}$$

Because some node types has more than just one expression, a node is allowed to have 0 or more transfers and the set of transfers for some node v is defined as: $T_v \subseteq (\mathit{Vars}, \mathcal{P}(\mathit{Vars} \cup \tau))$. Then the transfers can be expressed as.

$$t(v) = \bigsqcup_{(x,E) \in T_v} \bot[x \mapsto eval(in, E)]$$

Combining these three steps into a final systematised semantic. For some node v with known join mask J_v and transfers T_v the full equation is.

$$[\![v]\!] = j(v) \sqcup t(v)$$

$$[\![v]\!] = in[x \mapsto \bot \mid x \in J_v] \sqcup \bigsqcup_{(x,E) \in T_v} \bot[x \mapsto eval(in, E)]$$
$$where\ in = Join(v)$$

This equation can be used to prove the applicability of this reduction to some abstract semantics. If a join mask set J_v and transfer set T_v can be found for every node the reduction can model a given analysis. When these sets are found the equation can likely be rewritten to the original form. An example of this is done with the assignment on the normal form "$x = e$". Because the value x gets overridden it must not be propagated, so the join mask is $J_v = \{x\}$. Because an assignment only has one expression, there is only one transfer. This transfer is $T_v = \{(x, E)\}$ where E is the set of variables and taint constant used in the expression e. The conversion is ignored for brevity. Giving us the abstract semantics, which can be simplified into the same rule that is in the abstract semantics in section 2.1.3

$$\begin{array}{ll}
x = e: & [\![v]\!] = in[x \mapsto \bot] \sqcup \bot[x \mapsto eval(in, E)] \\
\quad \mathit{where} & in = Join(v) \\
 & \qquad \Downarrow \\
x = e: & [\![v]\!] = in[x \mapsto eval(in, E)] \\
\quad \mathit{where} & in = Join(v)
\end{array}$$

3.2 CUDA Analysis Design

By writing kernels directly, the programmer has direct control of what data individual threads accesses and what instruction they execute. These aspects were handled by CuBLAS in the previous approach. The approach would attempt to assign one GPU thread to each CFG node in the program, compared to the matrix implementation which used many threads per node.

This approach uses bit-vectors, a data structure that uses each bit in some memory region to store boolean values. For this purpose, each variable was assigned an index to a bit and if the bit at that index is one then the corresponding variable is tainted. The necessary information about each node including the bit-vector is copied onto the global memory on the GPU. Then each iteration of this solution would start by reading and joining each predecessor's bit-vectors, as shown in the pseudo code in listing 3.1. A join-mask and transfer rules are used to describe the abstract semantics of the analysis. The join mask describes what bits are joined, and the transfer rules are used to describe data-flow from expression into a variable. By the end of the iteration the new bit-vector for each thread is stored on the global memory. Before starting the new iteration, a synchronization takes place to ensure all threads have finished writing to the global memory and check if the algorithm is done.

Using this approach, the algorithm uses as many threads as there are nodes. But since the number of possible threads are limited there will be program sizes where there are more nodes than threads. When this happens, CUDA will context-switch between the threads, which still works as the algorithm synchronizes on the CPU after having run all the nodes. The problem when there are more nodes than threads is that the parallelism of the GPU is no longer fully utilized. Because some

```
function main():
    while has_changed:
        has_changed = false
        analysis(nodes)

gpu_function analysis(nodes):
    node_index = threadIdx.x + blockDim.x * blockIdx.x
    node = nodes[node_index]

    last_bitvector = node.bitvector
    joined_bitvector = join(node, node.predecessors)
    node.bitvector |= joined_data & node.join_mask

    transfer_function(node, joined_bitvector)

    if last_bitvector != current_bitvector:
        has_changed = true;
```

Listing 3.1: Pseudo code describing the design of the analysis using CUDA.

nodes have already reached their local least fixed point, and GPU resources are wasted on those nodes. This could be improved if the algorithm only ran on the necessary nodes, since not all nodes will have any new information from joining. It could therefore be relevant to create and compare to a GPU algorithm that utilizes this.

3.3 CUDA work-list

Before developing a new Cuda implementation, different elaborations of the first implementation were considered. Each of these improvements at their core attempt to allow programs of any size to be analyzed, which was a clear disadvantage to the last implementation.

Early on the concept of a worklist-based algorithm stood out. It promises to reduce redundant work and provide an intuitive solution to scheduling large programs over a limited number of threads. However, this comes at the cost of synchronization, the parallel algorithm would require a central work-list data-structure that is read and written to by potentially thousands of threads. This creates a risk for a bottleneck, but with the correct implementation this might be outweighed by the performance benefits of a work-list algorithm.

The different approaches can be categorized into two distinct approaches, **grouping** and **non-grouping** based algorithms. **Grouping** based approaches starts by grouping CFG nodes into groups that are sized so they can be executed by a single

GPU thread block with one thread per CFG node. Making the smallest scheduling unit a group of CFG nodes. Whereas a **non-grouping** approach will schedule individual nodes on the work-list.

For both approaches the specifics of the work-list data structure is important to minimize time spent scheduling new work. The naive solution would experience incredible slowdowns if the entire data structure had to be locked with every write. Therefore, changes were made to the work-list data structure for both approaches. The benefit of the **grouping** approach is that there are fewer threads that have to synchronize around the same resource. This could make a naive stack-based work-list algorithm possible. But it would still require locking the stack upon popping and pushing work. Another option is to add a boolean for each group that marks each group for whether it needs to be worked on. The only race condition for this solution is when two thread blocks want to take the group off the work-list at the same time. As multiple threads can mark a block, at the same time, without any possible race conditions. The **non-grouping** splits the entire work into work columns, each can contain as many items as the GPU can handle each iteration. We divide the work-list into a number of these work columns, such that each iteration of the algorithm a new work column is handled and new work discovered fills up the next work column. A work column is just an array of nodes that should be handled. Because we handle the entire work column at a time we have no need for contiguous node elements. This allows us to divide the lock on the work column into smaller locks, such that a thread only locks a single element in a work column at a time. The thread will pick an index in the work column in a pseudo random fashion by using a hashing function. This was done because the items must be placed on the work columns in a timely manner. Because this is a distributed algorithm multiple threads can add items at the same time, which results in slowdowns in naive methods like adding front to back due to synchronizations. Therefore, a hashing algorithm is used in an attempt to lower the time spent synchronizing. Collisions in the work column could be handled either by finding a new index or delaying the work to the next work column. This random placement means work columns will not be filled completely leaving some threads without work, but it saves a lot of time on waiting for locks instead. A major downside to this approach is that it requires the entire work column to be finished before it can move onto the next work column which might cause slowdowns, as a lot of threads can end up waiting on a single thread finishing its work. The two approaches were tested and the results showed that using multiple work columns upon collision were faster than attempting to completely fill each work column.

The pseudo code for adding entries to the work-list is shown in listing 3.2. The function iterates through a node's successors and adds them to the work-list. First the hash for the successor is calculated and then a while loop is used to change the work column until an empty position is found. Finally the successor is added to

```
function add_successors_to_worklist(node, worklist,
   column_index, worklists_pending):
    foreach successor in node.successors:
        work_column = worklist[column_index]
        new_worklists = 1
        hash = hash(successor) % worklist_size
        while work_column[hash] != -1:
            if work_column[hash] == successor:
                break

            work_column = worklist[column_index +
               new_worklists]
            new_worklists++

        work_column[hash] = successor
        worklists_pending = max(worklists_pending,
           new_worklists)
```

Listing 3.2: Pseudo code describing adding a node's successors to the work-list

the available position.

The actual analysis function, shown in listing 3.3 is similar to the previous in listing 3.1. The difference is that now all the nodes are arranged in work columns of a work-list. Instead of having a bool stating whether there is still work to be done, a counter is used instead. The counter states how many work columns still contain nodes to be analyzed.

3.4 CUDA generalisation

To extend the solution and make it more widely applicable, the Cuda algorithm will be generalized to allow similar analyses to run with the same approach. The generalization will be designed with the abstract semantics of a multi colored taint analysis in mind see section 3.4.2. Which help guide what aspects of the analysis should be generalised.

3.4.1 Design goals

The generalisation is going to be based on the cuda worklist algorithm, so an initial goal with the generalisation is to hide any worklist related details. As the worklist algorithm works unrelated to the specifics of the analyses that will be run using the generalisation.

Looking at the differences between single-colored taint and multi-colored taint

```
function main():
    while worklists_pending > 0
        worklists_pending--
        analysis(worklist, column_index, worklists_pending)
        column_index = (column_index+1) \% column_count

gpu_function analysis(worklist, column_index,
   worklists_pending):
    node_index = threadIdx.x + blockDim.x * blockIdx.x
    node = worklist[column_index][node_index]

    last_bitvector = node.bitvector
    joined_bitvector = join(node, node.predecessors)
    node.bitvector |= joined_data & node.join_mask

    transfer_function(node, joined_bitvector)

    if last_bitvector != current_bitvector:
        add_sucessors_to_worklist(node, worklist,
           column_index, worklists_pending)
```

Listing 3.3: Pseudo code describing the design of the work-list analysis using CUDA.

semantics. The state lattice stands out as the most important thing to modify in the semantics. The semantics rarely has to be modified due to the reliance on the least upper bound operator. So in theory with an interchangeable state lattice and least upper bound operator would suffice to create a useful generalization. But the reliance on a least upper bound operator proves problematic in practice as the least upper bound operation may be a slow for some data structures.

Instead of defining a least upper bound operation, the generalisation should require definitions of a separate join and transfer functions. This way each function can take advantage of potential optimizations available for a given state lattice.

If the state of a CFG node has changed, a signal has to be sent to the work-list indicating that the successors should be scheduled on the work-list. This is the only work-list detail that can not be completely hidden from the generalisation.

Because the starting point was a taint analysis, the base abstract semantic is designed to model data flow analysis and over approximate its results. Additionally, support for handling multiple functions is already built into the semantics making the generalisation useful for interprocedural analyses. The final result should be a generalisation that can help implement forward and backwards flowing, over-approximated and interprocedural static program analyses for SC. The generalisation is built on top of a bit-vector analysis and thus naturally the generalisation favors bit-vector implementations.

3.4.2 Multi-colored taint

Because the previous semantics were generalized using the least upper bound operator, defining a similar data-flow analysis is simple. By changing the abstract program state lattice and updating the *eval* function appropriately. Given a set T of all taint sources in a input program, the L_{taint} lattice can be modified to keep track of which taint sources have tainted a given variable.
$L_{m-taint} = (\mathcal{P}(T), \sqsubseteq)$
$\sqsubseteq = \{(x, y) \mid x, y \in \mathcal{P}(T) \land x \subseteq y\}$
This defines the multi taint lattice, to ensure completeness, $\bot$ and $\top$ are identified and the height of the lattice is $|T|$, and T is guaranteed to be finite in a finite input program. The $\emptyset \in \mathcal{P}(T)$ is the $\bot$ element in this lattice and it holds that $\forall T' \in \mathcal{P} : \emptyset \subseteq T'$. Similarly $T \in \mathcal{P}(T)$ is the $\top$ element as it holds that $\forall T' \in \mathcal{P}(T) : T' \subseteq T$. Using this new state lattice the node state can be defined as $State = Vars \rightarrow L_{m-taint}$. Then the *eval* function is modified for the taint source case.
$eval(\sigma, \tau) = \{\tau\}$ where $\tau \in T$ Having defined the differences between a multi-colored and single-colored taint analysis. These changes can now be expressed using the generalization.

Implementation

Using the generalisation described, implementing a new analysis is made very simple. Previous parsing and CFG transformations can be reused. The primary change in the new analysis is the state lattice changes. Which is changed from keeping track of what variables are tainted to what variables are tainted by each taint source. This change is implemented by dynamically sizing up the bit-vector by the amount of taint sources in the program.

Changing the underlying data structure, requires new join and transfer functions. These functions are effectively the same except they do the same behaviour for each taint source. If even one bit changes in the state lattice during the join and transfer functions the signal is sent to the work-list to schedule successor nodes to the work-list.

These are the only two changes needed to get the analysis running on the GPU. Additional code is required to use the results from the analysis but this will be considered auxillary code and not a part of the analysis code.

To implement the multi-colored taint analysis an analyze method is created that controls how the specific analysis is performed. This method can be seen in listing 3.4. The analyze method iterates through each taint source and for each it performs join and transfer on the nodes. A new join function is defined and is used to handle the new node structure with taint sources but the transfer method used in the other analyses is reused. In this analysis another data structure is used for

```
__device__ bool analyze(Node& current_node, NodeContainer&
    nodes, Transfer* transfers){
        bool add_successors = false;
        for(int source = 0; source < source_count; ++source){
            BitVector joined_data =
                multi_cuda_join(current_node.predecessor_index,
                nodes, source);

            BitVector last = current_node.data[source];
            BitVector current = last;

            joined_data.bitfield |= current.bitfield;

            if(joined_data.bitfield == 0)
                continue;

            transfer_function(current_node.first_transfer_index,
                transfers, joined_data, current);

            current.bitfield |= joined_data.bitfield &
                current_node.join_mask.bitfield;
            if(last.bitfield != current.bitfield){
                current_node.data[source] = current;
                add_successors = true;
            }
        }
        return add_successors;
    }
```

Listing 3.4: Analyze method responsible for performing the multi-colored taint analysis

the nodes, this data structure is identical to the previously used node data structure except that instead of having a single bit-vector as its data, it has an flexible array member of bit-vectors. This way it can store the bit-vector for each taint source in the set T.

With the semantics of the analysis defined in an Analyzer class the final configuration can be setup before the analysis is run. Both setup and execution can be seen in listing 3.5. The analyzer is instantiated here to allow meta data and additional parameters to be passed along to the GPU, in this case the source count is saved. As mentioned previously the size of abstract state type's size is determined at run time which complicates the process of uploading the data to the GPU. To solve this issue NodeUploader was used, it is a class that describes how the generalization should upload the data and its meta data to the GPU. In this case a SizedArray is used which is a array can store items with size determined at run

```
void multi_cuda::execute_analysis(DynamicArray<Node>& nodes,
    std::vector<Transfer>& transfers, const std::set<int>&
    taint_sources, int source_count){
    multi::Analyzer analyzer(source_count);

    worklist::NodeUploader<SizedArray<Node>> uploader;
    uploader.container.item_size = nodes.get_item_size();
    uploader.dev_nodes = (void**)&uploader.container.items;

    worklist::execute_some_analysis(analyzer, nodes, uploader,
        &transfers[0], transfers.size(), taint_sources);
}
```

Listing 3.5: Code example of how multi-colored taint analysis is configured and run

time. The NodeUploader is configured to push the data and size each item in data. Finally the generalization is called with the Analyzer class instance and the NodeUploader, additional parameters that remain unchanged are also passed along like an array of transfer objects and the set of taint sources.

Chapter 4

Optimizations

To compare the GPU implementations to the CPU implementation benchmarks are necessary. The first benchmark is performed on a program that has a lot of taint sources and as such should be better fit for the GPU implementations. The code can be seen in appendix B.1. This code example was analysed on the CPU, bit-cuda and bit-cuda worklist implementations. The code has been scaled from 27 to 270,000 nodes to see how it scales on the different implementations. Each test run was performed 10 times and the mean time was used to get a stable measurement as the times can vary. Parsing is not measured in the benchmarks since this is the same for all implementations.

Looking at this benchmark results in figure 4.1, it is clear that the GPU implementations perform worse than the CPU implementation. To determine where the time is used additional timings are measured on the bit-cuda worklist implementation to see the distribution of the time spent in the different components of the analysis. The result of the timings can be seen in figure 4.2. From this test it shows that only 3% of the time is spent on the actual analysis algorithm. The majority of the time is spent on creating the CFG nodes and transforming those nodes with information necessary for the GPU analysis. Specifically this time is used to create the bit-vector field, create transfer functions and set up the successors and predecessors. Both the creation of the CFG nodes and the variable reduction happens for both the GPU and CPU analysis. Based on this it shows that there is a potential for the GPU analysis to be useful if the overhead connected with the transformation can be reduced.

4.1 CUDA overhead

The benchmarks from both the Bit-cuda and cuda work-list algorithms, resulted in unusual runtimes. The resulting times were incredibly long and as such the natural step was to adjust different parameters. But changing these parameters

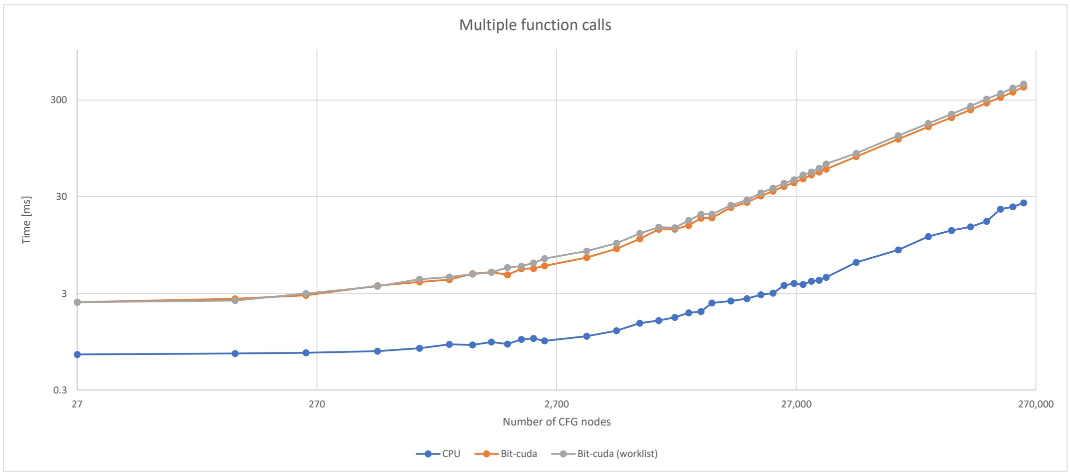

Figure 4.1: Benchmark result from analysing code from appendix B.1 with a scaling number of function calls

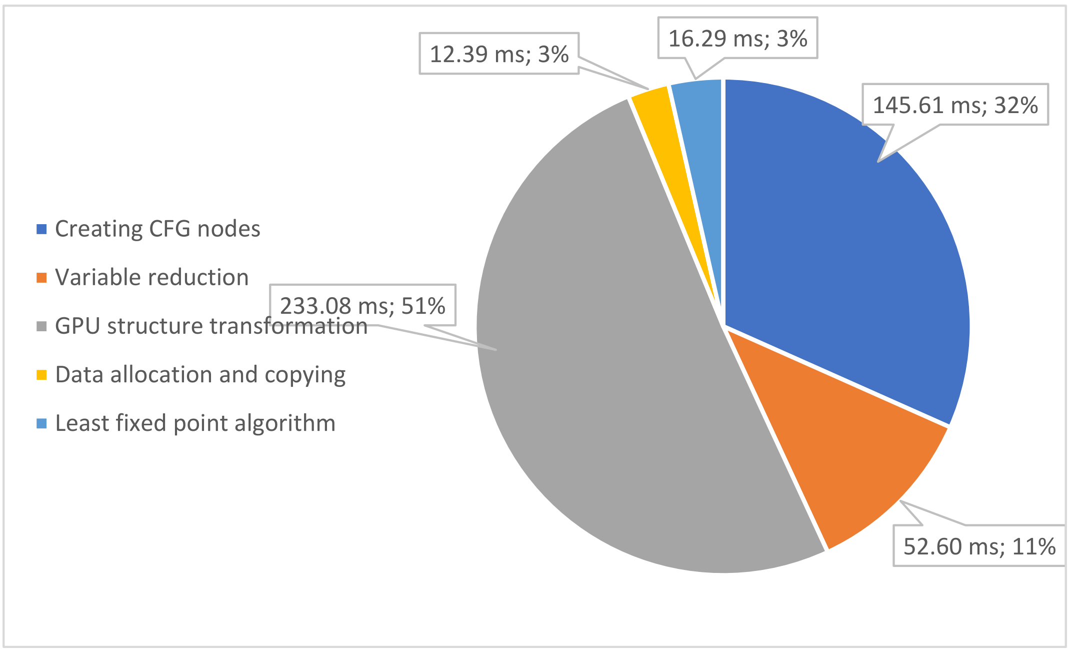

Figure 4.2: Run-time distribution of bit-cuda worklist on code from appendix B.1 with 10.000 function calls

only improved the algorithms an insignificant amount.

From there the work-list algorithm was profiled and the runtime of each call in the algorithm was measured. From this profiling an interesting result was discovered. The first CUDA call had an disproportionate runtime compared to similar calls later in the algorithm. It turned out that the first call to the graphics card induces a large overhead of around 0.1 seconds. Which in the small to medium sized programs, that are being tested on, takes up the vast majority of the runtime. This time comes from the creation of the CUDA context [12, Ch. 3.2.1]. This overhead will not be measured in benchmarks or comparisons between the implementations but the effects of the overhead on the analysis is discussed in section 7.2.

4.2 Associative Containers

To reduce the overhead of the GPU implementations, the run-time distribution as seen in figure 4.2 was used to prioritize what to optimize. To properly optimize the CPU bound overhead, a profiling tool called gprof was used to find the bottleneck. Initial results showed that up to 40% of the time was spent doing look-ups in associative containers, that is containers containing key value pairs.

These associative containers were generally used to add information to some object. An example would be variable reduction, which uses an associative container to assign indexes to variables. This was a pattern in many places of the program, associative containers used to decorate objects with additional information that was learnt throughout the program.

However with this pattern, look up times are too expensive. The default associative container uses a sorted binary tree to store its elements and thus incurring a logarithmic look up time [1]. This look-up is further slowed down by the fact that strings were used as keys, causing $O(log_2(N))$ string comparisons for each look up.

The first attempt to improve the performance of the slow look-ups, was done by changing the underlying data structure from a sorted binary tree to a hash map, which has a constant look up time [2]. However this did not give the performance improvements needed.

The current approach to improve performance is to get rid of the pattern of using associative containers to decorate existing objects with extra information. Instead adding more data fields to the CFG node data structures and populating the fields as we learn the information, this drastically reduces the amount of look ups needed.

After reducing the use of associative containers the run time is better in terms of the GPU structure transformation. The new run time distribution can be seen on figure 4.3. This optimization resulted in a speedup of about 128ms which is an improvement of 27.7% measured at 10,000 function calls or 240,000 CFG nodes.

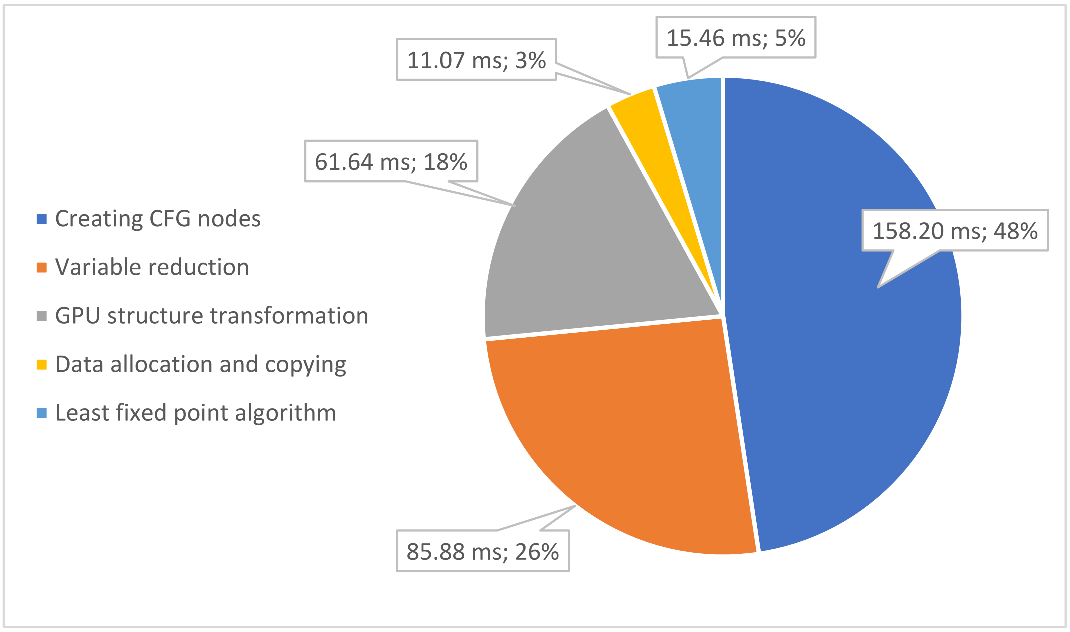

Figure 4.3: Run-time distribution of bit-cuda worklist on code from appendix B.1 with 10.000 function calls after reducing the use of associative containers.

4.3 Parser Optimization

From the run time distribution in figure 4.3, the three largest tasks are: create CFG nodes, variable reduction and GPU structure transformation. These tasks are run sequentially before the analysis is performed, causing significant overhead. These three tasks collectively take up 92% of the run time of a GPU analysis on a large sample program. The CPU overhead wastes time and resources and thus an attempt to optimize the overhead was made.

This optimization comes in three pieces, a change to the data structure, doing tasks concurrently and improving function in-lining. Before the optimization, an intermediate CFG data structure was used, this data structure was removed such that the tasks instead are performed directly on the GPU data structure. This had two benefits, it removed the need for the *GPU structure transformation* task. But it also got rid of the cache unfriendly nature of the intermediate data structure. Because the intermediate nodes were stored in a non-contiguous way on the heap using smart pointers, cache misses were much more likely. Instead with the new data structure it is stored contiguously, improving data locality significantly. Another optimization was to stop doing the tasks sequentially but instead do them concurrently, handling the *Variable Reduction* as each node is created. This further reduced the potential for cache misses during the *Variable Reduction* task as the target node is already in the cache. The final improvement was to the *Creating*

CFG nodes specifically function in-lining, which used to be a slow process as a tree like structure of heap allocated objects had to traversed and copied for each function call. Instead, the data was structured in a way to allow the copying to be done with a *memcpy* operation instead. A direct memory copy is much faster than copying separated data hidden behind layers of indirection. After the copy is done the copied nodes are iterated through and an offset is applied to successor and predecessor indexes. To ensure they match their new node indexes.

These optimizations completely replaces all CPU overhead in a GPU analysis which made up a total of 92% of the run time. With these optimizations the runtime of the analysis is reduced by 91%, when analyzing the program in appendix B with 10,000 function calls.

4.4 Splitting Data From Node Struct

Looking again at the time spent in the different analysis components, a large amount of time is spent on copying the analysis results back to the CPU. When running on the benchmark program with 240,000 CFG nodes, this time was 2.1 milliseconds. The data that is being copied here is the node struct that contains the predecessor, successor, transfer and state. But only the state data will have changed during the analysis. To reduce this time, the state data is separated from the rest of the node data. This optimization resulted in a small improvement of 1.8 milliseconds of the analysis. The component-wise comparison of this can be seen in figure 4.4.

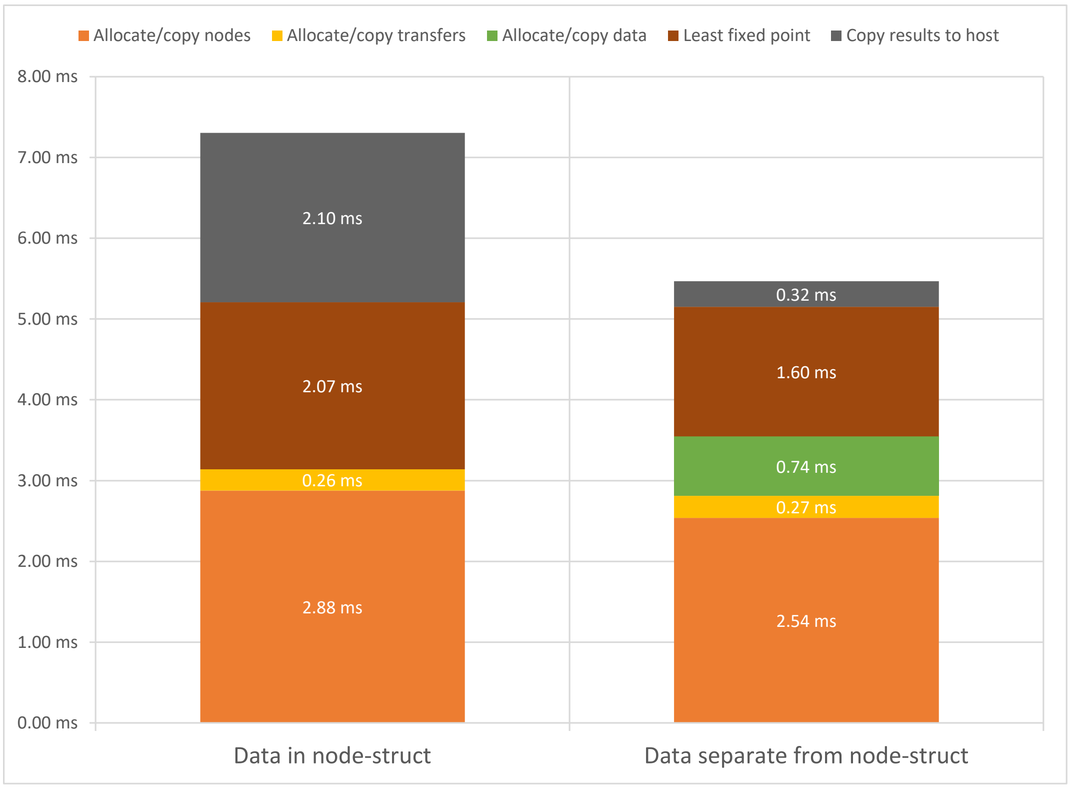

Figure 4.4: Component-wise comparison of bit-cuda implementation with and without separation of data. To simplify comparison, parsing is not included

Chapter 5

Implementation

This chapter will cover implementation details about how the CFG is constructed and the data structures for nodes and transfers which are used to perform the taint analysis. Implementation of the two different taint analysis algorithms, bit-cuda and cuda work-list, will also be covered.

5.1 Overview

When a SC program is analyzed, a series of transformations is applied to the source program before the least fixed point algorithm can be executed. The flow of performing an analysis can be seen on figure 5.1. When analyzing a SC program, it is necessary to transform the program into a CFG. The taint analysis is interprocedural, and this is done by duplicating the function nodes to each call site, effectively inlining every function call. Further implementation details of this can be found in the previous work[16].

During construction of CFG nodes, variable reduction is performed. This reduces the memory footprint of the analysis by taking advantage of the local variable scopes of functions. The reduction encodes variable ids into bit-vector indexes that stores the taint status of all variables at the node. The advantage with local variable scopes is that indexes of the bit-vector can be reuse for every function.

The transfer functions are created and referenced from the CFG node that is being constructed. A transfer function encodes how each CFG node affects the abstract program state.

While CFG nodes are created, taint sources are located in the source program. The id for nodes which contains a taint source is added to a vector which is used to provide the starting nodes of the analysis. This is done as an optimization primarily meant for the GPU implementation, as the GPU implementation can handle propagation from every taint source in parallel. This does however also provide a time save for the CPU counterpart as finding taint sources during the

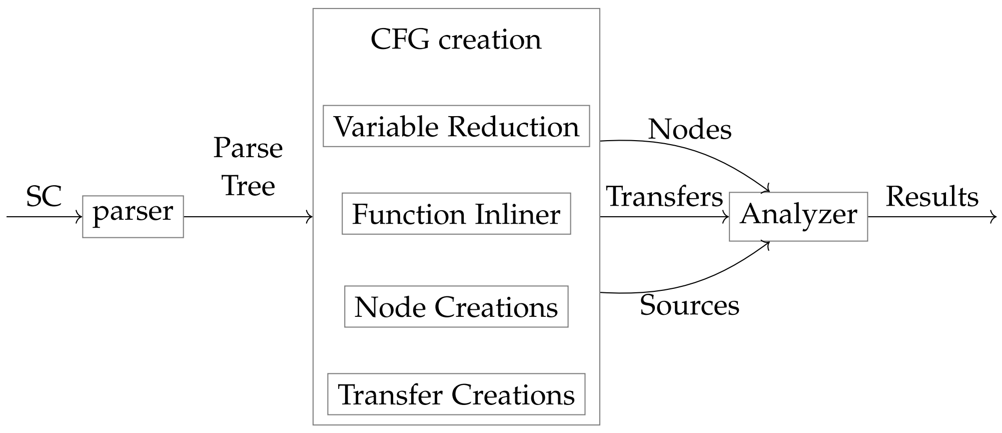

Figure 5.1: Program flow to perform taint analysis

analysis is not performed.

Once the source program have been converted into a CFG it can then be given to the analysis algorithm to propagate taints in the program. There is a slight deviation between a single taint and multi-colored taint analysis in how the taint information is stored. For the single taint analysis, only a single bit-vector describing taint values for variables is used, while in multi-colored taint there is a bit-vector for each taint source in the program.

For analysing the program, a work-list approach have been used for CPU and GPU since it is a common way to handle fix-point algorithms. Another analysis approach have been implemented on GPU which is thread-based, where each thread evaluates a single CFG node. When the least fix-point algorithm finishes, the result of the analysis is obtained and can be extracted from the CFG.

5.2 CFG Nodes

The SC language has a lot of different constructs that have different semantics, which naturally leads to branching execution in the CUDA kernel. This is an issue since GPU programs achieve better throughput if a warp of threads executes non-branching code. In order to reduce branching, a transformation is used to generalize nodes into a single type of data. An example of this generalization of CFG nodes can be seen in listing 5.1. This node struct is used to store CFG node information and is used for computing taint analysis on the GPU and CPU.

```
struct Node{
    int first_transfer_index = -1;
    int predecessor_index[5] = {-1,-1,-1,-1,-1};
    int successor_index[5] = {-1,-1,-1,-1,-1};
    BitVector join_mask = INT64_MAX;
};
```

Listing 5.1: CFG node data structure used for single taint analysis

To access the correct nodes when performing the joins, a *predecessor_index* array is used, similarly a *successor_index* array is used to reference the correct nodes which should be added to the work-list during the analysis. The *first_transfer_index* is used to access the transfer information of the statement at the CFG node. Transfer information is the information on how data flows from one variable to another in a CFG node, e.g. an assignment node has information flowing from the right hand side expression into the left hand side variable. This transfer information is stored in transfer objects, each transfer can store one flow from an expression into some variable. These transfers contains the index in the bit-vector of the variable which is assigned to in the CFG node. The transfers also contains a bit-vector describing which variables are present in the expression. In the case of a function call with multiple parameters, a single transfer struct cannot bind all the parameters. In this case transfers are linked together to perform multiple data flows in one node, this is done by making transfers have an index to the next transfer for the given CFG node. Since data in a variable is overridden when performing transfers for assignments and initializations, a *join_mask* is used to describe which variables should propagate from previous CFG nodes. However this does not affect the result of the evaluation of the expression as they are evaluated on the joined data of the predecessors. This means that if the assigned variable is also used in the expression, evaluation will be performed on the previous state of that variable.

For storing taint state of each CFG node, a separate vector of bit-vectors is used. These bit-vectors shares the same index as the CFG node which state the bit-vector is representing. Having the data as a separate data structure makes it faster to extract the final analysis results from the GPU.

These CFG nodes are created from the parse tree created by Antlr, a visitor pattern is used to access the nodes in the parse tree and create the corresponding CFG node for it. This process involve multiple modifications in order to correctly encode the program flow and mapping variables into bit-vectors. To perform variable reduction, the method *get_var_index* is used every time a variable name occurs in the program. The method can be seen in listing 5.2. This method returns the id for the bit in the bit-vector that represents the given variable. It makes use of a map *var_to_index* which stores the already seen variables in the given function scope.

```
int get_var_index(std::string name){
    auto it = var_to_index.find(name);
    if (it == var_to_index.end()) {
        var_to_index.emplace(std::move(name), next_var_index);
        return next_var_index++;
    }
    else {
        return it->second;
    }
}
```

Listing 5.2: Variable reduction during creation of CFG nodes

To create the transfer functions used in the analysis, expressions needs to be encoded into bit-vectors which informs which variables are present in the expression and allowing its data to flow into the variable on the left-hand side of the statement. Converting expressions is handled in the visitor shown in listing 5.3. It starts out creating the default bit-vector where no variables are present and as variables are discovered this bit-vector has the corresponding bit flipped to tell that the variable is present in the expression. If more than one expression is present, these following expressions are handled recursively, and the results are combined with a bitwise or operation.

```
virtual antlrcpp::Any
    visitExpression(scParser::ExpressionContext *ctx) override
{
     BitVector expression(0);
     if (ctx->ID() != nullptr){
         expression.set_bit(get_var_index(ctx->ID()->getText()));
     }else if(ctx->expression() != nullptr) {
         antlrcpp::Any result = ctx->expression()->accept(this);
         expression = result.as<BitVector>();
     }

     if (ctx->expressionM() != nullptr &&
        ctx->expressionM()->expression() != nullptr){

         antlrcpp::Any result =
            ctx->expressionM()->accept(this);
         expression |= result.as<BitVector>();
     }

     return expression;
}
```

Listing 5.3: Implementation for handling expressions in SC

An example of a CFG node creation can be seen in listing 5.4. First off, the expression of the assignment statement is visited which creates the bit-vector describing which data influence the left-hand side variable. Next is the CFG node created with use of *add_node* which code can be seen in listing 5.5. With the node created, it is necessary to alter the join mask such that the variable itself will override previous data stored in the variable. This is done by flipping the bit representing the variable in the *join_mask* of the CFG node. Lastly the transfer for the CFG node is created. This transfer contains the id of the CFG node, the variable which data from the expression is transferred to and the bit-vector of the expression itself.

```
virtual antlrcpp::Any
    visitStatementassign(scParser::StatementassignContext *ctx)
    override
{
     antlrcpp::Any result =
         ctx->expression().back()->accept(this);
     BitVector rhs_expression = result.as<BitVector>();

     Node& assignment = add_node();
     int var_index = get_var_index(ctx->ID()->getText());
     assignment.join_mask ^= 1 << var_index;
     add_transfer(assignment, var_index, rhs_expression);
     next_layer();

     return nullptr;
}
```

Listing 5.4: Contruction of a assignment statement CFG node

When adding new nodes to the CFG it is necessary to connect it to its predecessors. These predecessor ids are stored in the vector *last* and *next_last*. When a node is added to the CFG, that node is inserted into *next_last* such that it can be added as the predecessor of the following node that will be added. During the process of adding the predecessors, these predecessors will also add the new node to their successors. In the listing 5.4 the method *next_layer* is called. This method is used to tell that the current level of the CFG is finished being constructed and will move all the current node ids of *next_last* into *last*. This method also sets *succ_pred_index* to 0 since the next layer does not have any successor nodes yet.

```
Node& add_node(){
    int index = nodes->size();
    Node& node = nodes->emplace_back_resizable();

    for(int pred_index = 0; pred_index < last.size();
       ++pred_index){
        node.predecessor_index[pred_index] = last[pred_index];
        get_nodes()[last[pred_index]]
            .successor_index[succ_pred_index] = index;
    }
    ++succ_pred_index;

    next_last.push_back(nodes->size() - 1);

    return node;
}
```

Listing 5.5: Implementation for adding new CFG nodes to the CFG.

5.2.1 Function inline

Functions are created as separate CFG sub-graphs which are stored in a separate data structure for inlining functions on call site. This process creates entry and exit CFG nodes and insert the functions body in between these two nodes. Implementation for inlining function can be seen in listing 5.6. Inlining function is done by iterating through the function and recreating each CFG node in the function and applying an offset to all the ids stored in the predecessor and successor indexes. To preserve scoping of functions, the entry and return CFG nodes have their join mask modified to not allow variables to propagate through them.

5.3 Bit-Cuda Implementation

The code that launches the bit-cuda kernel is shown in listing 5.7. This code is run after all the data has been allocated and copied to the GPU. Here a boolean value `has_changed` is used to remember if any changes happened to the nodes during the iteration. If that is the case, then another iteration will be executed until a fixed-point is achieved.

The GPU kernel code for the bit-cuda implementation is shown in listing 5.8. First, the id of the current thread is calculated and is used to get that thread's node and bit-vector. Since each thread is assigned a node. Then for each node, a join (section 5.4.1) is performed with the node's predecessors and the transfer function (section 5.4.2) is applied on the node's data. Finally, the old bit-vector is compared

```
Node& clone_function(const Function<Node>& function){
    int offset = nodes->size();
    nodes->reserve(function.nodes.size());
    for(int i = 0; i < function.nodes.size(); ++i){
        const Node& node = function.nodes[i];
        Node& new_node = nodes->emplace_back_resizable(node);
        for(int i = 0; i < 5; ++i){
            if(new_node.successor_index[i] == -1)
                break;
            new_node.successor_index[i] += offset;
        }

        for(int i = 0; i < 5; ++i){
            if(new_node.predecessor_index[i] == -1)
                break;
            new_node.predecessor_index[i] += offset;
        }
    }
    for (int source : function.sources) {
        taint_sources->push_back(source + offset);
    }
    return (*nodes)[offset];
}
```

Listing 5.6: Implementation for function inlining

```
bool has_changed = true;
while(has_changed){
    has_changed = false;
    cuda_copy_to_device(dev_has_changed, &has_changed, sizeof(bool));

    // Launch a kernel on the GPU with one thread for each element.
    analyze<<<block_count, threadsPerBlock>>>(dev_nodes, dev_data, dev_transfers, dev_has_changed, nodes.size());

    // cudaDeviceSynchronize waits for the kernel to finish, and returns
    // any errors encountered during the launch.
    cudaStatus = cudaDeviceSynchronize();

    cuda_copy_to_host((void*)&has_changed, dev_has_changed, sizeof(bool));
}
```

Listing 5.7: CPU Code for launching kernel for bit-cuda. For better readability the error checking and handling after launching the kernel

to the new. If there has been a change then the `has_changed` boolean is set to true, so the host knows that another iteration is needed. To avoid race conditions, the boolean is reset to false by host such that the kernel only ever set it to true. This way, the boolean either remains false or, one or more threads sets it to true.

5.4 CUDA Work-list Algorithm

To utilize the most threads on the GPU, the work-columns should be filled as much as possible. This can however be a costly process since multiple threads attempt to insert into the work-list at the same time. This can lead to multiple failed attempts for inserting a node, leading to a lot of processing time before the next iteration. Therefore the focus was on having faster insertions which leads to less filled work-lists in between iterations. The size of the work-column is equal to the amount of threads which is assigned to the kernel that executes the algorithm. The thread id is then used to index the work-list to find the node which the thread should process. Since the work-list is a fixed size and a node can have multiple successors it is necessary to introduce multiple work-lists which the work for the next iteration is inserted into. When a work-list index is finished processing, the index is set to a default value which indicates that there is no work for the index. By resetting each index to the default value, it is possible to reuse previous iteration work-lists

```
__global__ void analyze(Node nodes[], BitVector data[],
    Transfer transfers[], bool* has_changed, int node_count) {
    int thread_id = threadIdx.x + blockDim.x * blockIdx.x;

    if(thread_id < node_count){
        Node& current_node = nodes[thread_id];

        BitVector current = data[thread_id];
        BitVector last = data[thread_id];

        BitVector joined_data =
            join(current_node.predecessor_index, nodes, data);
        current.bitfield |= joined_data.bitfield &
            current_node.join_mask.bitfield;

        transfer_function(current_node.first_transfer_index,
            transfers, joined_data, current);

        if(last.bitfield != current.bitfield){
            data[thread_id] = current;
            *has_changed = true;
        }
    }
}
```

Listing 5.8: Bit-cuda Kernel code

```
while(worklists_pending > 0){
    --worklists_pending;
    cuda_copy_to_device(dev_worklists_pending,
       &worklists_pending, sizeof(int));

    analyze<<<block_count, threadsPerBlock>>>(analyzer,
       dev_nodes, dev_data,
       (int(*)[THREAD_COUNT])dev_worklists, work_column_count,
       dev_transfers, node_count, dev_worklists_pending,
       current_worklist);

    cuda_copy_to_host((void*)&worklists_pending,
       dev_worklists_pending, sizeof(int));
    current_worklist = (current_worklist+1) %
       work_column_count;
}
```

Listing 5.9: CPU Code for launching kernel with the correct worklist. Execution stops when no worklist are left to be processed

to store new nodes to process.

Before entering the least fix-point loop shown in listing 5.9, some preparation work is required. This entails initializing the work-list with the taint sources of the program that is analysed and allocated GPU memory for the workload of the analysis. Once initialization and allocation is complete, data is then transferred to the GPU from CPU memory. The majority of the data is kept on GPU in between each iteration since the CPU part of the algorithm only requires knowledge of how much work is left. This is done by a shared variable *worklists_pending* which is a counter for how many work-lists contains nodes to compute, which also means how many iterations are left. This counter is updated while the GPU is computing an iteration. The kernel itself requires multiple parameters when launched. Note that the prefix "**dev_**" is short for device and indicate that the variables are pointers to GPU global memory. The data of these are shared for all the threads in the kernel.

- **analyzer**: Is a class which contains the logic from the abstract semantics, the analysis takes advantage of template instantiation to change implementation depending on which analysis is being executed.
- **dev_nodes**: A pointer to CFG nodes stored on the GPU. Each node contains arrays for predecessor and successor indexes. A index into transfers which expresses the data flow of the CFG node. The join mask which is used to perform joins from predecessor nodes.

- **dev_data**: A pointer to the data which stores taint information for CFG nodes. Indexes of the node corresponds to the same index in this data array.
- **dev_worklists**: A pointer to the work-lists used throughout the analysis for scheduling new nodes to compute.
- **work_column_count**: Is a integer value for the amount of work-lists that have been allocated. This is to ensure that indexing into work-lists are kept in bounds and also used wrap around the work-lists as the analysis progresses.
- **dev_transfers**: A pointer to transfers. A transfer describes how data flows for a given CFG node. Each transfer contains a integer value for a bit vector index that describes the variable that data flows into. To describe the data the flows into the variable a bit vector is used. This bit vector indicate which variables are present in the expression. Lastly is a index to the next transfer which is relevant to the CFG node, this is to describe multiple variable assignments. This covers the specific data flow for each variable in a function call.
- **node_count**: Is a integer value similar to **work_column_count** which is used to ensure that indexing into nodes is done within bounds.
- **dev_worklists_pending**: Is a integer value describing how many work-lists in **dev_worklists** contains nodes to compute.
- **current_worklist**: Is the index for the work-list which is being computed at the given iteration.

Implementation of the GPU kernel can be seen in listing 5.10. In CUDA work is usually assigned to threads according to their thread id which is a number calculated at the start of a kernel. This thread id is stored into *node_index* and used to locate the data which the thread should use in the current iteration. Each thread is assigned to a single index in the current work-list of the iteration. However not all indexes of the work-list contains an id for a node meaning that those threads are inactive for the current iteration. The node id is used to index into *nodes* to find the correct CFG node and provide it to the analyzer which does the taint propagation for that node. Once taint has been propagated, the node id is then cleared from the work-list and the result from the propagation is used to check if the node successors should be inserted into the next work-list.

Inserting of new node ids into the work-list is done by *add_successors_to_worklist* which code can be seen in listing 5.11. Each node contains an array of successor node ids, this array is used to insert the correct ids for the next work-list. Since there are multiple threads executing this add successors code, there is potential

```
int node_index = threadIdx.x + blockDim.x * blockIdx.x;
int* work_column = work_columns[current_work_column];

if(node_index < THREAD_COUNT && work_column[node_index] != -1){
    NodeType& current_node = nodes[work_column[node_index]];
    bool add_successors = analyzer.analyze(current_node, data,
        work_column[node_index], nodes, transfers);

    work_column[node_index] = -1;

    if(add_successors){
        int next_work_column = (current_work_column+1) %
            work_column_count;
        add_successors_to_worklist(
            current_node.successor_index, work_columns,
            work_column_count, next_work_column,
            worklists_pending);
    }
}
```

Listing 5.10: Implementation for GPU worklist kernel

for race conditions on shared data. In this part of the code there are two variables stored in global memory which are accessible for all the thread in the kernel. This being the work-list data structure *work_columns* and *worklists_pending*. To avoid the risk of data races, atomic operations are used when writing to these variables. For inserting new node ids into the work-list, *atomicCAS* is used. This replaces the value in the work-list with the node id if there is not a node id already on that location. It may occur that indexes of the work-list already stores another node id, additional attempts to insert using incremented indexes until *COLLISIONS_BEFORE_SWITCH* is reached. This will reset the collision count and attempt to insert into the next work-list followed by the one which were failed to insert into. The amount of additional work-list required to store the successor nodes is stored in the global variable *worklists_pending* which is used to inform the amount of work-lists that contains node ids. The least fix-point algorithm terminates once this variable reaches zero.

5.4.1 Join Function

The join function is responsible for for retrieving the combined state of the current node's predecessors. This information will later be used to update the current node's state with information about what variables have been tainted in one or more of its predecessors. The GPU join function is shown in listing 5.12. The

```
int current_work_column;
for(int i = 0; i < 5; i++){
    int amount_of_new_worklists = 1;
    current_work_column = initial_work_column;
    int succ_index = successors[i];
    if (succ_index == -1)
        return;
    unsigned long hash = succ_index*120811;
    int collision_count = 0;
    int* work_column = work_columns[current_work_column];

    while(atomicCAS(&work_column[hash % THREAD_COUNT], -1,
        succ_index) != -1){
        if(work_column[hash % THREAD_COUNT] == succ_index){
            break;
        }

        if(++collision_count >= COLLISIONS_BEFORE_SWITCH){
            current_work_column = (current_work_column + 1) %
                work_column_count;
            work_column = work_columns[current_work_column];
            amount_of_new_worklists++;
            collision_count = 0;
        }else{
            hash++;
        }
    }

    atomicMax(worklists_pending, amount_of_new_worklists);
}
```

Listing 5.11: Implementation for adding successor nodes to worklists on GPU

```
template<typename NodeType>
__device__ BitVector join(int predecessors[], NodeType
   nodes[], BitVector data[]) {
        BitVector joined_data;
        joined_data.bitfield = 1;

        int pred_index = 0;
        while (pred_index < 5 && predecessors[pred_index] !=
            -1){
            joined_data.bitfield |=
                data[predecessors[pred_index]].bitfield;
            ++pred_index;
        }
        return joined_data;
}
```

Listing 5.12: GPU join function

join function will iterate through all the node's predecessors and do a bitwise `or` operation on the current node's and the predecessor's state. The resulting joined data is then returned and used to update the node's state, after applying a join mask. The join mask will prevent some tainted values from the predecessors from being tainted in the node's state. It is used to prevent the variable being assigned to, from being tainted from its predecessors or to reset the state after returning from a function.

5.4.2 Transfer Function

The transfer function is responsible for updating a node's state based on its abstract semantics. Such as an assignment where taint flows from the expression to the assigned variable. The GPU transfer function is shown in listing 5.13. A while loop iterates through all the node's transfer functions. Each iteration checks if any variables in the right-hand side of the assignment or initialization are tainted. This check is performed using a bitwise `and` operation between the node's joined data and the transfer function `rhs` bit-vector field. If any of the `rhs` variables are tainted then the variable corresponding to the transfer function's `var_index`, will be tainted in the node's state. Finally the transfer function's field `next_transfer_index` is used to find the next transfer function and is used in the next iteration.

```
template<typename BitVectorType>
__device__ void transfer_function(int first_transfer_index, Transfer transfers[], BitVectorType& joined_data, BitVectorType& current){
    Transfer* transfer;
    int transfer_index = first_transfer_index;

    while(transfer_index != -1){
        transfer = &transfers[transfer_index];
        if((joined_data.bitfield & transfer->rhs.bitfield) != 0){
            current.bitfield |= (1 << transfer->var_index);
        }
        transfer_index = transfer->next_transfer_index;
    }
}
```

Listing 5.13: GPU transfer function

Chapter 6

Evaluation

This chapter will evaluate on the benchmarks performed on the final version of the taint analysis implemented on CPU and GPU. Since the results of the analysis is an important aspect for knowing that the implementation is correct, a method for validating the results have been used throughout development of the GPU algorithms. This method of validation results will be evaluated on as well. Since CFG nodes have limits to their number of successors and predecessors, it limits the programs which can be modelled with these CFG nodes. Therefore it is needed to evaluate how much of an impact this limitation have. Evaluation will also be done on data structures used for the analysis to provide a estimate of how large programs would be able to be analysed with use of the GPU algorithms.

6.1 Benchmarks

To evaluate the analysis implementations, they are run on a few different programs. The first program to benchmark is a program that is designed to work well on a parallel analysis (appendix B.1). The program is beneficial to analyze in a parallel manner, since it has many taint sources. Analyzing it in parallel allows all the taint sources to propagate in parallel as well. This means the GPU will be utilized better since there will be more work for the GPU and less idle threads. This program is run multiple times on different variation with an increasing amount of CFG nodes. The results from running this analysis is shown in figure 6.1. The results shows that after 96,000 CFG nodes, the bit-cuda implementation is faster than the CPU implementation. The bit-cuda implementation is about 24% faster than the CPU implementation at analysing programs with 960,000 to 2,400,000 CFG nodes.

The second benchmark performed is a worst-case scenario for a parallel analysis. Here the program analyzed only has a single taint source that is passed through the entire program with an increasing amount of function calls that the taint propagates through. This program is not optimal for parallel analysis since

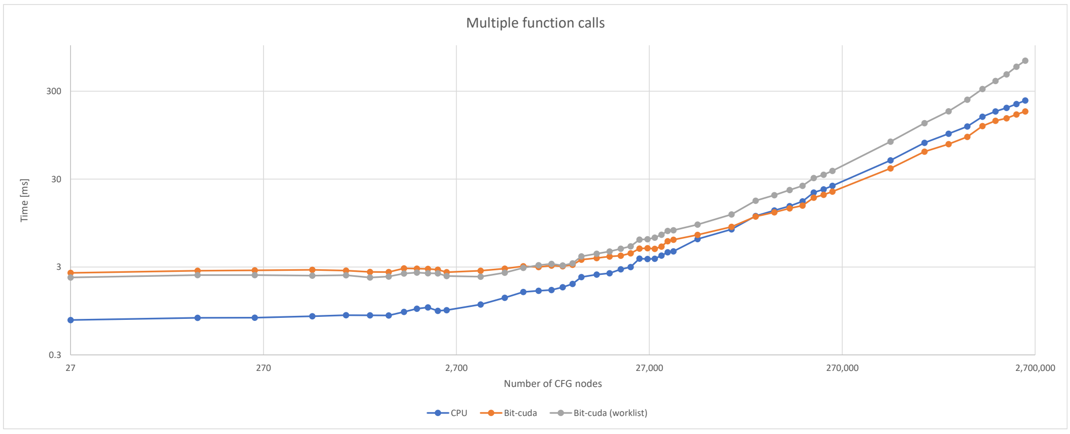

Figure 6.1: Benchmark results from analysing code from appendix B.1 with an increasing number of function calls.

the majority of the threads will be idle as only one taint source is being propagated every iteration. This means that it is not possible to properly utilize the GPU's parallel lock-step mechanism, since only a few threads will have work to do each iteration. The program is provided in appendix B.2 and the running time results of this analysis is shown in figure 6.2. In these results it is clear that the GPU implementations perform significantly worse than the CPU implementation when analysing this program.

Finally, a benchmark was performed on three different sample programs. The results can be seen in figure 6.3. All the sample programs are small programs with the largest having 452 CFG nodes. The results show that the CPU implementation is faster on all samples. At its best there is a 139% difference between the CPU and cuda work-list analysis.

Since there is major overhead in performing analysis on GPU it becomes difficult to tell how well the actual least fix point algorithm on GPU performs. Performing benchmarks only on the least fix point algorithm captures the difference in performance between GPU and CPU which can be seen in figure 6.4. It shows that bit-cuda's least fix point algorithm performs significantly better compared to cuda work-list and the CPU work-list algorithms on larger programs.

Based on these benchmarks, the results indicate that a GPU analysis implementation can be a viable option for a static program analysis in certain use-cases and program sizes. A static program analysis using a GPU can perform significantly worse than its CPU counterpart in certain situations and even in typical use case it is not matching the CPU implementation's performance. But the results shows that there are some specialized situations where a GPU implemented static program analysis can outperform a traditional CPU implementation.

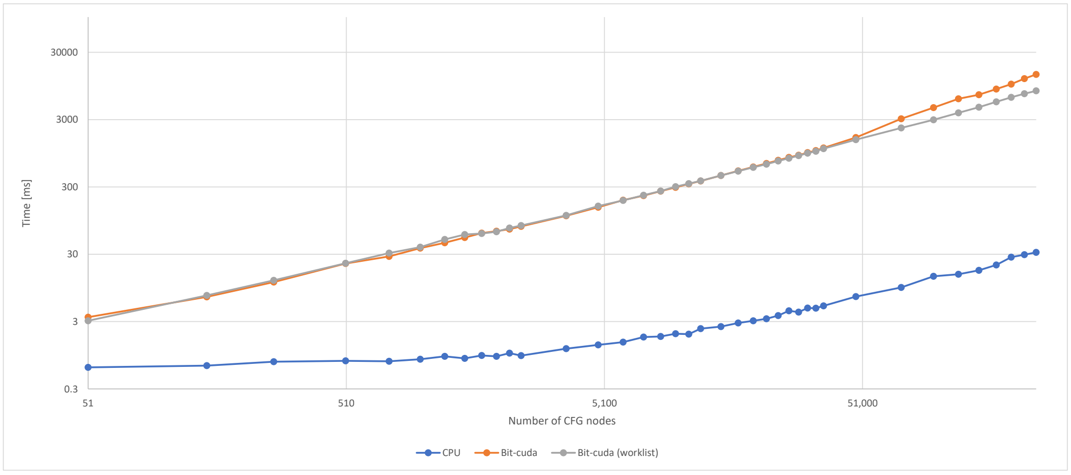

Figure 6.2: Benchmark results from analysing code from appendix B.2 with an increasing number of function calls.

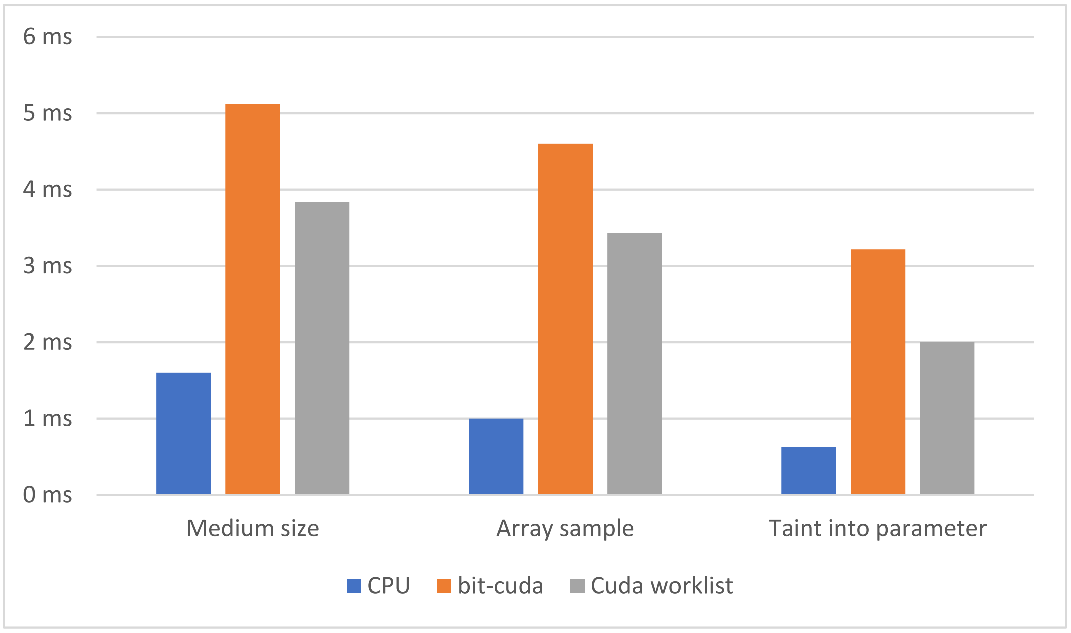

Figure 6.3: Benchmark results from analysing three different sample programs. The sample programs can be seen in B.3, B.4 and B.5 respectively

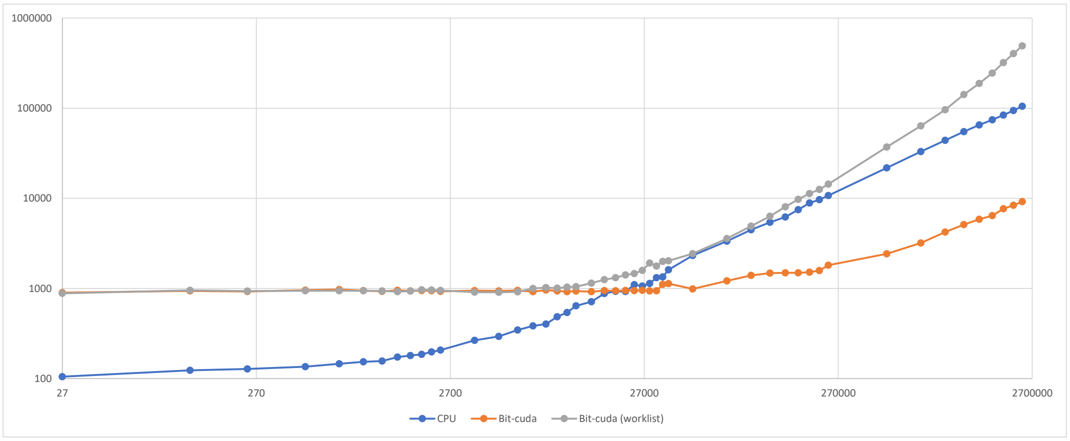

Figure 6.4: Benchmark of the least fix point algorithms. Performed on the sample program seen in appendix B.1

6.2 Validating Analysis Results

During the development of the different analysis implementations, it was necessary to attempt to ensure correctness of the analysis results, as the implementations were changed and optimized. To do this, all the analysis results from the different implementations were compared to each other. The check was performed simply by iterating through all the bit-vectors from each implementation and checking if they are identical. In case some of the bit-vectors differed from each other it was clear that some part of the analysis was giving the wrong results. This check was performed only on sample programs included in the benchmarks, so there is no guarantee that the implementations were checked with all constructs and their combinations. To complement this semi-automatic testing, it was necessary to manually verify at least one of the analysis implementations. This was done on smaller sample programs as larger programs would be infeasible to manually verify. If one of the analysis implementations gives the correct result, the semi-automatic verification can then be used to verify that the same results hold for the other implementations.

Using this type of verification, it could have been possible to keep a continuous chain of correct analysis results to check against with each change in any of the analysis implementations. But to be able to do this it would have been necessary to systematically test the analysis against a fixed set of sample programs each time a change was introduced to the implementations. This method of verification was only discussed at the very end of the development process and was thus not used. If the development is continued on the analysis implementations, this verification method would provide extra safety to confirm that the analysis results are correct.

6.3 CFG successor limitation

In the current implementation a simplification was used to minimize the amount of data structures that had to be maintained. This simplification works on the assumption that CFG nodes in the analysis will not have more than five successors. Due to this simplification some programs can not be represented by the CFG data structure specifically when a program results in at least one node with more than five successors. This only happens in the case of nested if statements or while loops. The simplification was used because C++ only allows one unbounded array to be in a struct. Thus the current implementation required some limitation to get the current node structure as seen in listing 5.1. If not for this limitation the successor and predecessor data has to be extracted into their own data structures similarly to the transfer data.

As mentioned earlier the assumption that five successors is enough is a faulty assumption in general use cases. However, for a controlled testing environment like the custom written sample programs used in this work, the assumption can hold, as they can be designed to avoid having a lot of nested statements. This simplification probably did alter results slightly as handling the more successors would likely add a slight overhead to the existing algorithms. But this simplification was deemed acceptable because it affects both the CPU and the GPU implementations as both would be affected by the added complexity of scaling successor counts. Interestingly there is a potential for such a change actually performing about the same or even better as overall memory usage and thus copying overhead can potentially be reduced, because most nodes only have one successor. However, the outcome of such a change cannot be shown without proper testing.

Because the performance changes if any from the simplification affects both implementations and this work focuses on the performance not correctness it is evaluated that the simplification does not negatively affect the findings of this work.

6.4 Space usage

As the input program grows in size so does the space requirements of the analysis. If the space requirements the exceed the local available memory on the GPU it has to page memory onto the host's main memory, which would cause too many slowdowns in a time critical operation. Therefore there is an effective memory limit on the analysis tool. The GPU used throughout this work is limited to 8GB of memory.

The algorithm makes use of three data structures, an array of nodes, an array of transfer objects and an array of bit-vectors. The node data structure (see listing 5.1) takes up 52 bytes of memory, and the transfer data structure takes up 16 bytes of memory. The amount of nodes is directly dependent on the size of the input program denoted N. Making the space usage of the nodes array $N * 52$ bytes. The transfer data structure is a bit more complicated, as the size of the array depends on the code. A lot of nodes are considered no-op nodes which do not have an associated transfer struct. But some nodes like function calls can have multiple transfer structs. To get around this, the node to transfers ratio T is calculated, but T is different for each sample program. With all the sample programs the range 4%-63% was calculated. There is some bias in this data as these sample programs were first and foremost written to test the algorithms for correctness. As an example the 63% sample program specifically tests function calls with many transfer objects making it unnaturally high. Therefore a generalization is made and the calculations will be made in ranges from best to worst case such that $T \in [4\%; 63\%]$.

This concludes in a formula $52 \times N + 16 \times N \times T$ to calculate the space usage. Solving this for N against the memory capacity of a GPU results in how big a input program can be on that GPU. For the 8GB GPU used in this work input programs can have 1.6×10^8 to 1.3×10^8 nodes depending on T.

6.4.1 Stress Test

A stress test was performed to check if it is possible to reach the previously discussed memory limit. This test also helps to check if the GPU implementations will break or give wrong results when given very large programs. Stress testing the bit-cuda analysis on the program in appendix B.1 shows that the maximum number of nodes that the analyzer can handle is around 4.6×10^7. Which is an order of magnitude lower than the expected. Indicating that the GPU memory is not the max node count bottleneck. But this restriction comes from the CFG creation when allocating space for the CFG nodes, which means that the same limitation also applies to the CPU analysis implementation. Before reaching this limit the bit-cuda analysis worked as expected and gave the same results as the CPU analysis.

Chapter 7

Discussion

This chapter serves as a discussion on whether things were done fairly and correctly, and what is needed to improve them. First, the fairness of using the CPU implementation as a comparison is discussed. Then the GPU start-time, discovered during the design of the bit-cuda implementation is discussed and analyzed. Finally, there is a discussion on the generalisation of the analysis about what benefits it provides and why it was not utilized.

7.1 CPU implementation as comparison baseline

To evaluate the performance of the GPU implementations a baseline was needed. The quality of this evaluation is dependent on the baseline. A good baseline implementation should have comparable performance to readily available alternatives. Because a custom language is being used, there exists no alternatives. Even comparing to analyses targeting other languages can be problematic as the language constructs could affect the analysis. Some features which exist in C, which SC is based on, like pointers could affect performance significantly depending on how the pointers are modelled. This makes a direct comparison difficult if not impossible. The baseline is implemented according to a work-list approach in [10]. But none of the mentioned optimizations were applied such as taking advantage of CFG node clustering or using a priority queue with SC's constructions to reduce work. The only optimization applied to the work-list was setting the initial work to be the set of nodes that contain taint sources, reducing the amount of unnecessary work. But not implementing these does not imply an unfair baseline implementation as these optimizations may also be applicable to a GPU work-list algorithm.

The CPU baseline is likely not performant enough to compete with industry standard taint analyzers. However, it shares a lot with the GPU implementations. They use the same data structures and in the case of the GPU work-list even the same algorithmic structure. This makes it easier to compare the CPU against GPU implementations as any optimization made to one usually has optimized the other. Because these implementations are so similar a significant performance deviation from the baseline would likely indicate that the difference is from using the GPU over CPU.

7.2 GPU Start-up

The first call to the Cuda library will trigger the creation of the Cuda context, this operation takes about 100ms [12, Ch. 3.2.1]. This overhead is constant and as such does not scale up with program size. Which means the overhead will have a limited effect on large programs. However it certainly limits the usage of Cuda algorithms on small to medium programs where the overhead is many times larger than a similar CPU algorithm's runtime.

The overhead is absent in direct comparisons during benchmarks. This is done to put more focus on the performance of the algorithms. Especially when comparing scalability of the algorithms, the overhead would overshadow the runtime changes as the program scales up. This overhead can in many real world applications be hidden, as the overhead can handled in parallel with other CPU tasks. For static program code analysis an obvious direction is to handle the Cuda startup during earlier parts of the compilation process like parsing. For the parsing time to match the time of the startup, the program should contain at least 720,000 CFG nodes. This means that the benchmark times are not accurate when the size of the programs is less than the 720,000 CFG nodes, since additional overhead could be present from the CUDA start-up time.

7.3 Generalisation

In section 3.4, an approach to generalise the analysis was introduced. This generalisation provides the ability to easily implement other over-approximated, interprocedural data flow analyzers. The only other analyzer that was designed and implemented was a multi-colored taint analysis. But during optimizations to the CPU and bit-cuda taint analysis, the structure of the implementations changed and due to time restrictions the multi-colored taint analysis was never re-implemented in the restructured project. Given more time, different analyzers could have been tested, some of which might have been better suited for massive parallelization and thus provided the GPU implementation with an advantage over the CPU implementation.

7.4 Semantic differences in implementation and Design

After formalising the implementation strategy in section 3.1 a problem became apparent. The abstract semantics in section 2.1.3 are not the same that are implemented. This can be proved by contradiction with the help of the formalised reduction.

Lets assume the abstract semantics of the implementation and design are the same. Given some assign return node v with $syn(v) = ''x = \tau\text{—}return''$. The reduction says there are two sets the join mask and transfer sets. For assign return these are defined as $J_v = \{x, \tau\text{—}return\}$ and $T_v = \{(x, \{\tau\text{—}return\})\}$ resulting in the full and simplified semantic rule:

$$[\![v]\!] = in[x \mapsto in(\tau\text{—}return), \tau\text{—}return \mapsto \bot]$$

$$where\ in = Join(v)$$

For comparison the same rule from the abstract semantics defined in section 2.1.3.

$$x = \tau\text{—}return : \qquad [\![v]\!] = [\![v']\!][x \mapsto [\![w]\!](\tau\text{—}return)] \quad where \begin{cases} v', w \in pred(v) \\ syn(v') = f(e_1, ...e_n) \\ syn(w) = Exit \end{cases}$$

Here some differences are immediately spotted. Primarily the abstract semantics takes data directly from each predecessor where relevant and the reduction takes data from both predecessors both times. This change in behaviour constitutes a contradiction. Thus the reduction fails to implement the abstract semantics completely. This issue was undiscovered until the reduction was formalised. However it is worth pointing out that the implementation actually simulates the same behaviour as the abstract semantics by maintaining some invariants. These ensure that $[\![v']\!]$ and $[\![w]\!]$ contain data in such a way that it does not matter if they are joined before each usage.

Thus the reduction fails to express the same abstract semantics but manage to take advantage of some invariants of the abstract semantics to simulate the exact same behaviour.

7.5 Minimizing bit-cuda synchronization on the CPU

Both GPU algorithms require a synchronization on the host to terminate the analyses. This synchronization will in some cases cause a lot of overhead. The synchronization for the bit-cuda analysis, can be seen in listing 5.7. Here, the kernel is launched in a while loop. After each iteration there is a check to see if any changes to the state occurred. This synchronization is necessary to detect if a least fixed point has been reached and then to be able to stop the algorithm.

An optimization that was tested was doing multiple iterations between each synchronization. The intuition is that if an analysis requires thousands of iterations to complete and the expensive part is checking for a least fix point then reducing checks could save some time. This optimization is at the simplest implemented on the bit-cuda so testing has been done with that implementation. The optimization was tested with the best and worst case sample programs, to see its effects on each one. On smaller programs and specifically the worst case sample program saw a large improvement in performance by a factor of ten. But the best case saw next to no improvement in the small cases, but as the programs scaled up both implementations started seeing performance drawbacks. It turned out the additional iterations increased the amount of redundant calculations significantly. Overall this optimizations can only be worth it in small programs where the amount of nodes is too small to reach full utilization of the GPU. So there is a potential for a dynamic optimizations that can be applied at run-time when a small program is inputted. Which could combat some of the algorithms worse use cases. But this optimizations will not be implemented as finding the right activation's parameters would be time consuming for a relatively small improvement.

Chapter 8

Conclusion

A new approach for reducing the abstract semantics of a static taint analysis was proposed. Similarly to the previous work this reduction contains GPU optimizations like removing divergence but unlike the previous work it uses bit-vectors instead of matrices. This reduction was put in use over two GPU implementations and also used to optimize the CPU baseline implementation. Both the bit-cuda and cuda work-list algorithms were implemented using the CUDA toolkit, and can be run on any NVIDIA graphics card. These algorithms only differ in how they delegate work, the bit-cuda algorithm assigns one thread for each CFG node every iteration ignoring redundant work. Where as the cuda work-list only does the required work but at the cost of maintaining a work-list. These two implementations were benchmarked against the CPU baseline implementation. The results show that the GPU analysis calculates the least fixed point algorithm significantly faster. However, an overhead occurs on the GPU analysis indirectly as the CUDA toolkit initialises. The initialization time proved to be a challenge as it took a constant time on the analysis which made it even more difficult for smaller analyses to compete with the run-time of the CPU analysis. Although attempts where made to hide the start-up time by running it in parallel with the parsing, it requires large programs to reach this goal. Another common source of overhead is allocating memory and copying data to and from the GPU. Despite all the overhead the bit-cuda algorithm manages to beat the CPU baseline on large non-worst case programs. The cuda work-list algorithm failed to outperform the overhead from the work-list with the current configuration. The speed up granted from using the bit-cuda algorithm alone is not worth applying GPU-acceleration to taint analysis for. However, because the GPU can run parallel to the CPU potential exists for parts of the compiler to be run on the GPU in parallel with the CPU. Additionally a generalization approach and implementation was proposed that allows a new analysis to be implemented with minimal effort and change. The generalization successfully hides the complicated CUDA constructs from the programmer.

To the best of our knowledge the taint analysis is implemented correctly. But the development lacked a proper thorough validation process. The only validation checked that the implementations give the same results but do not check these results for correctness.

Bibliography

[1] cppreference. *std::map - cppreference.com*. https://en.cppreference.com/w/cpp/container/map.

[2] cppreference. *std::unordered_map - cppreference.com*. https://en.cppreference.com/w/cpp/container/unordered_map.

[3] Pawan Harish and P. J. Narayanan. "Accelerating Large Graph Algorithms on the GPU Using CUDA". In: *High Performance Computing – HiPC 2007*. Ed. by Srinivas Aluru et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 197–208. ISBN: 978-3-540-77220-0.

[4] Mark Harris. *How to Optimize Data Transfers in CUDA C/C++*. https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/.

[5] Mark Harris. *Using Shared Memory in CUDA C/C++*. https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/.

[6] Intel. *Intel Core i9-9900K*. https://ark.intel.com/content/www/us/en/ark/products/186605/intel-core-i99900k-processor-16m-cache-up-to-5-00-ghz.html.

[7] JetBrains. *cuBool*. https://github.com/JetBrains-Research/cuBool.

[8] Vineeth Mekkat, Anup Holey, and Antonia Zhai. "Accelerating Data Race Detection Utilizing On-Chip Data-Parallel Cores". In: *Runtime Verification*. Ed. by Axel Legay and Saddek Bensalem. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 201–218. ISBN: 978-3-642-40787-1.

[9] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. "A GPU Implementation of Inclusion-Based Points-to Analysis". In: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '12. New Orleans, Louisiana, USA: Association for Computing Machinery, 2012, 107–116. ISBN: 9781450311601. DOI: 10.1145/2145816.2145831. URL: https://doi.org/10.1145/2145816.2145831.

[10] Anders Møller and Michael I. Schwartzbach. *Static Program Analysis*. 2021. URL: https://cs.au.dk/~amoeller/spa/.

[11] NVidia. *CuBLAS*. https://docs.nvidia.com/cuda/cublas/index.html.

[12] Nvidia. *CUDA Toolkit documentation*. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[13] NVidia. *NVIDIA AMPERE GA102 GPU ARCHITECTURE*. Tech. rep., pp. 44–45. URL: `https://images.nvidia.com/aem-dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf`.

[14] Tarun Prabhu et al. "EigenCFA: Accelerating Flow Analysis with GPUs". In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: Association for Computing Machinery, 2011, 511–522. ISBN: 9781450304900. DOI: `10.1145/1926385.1926445`. URL: `https://doi.org/10.1145/1926385.1926445`.

[15] Jaegeun Han & Bharatkumar Sharma. *Learn CUDA Programming*. Packt, 2019. ISBN: 78-1-78899-624-2.

[16] Jonas Svenningsen, Nicklas Hugöy, and Thorulf Neustrup. *GPU-accelerated Taint Analysis*. 2021.

[17] Wikichip. *Floating-Point Operations Per Second (FLOPS)*. `https://en.wikichip.org/wiki/flops`.

Appendix A

SC Semantics

$$env_V \in Env_V = Vars \rightharpoonup \mathbb{Z}$$

$$env_P \in Env_P = functionName \rightharpoonup (\langle stmt\rangle \times (\mathbb{N} \rightharpoonup Vars) \times env_P)$$

$$env_A \in Env_A = Vars \rightharpoonup \mathbb{N} \times (\mathbb{Z} \rightharpoonup \mathbb{Z})$$

$$\mathcal{Z} \in Literal \rightarrow \mathbb{Z}$$

$$\rightarrow_p \subseteq \langle prog\rangle \times Env_P \times \mathbb{Z}_\perp$$

$$\rightarrow_f \subseteq \langle funcDef\rangle \times Env_P \times Env_P$$

$$\rightarrow_s \subseteq \langle stmt\rangle \times Env_P \times Env_V \times Env_A \times Env_V \times Env_A \times \mathbb{Z}_\perp$$

$$\rightarrow_e \subseteq \langle expr\rangle \times Env_P \times Env_V \times Env_A \times \mathbb{Z}$$

$$(program)\frac{\begin{array}{c}\langle F_1, env_P\rangle \rightarrow_f env_P^1 \\ \ldots \\ \langle F_n, env_P^{n-1}\rangle \rightarrow_f env_P^n \\ env_P^n \vdash \langle S, []\rangle \rightarrow_s \langle env_V, ret\rangle\end{array}}{\langle F_1 \ \ldots \ F_n \ S, env_P\rangle \rightarrow_p ret}$$

$$(function\ def)\frac{env_P[func \mapsto (S, [1 \mapsto p_1 \ldots n \mapsto p_n], env_P)] = env_P'}{\langle int\ func(p_1 \ldots p_n)\{S\}, env_P\rangle \rightarrow_f env_P'}$$

$$(statements_{ret})\frac{\begin{array}{c}env_P \vdash \langle S_1, env_A, env_V\rangle \rightarrow_s \langle env_A'', env_V'', ret\rangle \\ env_P \vdash \langle S_2, env_A'', env_V''\rangle \rightarrow_s \langle env_A', env_V', ret'\rangle\end{array}}{env_P \vdash \langle S_1; S_2, env_A, env_V\rangle \rightarrow_s \langle env_A', env_V', ret'\rangle} \quad if \ \ ret = \perp$$

$$(statements_{earlyRet})\frac{env_P \vdash \langle S_1, env_A, env_V\rangle \rightarrow_s \langle env'_A, env'_V, ret\rangle}{env_P \vdash \langle S_1; S_2, env_A, env_V\rangle \rightarrow_s \langle env'_A, env'_V, ret\rangle} \quad if \;\; ret \neq \bot$$

$$(return)\frac{env_P, env_A, env_V \vdash e \rightarrow_e ret}{env_P \vdash \langle return\ e, env_A, env_V\rangle \rightarrow_s \langle env_V, ret\rangle}$$

$$(while\ true)\frac{\begin{array}{c} env_P, env_A, env_V \vdash e \rightarrow_e v \\ env_P \vdash \langle S, env_A, env_V\rangle \rightarrow_s \langle env''_A, env''_V, ret\rangle \\ env_P \vdash \langle while(e)\{S\}, env''_A, env''_V\rangle \rightarrow_s \langle env'_A, env'_V, ret'\rangle \end{array}}{env_P \vdash \langle while(e)\{S\}, env_A, env_V\rangle \rightarrow_s \langle env'_A, env'_V, ret'\rangle} \quad if \;\; v \neq 0 \wedge ret = \bot$$

$$(while\ ret)\frac{\begin{array}{c} env_P, env_A, env_V \vdash e \rightarrow_e v \\ env_P \vdash \langle S, env_A, env_V\rangle \rightarrow_s \langle env'_A, env'_V, ret\rangle \end{array}}{env_P \vdash \langle while(e)\{S\}, env_A, env_V\rangle \rightarrow_s \langle env'_A, env'_V, ret\rangle} \quad if \;\; v \neq 0 \wedge ret \neq \bot$$

$$(while\ false)\frac{env_P, env_A, env_V \vdash e \rightarrow_e v}{env_P \vdash \langle while(e)\{S\}, env_A, env_V\rangle \rightarrow_s \langle env_A, env_V, \bot\rangle} \quad if \;\; v = 0$$

$$(arrayInit)\frac{\begin{array}{c} env_P, env_A, env_V \vdash e_1 \rightarrow_e v_1 \\ .... \\ env_P, env_A, env_V \vdash e_n \rightarrow_e v_n \end{array}}{\begin{array}{c} env_P, env_V \vdash \langle int\ a[l] = \{e_1, ..., e_n\}, env_A\rangle \rightarrow_s \\ \langle env_A[a \mapsto (n, [0 \mapsto v_1, ..., i-1 \mapsto v_n])], env_V, \bot\rangle \end{array}} \quad \text{where } \mathcal{Z}(l) = n$$

$$(arrayAssign)\frac{\begin{array}{c} env_P, env_A, env_V \vdash e_1 \rightarrow_e i \\ env_P, env_A, env_V \vdash e_2 \rightarrow_e v \end{array}}{\begin{array}{c} env_P, env_V \vdash \langle a[e_1] = e_2, env_A\rangle \rightarrow_s \\ \langle env_A[a \mapsto (n, vals[i \mapsto v])], env_V, \bot\rangle \end{array}} \quad \text{where } \begin{array}{c} env_A(a) = (n, vals) \\ 0 \leq i < n \end{array}$$

$$(arrayExpr)\frac{env_P, env_A, env_V \vdash e \rightarrow_e i}{env_P, env_A, env_V \vdash a[e] \rightarrow_e vals(i)} \quad \text{where } \begin{array}{c} env_A(a) = (n, vals) \\ 0 \leq i < n \end{array}$$

$$(if\ true)\frac{\begin{array}{c} env_P, env_A, env_V \vdash e \rightarrow_e v \\ env_P \vdash \langle S_1, env_A, env_V\rangle \rightarrow_s \langle env'_A, env'_V, ret\rangle \end{array}}{env_P \vdash \langle if(e)\{S_1\}else\{S_2\}, env_A, env_V\rangle \rightarrow_s \langle env'_A, env'_V, ret\rangle} \quad if \;\; v \neq 0$$

$$(if\ false)\frac{\begin{array}{c} env_P, env_A env_V \vdash e \rightarrow_e v \\ env_P \vdash \langle S_2, env_A, env_V\rangle \rightarrow_s \langle env'_A, env'_V, ret\rangle \end{array}}{env_P \vdash \langle if(e)\{S_1\}else\{S_2\}, env_A, env_V\rangle \rightarrow_s \langle env'_A, env'_V, ret\rangle} \quad if \;\; v = 0$$

$$(assign)\frac{env_P, env_A, env_V \vdash e \rightarrow_e v}{env_P \vdash \langle a = e, env_A, env_V\rangle \rightarrow_s \langle env_A, env_V[a \mapsto v], \bot\rangle}$$

$$(funcCall)\;\frac{\begin{array}{c} env_P, env_A, env_V \vdash e_1 \rightarrow_e v_1 \\ \ldots \\ env_P, env_A, env_V \vdash e_n \rightarrow_e v_n \\ env'_P \vdash \langle S, env'_A[], env'_V[p(1) \mapsto v_1 \\ \ldots \\ p(n) \mapsto v_n]\rangle \rightarrow_s \langle env_A, env''_V, ret\rangle \end{array}}{env_P, env_A, env_V \vdash \langle func(e_1 \ldots e_n)\rangle \rightarrow_e ret}\;\; env_P(func) = (S, p, env'_P)$$

$$(exp)\;\frac{env_P, env_A, env_V \vdash exp_1 \rightarrow_e v_1 \quad env_P, env_A, env_V \vdash exp_2 \rightarrow_e v_2}{env_P, env_A, env_V \vdash \langle exp_1 \oplus exp_2\rangle \rightarrow_e v}\;\; where\;\; v_1 \oplus v_2 = v$$

$$(exp_{literal}) \quad env_P, env_A, env_V \vdash l \rightarrow_e v \quad if \;\; \mathcal{Z}(l) = v$$

$$(exp_{var}) \quad env_P, env_A, env_V \vdash var \rightarrow_e v \quad if \;\; env_V(var) = v$$

Appendix B

Code Samples

```
int g(n){
    out = 0;
    if(n){
        out = τ;
    }else{
        out = n;
    }
    return n;
}

int f(){
    n = τ;
    j = g(n);
    x = g(j);
    return n+j+x;
}

void main(j){
    a = f();  ⎫
        ⋮     ⎬ X times
    a = f();  ⎭
}
```

Listing B.1: SC program containing many taint sources which propagates a short duration

```
int g(int n){
    int out = 0;
    if(n){
        out = 5;
    }else{
        out = n;
    }
    return n;
}

int f(int a){
    int n = a;
    int j = g(n);
    int x = g(j);

    return x;
}

void main(int j){
    int a = f($);  ⎫
    int a = f(a);  ⎪
         ⋮         ⎬ X times
    int a = f(a);  ⎭
}
```

Listing B.2: SC program with a single taint source the is passed through function calls. This sample is designed to perform bad when parallelized.

```
int read(){
    return $;
}

int detour(int baba){
    int is = read();
    int you = 0;

    if(baba + is + you){
        baba = 5;
    }else{
        you = baba + is;
    }

    while(baba){
        baba = baba + baba;
    }
    return baba;
}

int takingmanyparams(int a, int b, int c, int d, int e){
    int value = 0;
    if(a){
        if(b){
            if(c){
                if(d){
                    value = 4+d;
                }else{
                    value = 3+c;
                }
            }else{
                value = 2+b;
            }
        }else{
            value = 1+a;
        }
    }else{
        value = 0;
    }
    int value = detour(a+b+c);
    int value1 = detour(d+e);
    int something = 0;
    int comb = 0;
    if(value){
        something = something + value;
        something = e;
        something = d;
        something = c;
        something = b;
        something = a;
    }else{
        comb = a+b;
        comb = b+c;
        comb = c+d;
        comb = d+e;
    }
    int iffer = 0;

    if(a+b+c+d+e){
        if(c+d){
            iffer = c+d;
        }else{
            iffer = a+b+e;
        }
    }else{
        if(b+e){
            iffer = b+e;
        }else{
            iffer = a+c+d;
        }
    }
    int er = a+b+c+d+e;
    return er;
}

int paramtester(){
    int t = read();
    int value1 = takingmanyparams(t, 2, 3, 4, 5);
    int value2 = takingmanyparams(1, t, 3, 4, 5);
    int value3 = takingmanyparams(1, 2, t, 4, 5);
    int value4 = takingmanyparams(1, 2, 3, t, 5);
    int value5 = takingmanyparams(1, 2, 3, 4, t);
    int value0 = takingmanyparams(1, 2, 3, 4, 5);
    return value0;
}

int main(){
    int value = paramtester();
    while(value){
        value = value-1;
    }
    if(value){
        int wat = read();
    }else{
        int wat = value;
    }

    int rut = detour(wat);
    return wat;
}
```

Listing B.3: A larger SC sample program

```
int read(){
    return $;
}

int comp(int i){
    int returnvalue = 0;
    if (i){
        returnvalue = 1;
    }else{
        returnvalue = i * (i-1);
    }
    return returnvalue;
}

int accumelate(int j, int in){
    int returnvalue = 1;
    if(in){
        int value = read();
    }else{
        int value = 1;
    }
    while(j){
        int callretvalue = comp(value);
        int callindexretvalue = comp(j);
        int combined = callretvalue + callretvalue;
        returnvalue = returnvalue * callretvalue;
        j = j - 2;
    }
    return returnvalue;
}

int main(){
    int index = 4;
    int data[5] = {3,5,7,9,11};
    int output[5] = {0,0,0,0,0};
    while (index){
        int callretvalue = accumelate(data[index], 0);
        output[index] = callretvalue;
        index = index - 1;
    }
    return output;
}
```

Listing B.4: Sample SC program with arrays

```
int broke(int d){
    int b = d;
    return b;
}

int main(){
    int a = 0;
    int c = $ + a;
    int ret = broke(c);
    return ret;
}
```

Listing B.5: Small SC sample program with where a function parameter is tainted