FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Improving casmi - The AST Interpreter for CASM

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Emmanuel Pescosta
Registration Number 1326934

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall
Assistance: Univ.Ass. Dipl.-Ing. Philipp Paulweber, BSc. (UNIVIE)

Vienna, 11th March, 2017

Emmanuel Pescosta	Andreas Krall

Corinthian Abstract State Machine

Language, Interpreter, and Compiler

The origin of the name *Corinthian is unclear,*
whether it is taken from "the letters,
the pillars, the leather, the place,
or the mode of behavior".

Puck. The Sandman by Neil Gaiman

https://casm-lang.org
github.com/casm-lang
twitter.com/casm_lang

Abstract

This bachelor's thesis builds upon `casmi`, which was introduced by Florian Hahn in [Hah14], and improves its runtime performance and memory consumption. `casmi` is an abstract syntax tree (AST) interpreter for the Corinthian Abstract State Machine (CASM) language, an abstract state machine (ASM) based programming language for accurate high-level modeling and analysis of soft- and/or hardware systems. The first part of this thesis focuses on the implementation of a new update set while the second part evaluates different techniques to optimize the hashing of ASM function arguments. Both parts are crucial to achieve good performance when executing large CASM specifications, as they occur in translation validation and processor simulation. Finally the improvements of `casmi` will be compared to the legacy version of `casmi` [LPK14] and to the `CoreASM` interpreter.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Translation validation is one approach to show that a compiler produces semantically equivalent output in respect to its input, which is a relatively important part of the certification procedure of safety-critical applications. The CASM language, which was introduced by Roland Lezuo in [LBK13], allows such translation validation to formally prove that a single compiler output is semantically equivalent to its input. This is done by symbolic execution of the compiler in- and output, which are described by CASM models and by proving the equivalence of both execution traces via a first-order logic theorem prover (see [Lez14]). For industrial-scale applications the resulting CASM models can become very large and thus the performance and memory efficiency of the CASM interpreter `casmi` is crucial for executing such models.

While `casmi` already delivers superior performance when compared to the first generation CASM interpreter (see [Hah14]), it suffers from some performance bottlenecks and memory leaks. Another big drawback is that the *update set* implementation, which is currently used by `casmi` (see [Pau14]), was developed in cooperation with an industry partner and therefore the CASM toolset could not be distributed freely and released as open source software.

1.2 Problem Statement

In the scope of this thesis a new *update set* implementation will be realized and compared to the previous implementation in terms of performance and memory efficiency. The new *update set* implementation should be efficient, easy to understand and test, and isolated from the rest of the AST interpreter with the goal that it can easily be reused in the intermediate representation (IR) interpreter, which will be realized in the near future.

Along with the new *update set* implementation a new memory allocator will be implemented, which is optimized for the allocation and deallocation patterns of the interpreter (see [Pau14]). The memory allocator should be good at handling stack like allocations while keeping the overall memory consumption as low as possible.

Furthermore the hashing of functions will be optimized and it should be evaluated which hashing technique is best for the requirements of `casmi`. Hashing plays a major role in the runtime performance of `casmi` given that it's involved in every single function update and lookup operation.

The result of this thesis should be a fast and memory efficient CASM AST interpreter, which doesn't require any proprietary module to run, so that the complete toolset can be released as open source software.

1.3 Methodological Approach

At the beginning the numeric and symbolic execution code from Florian Hahn [Hah14] must be adapted to the new front-end, which contains quite a few changes in regard to the AST visitation. Alongside with the adaption of the legacy code the *update set* handling must be implemented from scratch so that the interpreter can run in the first place.

As soon as the interpreter runs the existing code needs to be analyzed to detect bugs, crashes and weak spots of memory consumption and performance. After that the faults have to be analyzed and fixed to achieve a solid and test-covered base for further improvements and additions.

The next step is to implement multiple different hashing techniques and to compare them in regard to performance and memory consumption.

1.4 CASM Language

CASM is a statically-typed programming language build upon the concept of abstract state machines (ASM) which were introduced by Yuri Gurevich in [Gur95]. ASMs provide a formal way to model problems on arbitrary levels of abstraction, starting with an abstract ground model and performing stepwise refinement to get a more detailed model. ASMs can be used for accurate high-level modeling and analysis, both verification and validation by testing and simulating, of soft- and/or hardware systems (see [BS03]).

ASMs have a clear notion of state and state transitions. In CASM the state transitions are reflected by rules. Each agent has a program rule which will be called on each step, on start-up the program rule will be initialized with the rule defined by the **init** statement. Please note that CASM is currently limited to a single execution agent, multi-agent ASMs will be possible in a future version of `casmi`. On each step the rules produce a set of updates which are then applied to the global state causing a transformation of the

global state (see Definition 3.1.7). If a rule produces an inconsistent update set, i.e. if it contains multiple updates for the same location but with different update values (see Definition 3.1.6), no updates will be applied and instead the executing will be aborted. The execution of the CASM program will also be stopped if the update set of a step is empty, i.e. when the ASM reaches a final state, or when the program rule of the agent has no defined value, which is called *undef* in CASM.

A state maps locations to values. All functions are interpreted in states as total functions in the mathematical sense. Undefined function locations return the special value *undef* or the specified default value if the functions is a *defined* function. Functions have a specified arity and can be initialized via the **initially** keyword.

```
1  CASM
2
3  init swap
4
5  function x: Integer initially { 21 }
6  function y: Integer initially { 42 }
7
8  rule swap =
9  par
10     x := y
11     y := x
12 endpar
```

Listing 1.1: Swapping x and y in Parallel Rule

Listing 1.1 shows a simple CASM specification of the swap algorithm using parallel execution semantics. The initial global state is $\{(x(), 21), (y(), 42)\}$ where x and y are 0-ary dynamic functions. When executing this model the *swap* rule will produce an update set containing two updates $((x, ()), 42)$ and $((y, ()), 21)$. An update basically consists of a value and a location, which is an n-ary function with arguments denoted as $(f, (a_1, ..., a_n))$. At the end of the first step the current global state together with the update set will yield a new global state $\{(x(), 42), (y(), 21)\}$ where the values of x and y are swapped. This example clearly shows a big advantage of ASMs, namely the parallel execution of rules with the respect to the same global state.

```
1  CASM
2
3  init swap
4
5  function x: Integer initially { 21 }
6  function y: Integer initially { 42 }
7
8  rule swap =
9  seq
10     x := y
11     y := x
12 endseq
```

Listing 1.2: Swapping x and y in Sequential Rule

Listing 1.2 shows almost the same algorithm using sequential instead of parallel execution semantics. Statements of a sequential execution block are executed in sequential order in respect to the same temporary state. This behavior is comparable to traditional programming languages like C. After each statement the updates are merged into the temporary state of the sequential execution block, thus the updates produced by a single statement are visible to all following statements. This, in turn, means that statements can overwrite the changes made by previous statements, in parallel execution blocks this would lead to an inconsistent update set. The initial global state of this example is the same as in 1.1. The first statement of the *swap* rule produces an update $((x, ()), 42)$, this yields a temporary state $\{(x(), 42), (y(), 42)\}$. The second statement, which assigns the value of x from the temporary state to y, produces an update $((y, ()), 42)$ and applying it to the temporary state yields $\{(x(), 42), (y(), 42)\}$ and thus the final global state after the first step is $\{(x(), 42), (y(), 42)\}$ where the swapping wasn't performed correctly. A temporary variable holding the original value of x before it will be overwritten is required to solve the problem of this implementation.

Both examples show that the choice of the execution semantics, either sequential or parallel, has a significant impact on the semantics of the whole program. Parallel and sequential execution can also be interleaved, see as an example the Listing 3.2.

A detailed description of the CASM language can be found in [Lez14] and [Pau14].

CHAPTER 2

Related Work

2.1 ASM Interpreters

The first C++-based version of `casmi` was introduced by Inführ in [Inf13]. The aim of his project was to replace the old Python-based prototype interpreter *casmintr*, written by Lezuo as part of his PhD thesis [Lez14], by a fast C++ implementation which can share most of the frontend related code (lexer, parser and type checker) with the CASM compiler. As a result of the re-write the interpreter was at least 190 times faster than the prototype implementation (see Chapter 6 in [Inf13]).

The second version of `casmi`, which was a major re-write of the first one, was introduced by Hahn in [Hah14]. His interpreter uses the update set implementation of the CASM compiler (see [Pau14]) instead of the default C++ hash map implementation used by Inführ's interpreter to achieve a better performance. Also some memory management and type inference flaws of the first version of `casmi` have been fixed during the re-write. Although the whole re-implementation was done as a clean room implementation the `casmi` code still couldn't be released as open source software because of the usage of the proprietary update set implementation. As a result of the second re-write the performance could be increased once more by about 2-4 times compared to the first version of `casmi`.

Farahbod introduced `CoreASM` in [FGG07] as an research effort to make ASM specifications executable. The focus of this project is to be as close as possible to the mathematical definition of pure ASMs [FGG07]. While CASM, which was inspired by CoreASM, is a statically-typed language, the CoreASM language is a completely untyped language to be closer to a real abstract model. The interpreter itself is completely written in Java. The design is based around an highly extensible core engine so that almost all aspects of the interpreter can be extended through plug-ins. In addition to the interpreter the CoreASM project provides many additional tools such as a graphical user interface to visualize and debug the simulation runs of ASM specifications.

2.2 Update Set Handling

In [Sch01] Schmid introduced a compiler for the FALKO project to translate ASM specifications into efficient C++ code. The main aim of this compiler was to generate fast code for efficient execution of industrial-strength ASM specifications. To reach this target an efficient update set handling has been implemented. The basic idea of his implementation is to use *double buffering* which is well known from from graphics rendering to avoid tearing and rendering artifacts. Each function location basically uses a state counter and two variables to hold the saved value and the new value. During a rule execution the function state counters together with the global state counter are used to keep track if a function location has already been updated during a transition, which would require a consistency check for each additional function write operation. Firing of updates is just a matter of copying the new value into saved value and adjusting the counters of each function location, which has been changed during a transition. An optimized version of this method (more like *page flipping*) is described in [Sch01] which avoids the overhead of the value copying by using a boolean value to indicate which of both variables is currently used for reading and for writing. This update set handling approach completely avoids the expensive update set hashing which results in a huge performance gain. The major disadvantage of this approach is that it only supports parallel execution semantics which makes it impractical for `casmi`.

Another approach for efficient update set handling was proposed by Lezuo in [Lez14] and implemented by Paulweber in [Pau14] as part of their CASM compiler work [LPK14]. This approach is based on the assumption that the update set is much smaller than the global state [Lez14], so it is more efficient to track only the changes instead of duplicating the whole state. It basically uses a single hash map to keep track of all generated updates during a transition. Due to compilation reasons a pseudo state counter was introduced. With the help of this pseudo state counter nested update sets can be realized and thus it supports the interleaving of parallel and sequential execution blocks. Updates can be fired by iterating through all updates of the update set and apply each of them to the global state. Paulweber's implementation was also used by the previous version of `casmi` and will be compared with the new implementation in Section 3.4.

2.3 Hashing Methods

In [RAD15] Richter et al. did an analysis of multiple different hashing methods for 64-bit integer keys. They compared *chained hashing*, *linear probing*, *quadratic probing*, *robin hood hashing* and *cuckoo hashing* under different conditions, like different maximum load factors, using different hashing functions. As a result of their work they suggested a decision graph which should help to select a good hash map for the given requirements.

CHAPTER 3

Update Set

3.1 Definitions

The following definitions are taken from Chapter 2.4 "Detailed Definition (Math. Foundation)" of the book [BS03]:

Defintion 3.1.1. *A **signature** Σ is a finite collection of n-ary functions where $n \in \mathbb{N}_0$.*

Defintion 3.1.2. *A **state** $\mathfrak{U}$ for the signature Σ is a non-empty set X, the **superuniverse** of $\mathfrak{U}$, together with an interpretation $f^{\mathfrak{U}} : X^n \to X$ of each function name $f \in \Sigma$. The default value of the superuniverse is **undef**.*

Defintion 3.1.3. *A **location** of $\mathfrak{U}$ is a pair $(f, (t_1, ..., t_n))$, where f denotes the name of an n-ary function and $(t_1, ..., t_n) \in X^n$ are function arguments.*

Defintion 3.1.4. *An **update** is a pair $u = (l, v)$, where l denotes the location and $v \in X$ the value of the update.*

Defintion 3.1.5. *An **update set** is a collection of updates $\mathcal{U} = \{u_1, u_2, ..., u_n\}$.*

Defintion 3.1.6. *An update set $\mathcal{U}$ is called **consistent** if it has no clashing updates, i.e., if for any location l and all elements u, w holds that if $(l, u) \in \mathcal{U}$ and $(l, w) \in \mathcal{U}$ then $u = w$.*

Defintion 3.1.7. ***Firing** of a consistent update set $\mathcal{U}$ into a state $\mathfrak{U}$ yields a new state $\mathfrak{U} + \mathcal{U}$ with the same superuniverse as $\mathfrak{U}$ so that for every location $l \in \mathfrak{U}$ holds:*

$$(\mathfrak{U} + \mathcal{U})(l) = \begin{cases} u & \text{if } (l, u) \in \mathcal{U} \\ \mathfrak{U}(l) & \text{if } \nexists u : (l, u) \in \mathcal{U} \end{cases}$$

Defintion 3.1.8. *Parallel execution semantics of P **par** Q is given by executing both rules in respect to the same state $\mathfrak{U}$.*

Defintion 3.1.9. *Sequential execution semantics of P **seq** Q is given by first executing P in the state $\mathfrak{U}$ which yields the update set $\mathcal{U}$, then executing Q in the resulting state $\mathfrak{U} + \mathcal{U}$.*

3.2 Implementation

3.2.1 Update

An update is represented through a simple C structure which stores the updated value, a pointer to the function arguments, the id of the function and the source code location of the update statement. Anything expect of the updated value is only used to provide useful debug information to the user when clashing updates are detected while adding or merging updates. To achieve optimal performance all updates are allocated by a block allocator [Pau14] which is described in Section 3.2.4.

3.2.2 Update Set

An *update set* keeps track of all updates of a specific execution block. Depending on the actual execution semantics of the block a sequential or a parallel *update set* has to be used. The *update set* itself uses a single-linked hash-map with the *location* as key to store the updates in an efficient manner.

The basic operations provided by an *update set* are adding and searching for updates for a specific *location*. The behavior of adding and searching for updates depends on the actual type of the *update set*. Adding an update into an *update set* with sequential execution semantics will either insert the update when no update for the given *location* exists or overwrite an existing update with the same *location*. Whereas adding an update into an *update set* with parallel execution semantics will either insert the update when no update for the given *location* exists or throw a conflict exception when an update for the same *location* but with a different update value exists. Given that clashing updates are detected and handled at the time of insertion an *update set* is always consistent. Adding of an update to an *update set* takes constant time regardless of the type.

Searching for updates requires a lookup in multiple *update sets*, starting with the *update set* of the current execution block and walking up the *fork tree* until the root node is reached. Each sequential *update set* on this path needs to be checked until an *update set* has been found which contains an update for the requested *location*. *Update sets* with parallel execution semantics are ignored (see Definition 3.1.8). If none of these *update sets* contains an update for the requested *location* a **nullptr** will be returned, indicating the absence of an update. The complexity of searching for an update is linear in the number of nested execution blocks, i.e., linear in the number of enclosing *update sets*.

To support the interleaving of sequential and parallel execution semantics of *CASM*, *update sets* can be forked and merged. Forking an existing *update set* produces a new *update set* with the demanded execution semantics as well as a link to the *update set*

from which the fork originates from, also called the *parent update set*. The forking of an *update set* takes constant time. All the *update sets* together form a *fork tree* where every tree node can have an arbitrary number of forks. The root node of an *fork tree* corresponds to the *update set* of the outermost execution block and thus has no *parent update set*. At the end of each block the updates need to be applied to the *update set* of the surrounding block, this is done by merging an *update set* into its *parent update set*. The merging itself is rather simple, it just iterates over all updates of an *update set* and adds each update into its *parent update set*, thus the complexity of merging is linear in the size of the *update set* to merge. Using the add operation of the *parent update set* for merging guarantees that the updates are always applied under the execution semantics of the *parent update set*.

The firing of updates hasn't been implemented into the *update set* itself because this operation heavily depends on the interpreter specific implementation of the global state. But the required work is just a matter of iterating through all updates of the *update set*, applying each of them to the global state and clearing the *update set* afterwards.

3.2.3 Update Set Manager

The update set manager keeps track of all *update sets* by maintaining a stack of them where the top of stack denotes the current *update set*. The manager provides almost the same operations as the *update set*. Each operation basically forwards the request to the current *update set* or if the stack is empty it takes appropriate action. The fork operation adds a fork of the current *update set* onto the top of the stack. The merge operation merges the current *update set* into its parent and removes it from the stack.

```
1  Visitor::visit_seqblock_pre()
2      manager.fork( UpdateSet::Type::Sequential )
3
4  Visitor::visit_seqblock_post()
5      manager.merge()
6
7  Visitor::visit_parblock_pre()
8      manager.fork( UpdateSet::Type::Parallel )
9
10 Visitor::visit_parblock_post()
11     manager.merge()
12
13 Visitor::visit_update()
14     manager.add( location, value )
15
16 Visitor::visit_function()
17     update = manager.lookup( location )
18     if update is absent {
19         return global_state.value( location )
20     } else {
21         return update.value()
22     }
```

Listing 3.1: Visitor Methods using the Update Set Manager

The manager was added to simplify the usage of *update set* implementation in the AST interpreter. Listing 3.1 shows more or less how the AST interpreter uses the *update set* manager during execution.

3.2.4 Memory Allocator

While most systems use a very general memory allocator which supports allocation and deallocation of arbitrary memory on the heap but which suffers from memory fragmentation and decreased performance, `casmi` can make use of an optimized allocator to make allocations and deallocations of updates fast due to the transactional semantics of the language itself. While executing ASM rules `casmi` has to allocate an update object for each update statement in the rule, at the end of the rule execution after the *update set* has been merged back into the global state, all previously allocated update objects have to be freed simultaneously. This ever-growing and release everything at once behavior of `casmi` makes it perfectly suitable for an allocator which supports stack-like allocation and deallocation. Ideally the freeing of all updates is just a matter of setting the top of stack pointer to the bottom of the stack [Pau14].

Based on the idea from [Pau14] and [LK12] a fixed-size block allocator has been implemented, which supports fast allocation of updates and deallocation of all updates in one go. The block allocator requests fixed-size memory blocks from the memory pool whenever a new block is needed, all blocks together form a single-linked list. The memory pool is used to recycle unused blocks to avoid the invocation of the rather expensive system allocator as good as possible. If the memory pool is empty the system memory allocator will be used to allocate a block-size aligned memory block. Updates are allocated by increasing the top of stack of the current block and returning the previous top of stack memory address. Freeing of all updates is just a matter of iterating through all blocks and put each block back into the memory pool.

Each block stores some management information along with the user data. The management information, which is stored below the user data, consists of the current top of stack and a pointer to the previous block. Given that all blocks are block-size aligned and the block-size must be a power of two, it is relatively simple to find the responsible block for a given memory address. The memory address of the block can be calculated by $A_B = A_G \& (\sim(Blocksize - 1))$ where A_G denotes the given stack address and A_B is the address of the block, the operators are bitwise. With the help of the reset method, which uses this technique, it is possible to use the memory allocator as a stack which supports push and pop operations in addition to free all.

3.3 Update Set Size Prediction

To further improve the performance of the *update set* handling, a set of CASM rules have been developed to predict the possible upper bound number of updates produced by each execution block and CASM rule. These upper bounds can help to reduce the amount of

rehashing by using a properly sized *update set* in the first place and they make it also possible to eliminate the forking and merging of empty *update sets*. Additionally the forking and merging of *update sets* with only a single update can be avoided, because such updates can be directly committed to the enclosing *update set*. No size prediction can be performed if a CASM rule contains recursive calls both directly and indirectly through any chain of other CASM rules, or when a CASM rule contains unbounded `forall` or `iterate` rules.

Rules:

Rule 3.3.1. $U(f(t_1, ..., t_n) := t) = 1$

Rule 3.3.2. $U(P \textbf{ par } Q) = U(P) + U(Q)$

Rule 3.3.3. $U(P \textbf{ seq } Q) = U(P) + U(Q)$

Rule 3.3.4. $U(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q) = max(U(P), U(Q))$

Rule 3.3.5. $U(\textbf{case } \varphi \textbf{ of } \{ \ c_1 : P_1, c_2 : P_2, ..., - : P_n\}) = max(U(P_1), U(P_2), ..., U(P_n))$

Rule 3.3.6. $U(\textbf{let } x = t \textbf{ in } P) = U(P)$

Rule 3.3.7. $U(\textbf{rule } r(t_1, ..., t_n) = P) = U(P)$

Rule 3.3.8. $U(\textbf{call } r(t_1, ..., t_n)) = \begin{cases} U(r(t_1, ..., t_n)) & \textit{if caller } \neq r \\ \infty & \textit{otherwise} \end{cases}$

Rule 3.3.9. $U(\textbf{forall } x \textbf{ in } \varphi \textbf{ do } P) = \begin{cases} 0 & \textit{if } U(P) = 0 \\ |\varphi| * U(P) & \textit{if } \varphi \textit{ is bounded} \\ \infty & \textit{otherwise} \end{cases}$

Rule 3.3.10. $U(\textbf{iterate } P) = \begin{cases} 0 & \textit{if } U(P) = 0 \\ \infty & \textit{otherwise} \end{cases}$

Rule 3.3.11. $U(r) = 0$ *... for all other rules*

These rules have been implemented in a new AST interpreter pass which annotates the AST nodes with the predicted number of updates. Given that no other optimizations like redundant update elimination [Lez14] (causes overestimation of upper bound) or constant folding/propagation (prevents loop bound estimation of Rule 3.3.9) are performed in the AST interpreter the measurable performance improvement of this additional pass is rather limited for most benchmarks. However in the IR interpreter [PZ16] where all these optimizations are possible the outcome of the update set size prediction will be useful for further optimization steps like preventing unnecessary *update set* forks, which will result in a reduced number of bytecode instructions and memory allocations.

This information can also be used to select the optimal data structure for the predicted number of updates and thus emitting the correct bytecode instructions. For a small number of updates a simple array is far more efficient than a fully fledged hash map. In [Lez14] the idea of using arrays for small update sets has already been implemented, but an array is always used regardless of the number of updates. If the array becomes to large an expensive upgrade to a hash map has to be performed. This upgrade can be completely avoided by using a hash map right from the beginning if the predicted number of updates is big enough.

Example:

```
1  CASM
2
3  init GameOfLife
4
5  function alive : Integer * Integer -> Boolean defined { false }
6
7  derived aliveNeighbours(x : Integer, y : Integer) =
8      asInteger(alive(x - 1, y - 1)) +
9      asInteger(alive(x, y - 1)) +
10     asInteger(alive(x + 1, y - 1)) +
11     asInteger(alive(x - 1, y)) +
12     asInteger(alive(x + 1, y)) +
13     asInteger(alive(x - 1, y + 1)) +
14     asInteger(alive(x, y + 1)) +
15     asInteger(alive(x + 1, y + 1))
16
17 rule UpdateCell(x : Integer, y : Integer) = // 10. 1 update
18     let c = aliveNeighbours(x, y) in        //  9. 1 update
19         if c = 3 then par                   //  7. max(1, 1) -> 1 update
20             alive(x, y) := true             //  1. 1 update
21         endpar else if (c < 2) or (c > 3) then par
22             alive(x, y) := false            //  1. 1 update
23         endpar
24
25 rule GameOfLife =                          // 10. 100 updates
26     forall x in [1..10] do                  // 12. 10 * 10 -> 100 updates
27         forall y in [1..10] do              // 12. 10 * 1 -> 10 updates
28             call UpdateCell(x, y)           // 11. 1 update
```

Listing 3.2: Update Set Size Prediction of Conway

Listing 3.2 demonstrates the use of the update set size prediction rules. Initially the lines 20 and 22 are evaluated by applying the Rule 3.3.1, both lines produce exactly one update. Then the enclosing **if** statement can be evaluated by applying the Rule 3.3.4. Given that either the **then** or the **else** block of an **if** statement is executed, the upper bound can simply be determined by taking the maximum value of both blocks without the need of any knowledge about the **if** condition. In this example the maximum number of updates produced by the **if** statement is one and so the resulting upper bound of

updates produced by the *UpdateCell* CASM rule is one. The call on line 28 produces at most one update because of the upper bound of *UpdateCell*. By applying the Rule 3.3.9 to the `forall` loop on line 27 we get a maximum number of 10 updates because the interval 1..10 causes exactly 10 loop iterations and every loop iteration produces one update when assuming the worst case scenario. Doing the same for line 26 gives the maximum number of 100 updates which is the upper bound of updates produced by the *GameOfLife* CASM rule. Given that the *UpdateCell* CASM rule produces only a single update, no *update set* forking is required when calling this rule, thus the update produced by *UpdateCell* could directly be committed to the *update set* of the inner `forall` loop of the *GameOfLife* CASM rule without any intermediate *update sets*.

3.4 Comparison

The legacy implementation uses the concept of **pseudo states** [Pau14] to represent the nested composition of *update sets*. A *pseudo state* basically reflects the depth of the nested composition, starting with 0, and the type of the execution block. An even number equals a block with parallel and an odd number equals a block with sequential execution semantics. Whenever an *update set* has to be forked or merged, the *pseudo state* counter has to be increased or decreased. The keys for the *update set* are 64 bit values, where the bits 0-15 reflect the *pseudo state* of the corresponding block of an update, the remaining bits are filled with lower bits of the memory address of the slot which holds the *location* value. As stated in [Lez14] this has some (theoretically) limitations like a maximum number of 65536 nested states and possible key collisions if the memory addresses of the slots only differ in the uppermost 16 bits. Additionally this key composition has some negative effects on the quality of the hash code especially when using bit-mask compression which will be discussed in Chapter 4. The new implementation doesn't have such limitations. The maximum number of nested states is theoretically unlimited and it supports the full 64 bit memory space without producing key collisions because the key for the *update set* is basically the memory address of the slot which holds the *location* value.

While the old implementation spread the update set handling all over the AST execution code, the new implementation properly encapsulates all this to make it easier to understand and use, easier to unit test and also reusable so that this module can be shared between the AST and IR interpreter. This results in a clean separation between update set handling and AST execution.

Another advantage over the legacy implementation is that sequential and parallel executing semantics can freely be mixed without being restricted to the *pseudo state* and it's also possible to start with sequential execution blocks, which was previously impossible due to the *pseudo state* limitation. This has greatly simplified the implementation of the numeric and symbolic AST interpreter. All restrictions, checks and work arounds to get the correct *pseudo state*, e.g. by wrapping the top-level block of a rule with a parallel execution block when the top-level block is a sequential execution block, could be dropped.

Given that in the new implementation the consistency of the *update set* is checked at the time of insertion of an update and that the new implementation uses one hash map for each *update set* it is possible to optimize some special cases. Merging of an *update set* into an *parent update set* which is empty can be optimized by simply swapping the hash tables of both *update sets*, this avoids a huge amount of rehashing.

Currently the interpreter uses the memory address of the function location object as key for the update set as described above. But the new update set implementation doesn't necessarily rely on such integer keys contrary to the legacy implementation. The key can also be for example a real location object which contains the function and the arguments, which is more similar to the notation of an ASM location (see Definition 3.1.3).

CHAPTER 4

Location Hashing

4.1 Terminology

The *initial bucket* is the bucket, which a given entry initially maps to without doing any collision resolution.

The *probe sequence length* (PSL) is the length of a sequence of hash map entries, starting from the *initial bucket*, which needs to be checked when searching or inserting entries into the hash map.

4.2 Hashing Schemes

Inserting various entries into a hash map causes collisions sooner or later. To resolve such collisions efficiently different collision resolution strategies have been developed in the last few decades (see [RAD15]). For this thesis four well-known and widely used hashing schemes have been selected. Each one has been implemented and evaluated in regard to `casmi`'s requirements.

4.2.1 Chained Hashing

Chained hashing [CSRL01] is a relatively simple collision handling approach. Each bucket maintains a list of entries, which belong to the same initial bucket. An entry can be retrieved by iterating through the entry list of the initial bucket until an entry for the given key could be found or the end of the list is reached, in this case no entry with the given key exists in the hash map. The insertion of a new entry is just a matter of appending the entry to the entries list of the initial bucket. This collision handling approach may provide sub-optimal performance due to the additional cache misses which occur while traversing the linked-list and thus some slightly optimized versions of this approach exist (see [RAD15]).

4.2.2 Linear Probing

Linear probing [CSRL01] is a well known open addressing scheme. It uses the hashing function $h_l(k,i) = (h'(k) + i) \bmod m$ to resolve collisions, where $h'(k)$ is a hashing function for key k, m is the capacity of the hash map and i denotes the probing iteration. The probing starts from the initial bucket and on each iteration the index of the bucket, which is to be inspected is calculated by $h_l(k,i)$. The probing can be aborted either when a bucket for the given key could be found or if the currently inspected bucket is empty, in this case it's guaranteed that no bucket for the given key exists in the hash map.

4.2.3 Quadratic Probing

Quadratic probing [CSRL01] is another widely used open addressing scheme. It uses the hashing function $h_q(k,i) = (h'(k) + c_1 * i + c_2 * i^2) \bmod m$ to resolve collisions, where $h'(k)$ is a hashing function for key k, m is the capacity of the hash map and i denotes the probing iteration. The choice of the constants c_1 and c_2 together with the capacity m are crucial to eventually reach all buckets of the hash map. Using $c_1 = c_2 = 1/2$ as well as a a power of two capacity gurantees that every bucket will be considered in worst-case (see [CSRL01]). This makes it possible to replace the expensive modulo operation of $h_q(k,i)$ by fast bit-masking. The probing procedure is the same as in *linear probing* but using $h_q(k,i)$ instead of $h_l(k,i)$.

4.2.4 Robin Hood hashing

Robin Hood hashing was introduced by Pedro Celis in [Cel86] as a reordering scheme which only requires minor modifications to standard search and insertion algorithms. In this thesis this reordering scheme will be used on top of *linear probing* [RAD15]. The main difference compared to the normal *linear probing* is that entries which are already stored in the hash map may be moved as new entries are inserted into the hash map. The idea behind it is to reduce the maximum PSL of the hash map by moving the entries around based on the distance to their initial bucket. The reordering basically minimizes the variance in distance to the initial bucket for all entries. According to [Cel86] this results in a very efficient hash map due to the fact that the probe sequence length has a small and almost constant variance, it requires far less probes on average to perform a successful search even when the table is almost full. Each bucket stores the PSL of the entry it holds, this information is used to optimize the look up of entries. The probing procedure is the same as in *linear probing* with the addition that the probing can also be aborted if the current probing distance is larger than the stored PSL of the currently inspected bucket.

4.3 Arguments Hashing Approaches

The goal of the arguments hashing is to reduce the multidimensional function arguments to a single value, which can then be used in a hash table. Hash collisions, meaning that

different function arguments produce the exact same hash value, should be avoided as good as possible while avoiding the huge computational overhead of cryptographic hash functions.

4.3.1 FNV-1A

The FNV algorithm was created by Fowler, Noll and Vo in 1991 and is specified in [FNV]. It is a non-cryptographic hash function which can produce 32-, 64- and up to 1024-bit hash values. FNV-1A is a slightly modified version of the basic FNV which has a better avalanche characteristic [FNV], meaning a small change in the input changes the output significantly. This algorithm has the advantage that it is really easy to implement, it needs only a few lines of code and it can easily be reused on 32- and 64-bit systems by simply using a different *offset basis* and *fnv prime*.

4.3.2 Murmur3

The Murmur3 [App] hashing algorithm is another widely adopted hash function which was created by Appleby in 2008. It was designed as a fast and non-cryptographic hashing algorithm which produces high quality hashes. Optimized version for 32- and 64-bit systems are available.

4.3.3 SipHash

SipHash was introduced by Aumasson and Bernstein in [AB12] as a pseudo random function optimized for short inputs with well-defined security goals to protect hash-tables against hash-flooding denial-of-service attacks. It is a keyed cryptographic hash function which computes a 64-bit hash from a 128-bit secret key and a variable-length data array. The results in [AB12] show that this algorithm is almost on par with other state-of-the-art non-cryptographic hashing algorithms. The authors of [AB12] propose that SipHash should be used as a hash function for hash tables.

4.4 Implementation

All the different hash map implementations share a common base class which provides all the required methods and iterators to the user. Each hash map type specializes this base class and provides the implementation for searching and inserting new hash map entries as well as the structure of entries and buckets. These specialized methods and structures are used in the base class to provide insert, insert-or-assign and lookup methods. No remove methods are provided basically because they aren't needed in `casmi`'s use case. The implementation of the hash map makes it relatively easy to switch between different compression functions and resizing behaviors, the default compression function is masked-hashing with simply doubling the capacity on each resize. Hash map entries are allocated using the previously introduced *block allocator* (see Chapter 3.2.4), this avoids a huge amount of small entry allocations, which usually happens with *chained*

hashing. This is possible because the user can only add new items to the hash map during the life time of the hash map and on destruction all previously allocated entries are freed simultaneously. All entries have a link to their predecessor, basically forming a single-linked list which is used by the hash map iterator. Due to the fact that all entries are allocated in blocks, a good memory locality can be achieved when using the *chained hashing* approach.

The function arguments are basically a collection of `casmi` value objects, each one storing the value type and the actual data which can also be a pointer to another data structure. Because of this the data of the arguments are distributed in memory, thus it isn't possible to use existing implementations of the mentioned hashing algorithms which operate on a contiguous block of memory. Instead the hash algorithms have been re-implemented so that they can operate on `casmi`'s value objects directly. Given that the hashing of function arguments is relatively costly, each bucket caches the calculated hash value.

The global state is modeled as a list of hash maps. Each CASM function has a specified id, a monotonically increasing number, which is used as a list index. For each function a hash map exists which stores the location value pairs.

4.5 Comparison

The old version of `casmi` used a pretty simple algorithm to compute hash codes from function arguments (see [Hah14]). It simply summed up the hash values of all arguments and returned the final value (see Algorithm 1). While this was a really fast hashing function (in terms of cpu cycles per byte) it produced a huge amount of hash collisions. This had the dramatic consequence that the performance of the function states, which were using hash tables from the C++ stdlib internally to achieve constant lookup and insertion times, dropped significantly and the hash tables of the function states basically became linked lists with linear lookup and insertion times. For example in the Conway specification (see Listing 3.2) the function arguments $(4, 0)$, $(3, 1)$, $(2, 2)$, $(1, 3)$, $(0, 4)$ of the *alive* function would all produce the exact same hash code when using Algorithm 1. This made the execution time of CASM specifications basically unpredictable slow in some situations.

Algorithm 1 Legacy arguments hashing [Hah14]

```
1: function ArgumentsHash(args)
2:     h ← 0
3:     foreach arg ∈ args do
4:         h ← h + ValueHash(arg)
5:     end foreach
6:     return h
7: end function
```

CHAPTER 5

Evaluation

In this section the performance and memory efficiency of `casmi` will be compared to the legacy interpreter [Inf13] denoted as `casmi-legacy`. Additionally the performance of the interpreter will be compared to the `CoreASM` interpreter [FGG07]. Sadly it wasn't possible to compare it to Hahn's version of `casmi` [Hah14] because there was no running version of his interpreter available. The memory and runtime measurements include everything from starting the interpreter until the interpretation of the specification is done and the process has finished.

5.1 Environment

The hard- and software configuration of the machine which was used for running the benchmarks are described in Table 5.1 and additionally the applied flags are described in Table 5.2. The new and the legacy version of `casmi` have been compiled within the same environment. Commit *e02dd07* of the *libcasm-fe* repository together with the commit *be1941d* of the *casmi* repository have been used to compile `casmi`.

5.2 CASM Specifications

The performance and memory evaluation has been performed with multiple different CASM specifications. The *Conway* benchmark is based on the Conway ASM specification from [BS03]. The *Updateset* benchmark has been written by Paulweber, as part of his work on [Pau14], as a stress test for the update set implementation. The other benchmarks have been written by Hahn and Lezuo as part of their work on `casmi` (see [LK12] and [Hah14]) and those benchmarks were already used in earlier CASM interpreter and compiler evaluations. All of these benchmarks have also been adapted to `casmi-legacy` as well as the `CoreASM` interpreter.

CPUs	Intel Core i7-6700k (4.00GHz, 8MB Cache)
Main Memory	16GB DDR4-2133 (2 x 8GB)
Operating system	Linux 64 bit (Arch Linux)
Compiler	clang 3.9.1 (Arch 3.9.1-2) 20170126
Heap Profiler	Massif (valgrind-3.12.0)
Java	OpenJDK (build 1.8.0_121-b13)
Python	3.6.0
pytest	3.0.6

Table 5.1: Hard- and Software Configuration of the Benchmarking Environment

Clang	-std=c++11 -Wall -O3 -DNDEBUG
Massif	–stacks=yes –heap=yes
`casmi`	–ast-exec-num
`casmi-legacy` (Conway)	-M 55
CoreASM	-y -p

Table 5.2: Applied Flags

The *Bubblesort* benchmark iteratively sorts 200 items using the bubble sort algorithm. This benchmark performs a huge amount of steps, on each step a relatively small update set will be applied to the global state.

The *Conway* benchmark implements the well-known cellular automaton introduced by Conway in [Con70]. The universe is an infinite two-dimensional grid. Each grid cell can either be dead or alive. The next state of each cell depends on the actual state of its neighboring cells. The transition rules determine if a cell dies or becomes alive. The evolution of the cellular automaton depends on its initial state. The *Conway* benchmark uses an initial state[1] where all cells have died after 54 generations, meaning that the interpretation will stop because no further transitions can be made. This benchmark produces only medium-sized update sets on each step but performs a lot of function lookups in the two-dimensional grid. This is the only benchmark which uses multi-dimensional function locations.

The *Fibonacci* benchmark calculates the fibonacci number of 7500 using a linear programming approach. Due to the current representation of integer numbers in `casmi` (64-bit signed integer) the outcome isn't correct because of overflows. This benchmark produces only a single but relatively huge update set and heavily relies on recursion.

The *Quicksort* benchmark sorts 300 items using the quick sort algorithm and produces small update sets. The *Sieve* benchmark is a prime siever which determines all prime

[1] `https://de.wikipedia.org/wiki/Datei:Game_of_life_U.gif`

numbers between 0 and 10000, this produces one huge update set. Both benchmarks heavily rely on sequential execution.

The *Gray* benchmark calculates gray codes for all 16-bit numbers. This benchmark heavily relies on arithmetic operations and rule invocations.

Finally the *Updateset* benchmark performs a huge number updates, forks and merges, testing the overall performance of the update set implementation. It interleaves parallel and sequential scopes and overrides the same locations a good few times.

5.3 Execution Time

Benchmark	`casmi`	`casmi-legacy`	`CoreASM`
Bubblesort	2,183.54 ms	5,784.10 ms	-
Conway	1,069.45 ms	4,632.15 ms	23,557.2 ms
Fibonacci	16.03 ms	26,500.64 ms	45,518.5 ms
Gray	3,069.30 ms	7,339.59 ms	102,472.5 ms
Quicksort	26.18 ms	83.87 ms	-
Sieve	29.72 ms	253.76 ms	21,272.2 ms
Updateset	4,202.57 ms	16,923.18 ms	287,136.6 ms

Table 5.3: Execution Times of Different Benchmarks

Benchmark	`casmi-legacy/casmi`	`CoreASM/casmi`
Bubblesort	2.65	-
Conway	4.33	22.03
Fibonacci	1,652.91	2,839.10
Gray	2.39	33.39
Quicksort	3.20	-
Sieve	8.54	715.70
Updateset	4.03	68.32

Table 5.4: Speedup of `casmi` compared to `casmi-legacy` and `CoreASM`

Table 5.3 shows the minimum execution time of at least five repetitions measured by Pytest. Table 5.4 shows the achieved speedup of `casmi` compared to `casmi-legacy` and `CoreASM`.

The performance improvements (expect of conway) are mostly on par with the results of Hahns interpreter (see [Hah14]). Yielding the same performance is a good outcome, given that a lot of missing value validations and sanity checks have been added to the interpreter. Furthermore, the function arguments hashing algorithm (see Section 4.3) has also been replaced by another one to avoid a huge amount of key collision when

using multi-dimensional functions. All these changes together with the more advanced memory handling produce a lot of additional overhead when executing these relatively small benchmarks.

5.4 Memory Consumption

The memory consumption has been measured with Massif which is part of the Valgrind analysis framework[2]. Heap and stack profiling has been enabled. The graphs and memory peak statistics have been generated via the open source tool Massif-Visualizer[3].

Bubblesort 5.6a creates the items before sorting, so the heap as well as the stack consumption is constant while sorting the items.

Conway 5.6c generates a lot of living cells at the start, thus the fast increase in the memory consumption at the very beginning. As soon as no new cells are generated the memory consumption is constant. The memory consumption isn't decreasing because dead cells are not deleted from the global state.

Fibonacci 5.6e uses recursion thus the fast (stack) memory increase in the first quarter. The memory usage is constantly decreasing after the peak because the algorithm just uses the partial results from the rule calls and calculates the result.

Gray 5.6g only uses a fixed number of temporary variables for the conversion. The gray codes are only displayed on the screen but not stored in the global state, thus the memory consumption of this benchmark is constant.

Quicksort 5.6i creates the items before sorting. During the sorting process it uses a stack to store the current start and end positions, thus the small increase in memory during sorting.

Sieve 5.6k constantly computes new values and adds them to the temporary state thus the constant increase in the memory consumption.

Updateset 5.6m has an almost constant memory consumption because each step produces the exact same number of updates. Thus only the first step has to allocate memory, all subsequent steps can just reuse it.

When comparing `casmi` to `casmi-legacy` the average peak memory usage for these benchmarks has decreased by about 99,08%. This is a clear improvement over the old one which in turn enables us to execute industrial-scale CASM specifications in `casmi`.

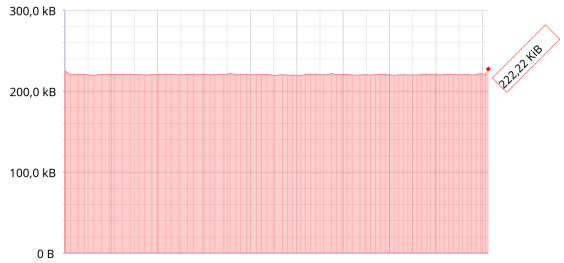
All benchmarks also show that the memory consumption of `casmi-legacy` is constantly increasing and one benchmark even runs out of memory pretty soon. Please note that the interpreter can not make use of the whole 16GB memory while running under massif, because massif keeps the heap snapshots in memory. Thus the updateset benchmark runs out of memory while running under massif but works fine when running without massif.

[2] `http://valgrind.org/`

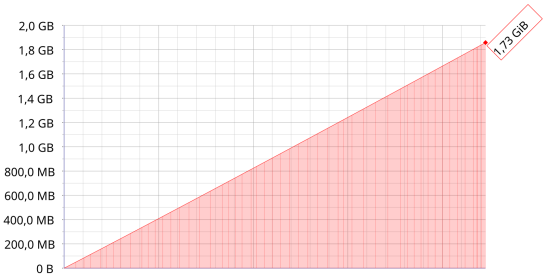
[3] `https://cgit.kde.org/massif-visualizer.git`

Benchmark	`casmi`	`casmi-legacy`	Consumption Decrease
Bubblesort	222.22 KiB	1.73 GiB	99.988%
Conway	275.01 KiB	1.07 GiB	99.975%
Fibonacci	6.20 MiB	5.62 GiB	99.892%
Gray	317.89 KiB	2.54 GiB	99.988%
Quicksort	352.95 KiB	30.02 MiB	98.852%
Sieve	5.00 MiB	117.89 MiB	95.759%
Updateset	581.39 KiB	Out of memory	-

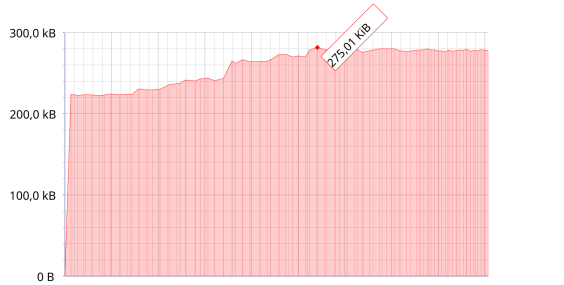
Table 5.5: Peak Memory Consumption



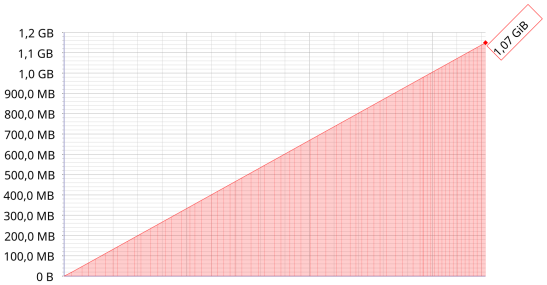
(a) Bubblesort (`casmi`)



(b) Bubblesort (`casmi-legacy`)



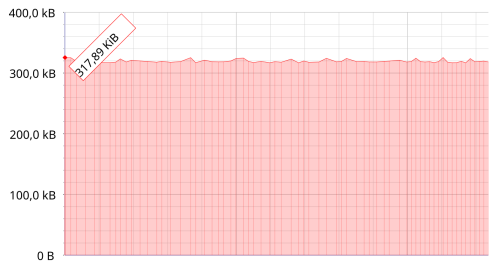
(c) Conway (`casmi`)



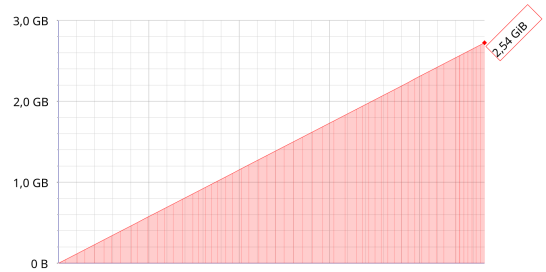
(d) Conway (`casmi-legacy`)

(e) Fibonacci (`casmi`)

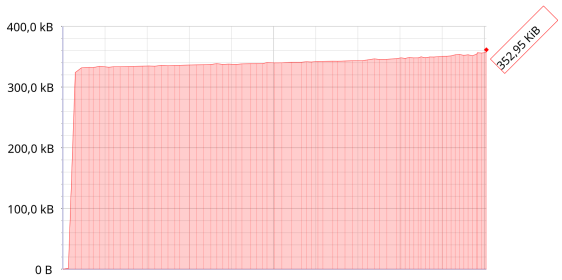
(f) Fibonacci (`casmi-legacy`)



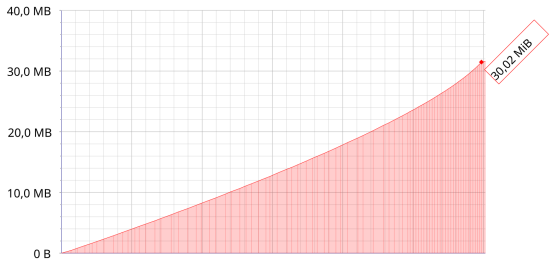
(g) Gray (`casmi`)



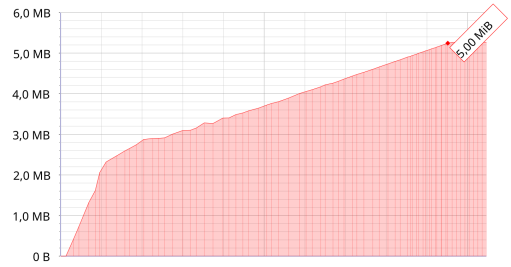
(h) Gray (`casmi-legacy`)



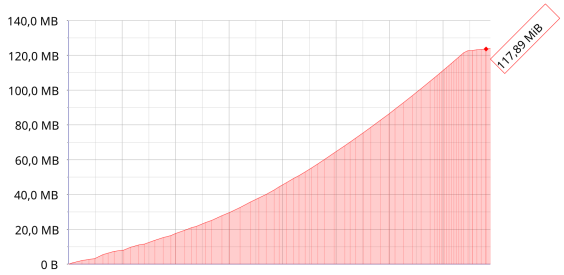
(i) Quicksort (`casmi`)



(j) Quicksort (`casmi-legacy`)



(k) Sieve (`casmi`)



(l) Sieve (`casmi-legacy`)

(m) Updateset (`casmi`)

Table 5.6: Heap + Stack Memory Profiles (`casmi` vs. `casmi-legacy`)

5.5 Arguments Hashing

The different hashing algorithms are compared by using stencil codes. Stencil codes were chosen because they require iterative updating of cells, which results in a huge amount of reads and writes to different cells and because they can easily be extended to multi-dimensional problem spaces. This makes it perfectly suitable to test the different multi-dimensional arguments hashing approaches, the number of dimensions directly corresponds to the number of arguments of the *data* and *kernel* functions. Each dimension of *data* function has the length five and each dimension of *kernel* function has the length three. This means that *data* contains exactly 5^d and *kernel* exactly 3^d cells, where d denotes the number of dimensions. The stencil operation is applied to each *data* cell exactly once.

Dimensions	FNV-1A	Murmur3	SipHash	Legacy
2	2.25 ms	2.38 ms	2.28 ms	2.30 ms
3	3.26 ms	3.14 ms	3.13 ms	3.59 ms
4	20.78 ms	19.00 ms	19.37 ms	44.19 ms
5	324.30 ms	285.97 ms	293.59 ms	2,294.58 ms
6	5,437.64 ms	4,659.83 ms	4,857.01 ms	167,112.40 ms

Table 5.7: Execution Times of Different Hashing Algorithms

Table 5.7 shows the minimum execution times out of at least five rounds. The *FNV-1A* algorithm is the fastest one for the two-dimensional stencil problem, but with growing number of dimensions it clearly loses the competition. *Murmur3* is the fastest one for the six-dimensional stencil problem and also has the best overall performance. As the authors of *SipHash* already stated in [AB12], *SipHash* performs quite well compared to the other hashing functions while providing stronger security goals. The benchmarks also clearly show that the legacy approach didn't scale well with a growing number of function arguments, although it's the fastest for zero- and one-dimensional functions.

The *Murmur3* algorithm has been selected as the default hashing algorithm for `casmi`, both for 32- and 64-bit architectures. It may be worth to check out the Murmur inspired CityHash [PA] and its successor FarmHash [Pik], both promise better performance by making use of modern CPU features.

5.6 Hash Maps

The maximum probe sequence length (PSL) of the hash map search operation has been measured using different types of hash maps. Each benchmark was repeated 25 times and the maximum number of all runs was selected. A maximum load factor of 1.0 was used for *chained hashing* and 0.5 for the other hash map types. Table 5.8 shows the maximum PSL of hash maps used to store the updates of an update set. Table 5.10 shows the maximum PSL of hash maps used to hold the function states. In addition to the maximum PSL some other interesting hash map related numbers were collected. Table 5.9 shows the number of reads, failed reads and writes of update set hash maps, Table 5.11 shows the same numbers for function state hash maps. Please note that the number of reads and writes is independent from the selected hash map type and load factor.

Table 5.8 and Table 5.10 show that the maximum PSL of the *Updateset* benchmark is always 0. The reason for that is that this benchmark contains only three 0-ary functions, resulting in collision free inserts and lookups.

As seen in Table 5.9 *Conway*, *Bubblesort* and the *Stencil* benchmarks never perform any lookups in the update set hash map. The reason for that is that these benchmarks are

Benchmark	Linear Probing	Quadratic Probing	RH Hashing	Chained Hashing
Bubblesort	1	1	2	0
Conway	24	10	3	0
Fibonacci	580	35	157	2
Gray	0	0	0	1
Quicksort	3	3	2	0
Sieve	791	37	238	13
Updateset	0	0	0	0
Stencil2	1	1	1	0
Stencil3	1	2	1	0
Stencil4	4	7	2	0
Stencil5	316	28	31	0
Stencil6	693	36	244	0
Average	201	13	56.8	1.3

Table 5.8: Maximum Search Probe Sequence Length (update set)

Benchmark	Reads	Failed Reads	Writes	Failed Reads (%)
Bubblesort	0	0	4,141,009	0.00
Conway	0	0	3,530	0.00
Fibonacci	67,488	37,490	11,253	55.55
Gray	8,454,144	2,162,688	5,308,420	25.58
Quicksort	275,998	163,348	95,110	59.18
Sieve	270,313	74,065	194,820	27.40
Updateset	54,814,467	246,912	33,456,587	0.45
Stencil2	0	0	27	0.00
Stencil3	0	0	127	0.00
Stencil4	0	0	627	0.00
Stencil5	0	0	3,127	0.00
Stencil6	0	0	15,627	0.00
Average				14.01

Table 5.9: Hash Map Read/Write Statistics (update set)

only using parallel execution semantics and thus function values are only retrieved from the global state but never from the temporary states (see Definition 3.1.8).

Table 5.11 clearly shows that the function state hash maps are read dominated. This is because each function update and lookup always involves exactly one lookup in the function state (see Chapter 3.4). Writing into a function state only happens the first time a specific location is updated, all subsequent updates of the same location only perform

Benchmark	Linear Probing	Quadratic Probing	RH Hashing	Chained Hashing
Bubblesort	10	6	4	3
Conway	21	14	6	3
Fibonacci	27	11	6	2
Gray	1	1	1	1
Quicksort	10	7	4	2
Sieve	27	12	6	5
Updateset	0	0	0	0
Stencil2	2	2	2	0
Stencil3	5	3	2	1
Stencil4	16	9	5	3
Stencil5	27	15	8	4
Stencil6	39	17	12	5
Average	15.4	8.1	4.7	2

Table 5.10: Maximum Search Probe Sequence Length (function state)

Benchmark	Reads	Failed Reads	Writes	Failed Reads (%)
Bubblesort	29,796,713	204	204	0.00
Conway	4,949,365	4,799,189	471	96.97
Fibonacci	45,003	15,003	7,502	33.34
Gray	11,730,950	50	50	0.00
Quicksort	335,408	612	312	0.18
Sieve	280,219	10,004	10,004	3.57
Updateset	87,653,770	5	5	0.00
Stencil2	478	404	26	84.52
Stencil3	6,878	5,840	126	84.91
Stencil4	101,878	87,908	626	86.29
Stencil5	1,521,878	1,337,792	3,126	87.90
Stencil6	22,796,878	20,391,284	15,626	89.45
Average				47.26

Table 5.11: Hash Map Read/Write Statistics (function state)

a function state lookup. This behavior is denoted as *branding* (see [Pau14]). Firing of updates doesn't involve any writes into the function state hash map because an update already knows the exact memory location of the function state value which needs to be overridden, thus the number of writes is relatively low compared to the number of reads.

As seen in Table 5.8 and Table 5.10 *chained hashing* has the smallest maximum PSL in both scenarios. When using an open addressing hash table, then *quadratic probing* has

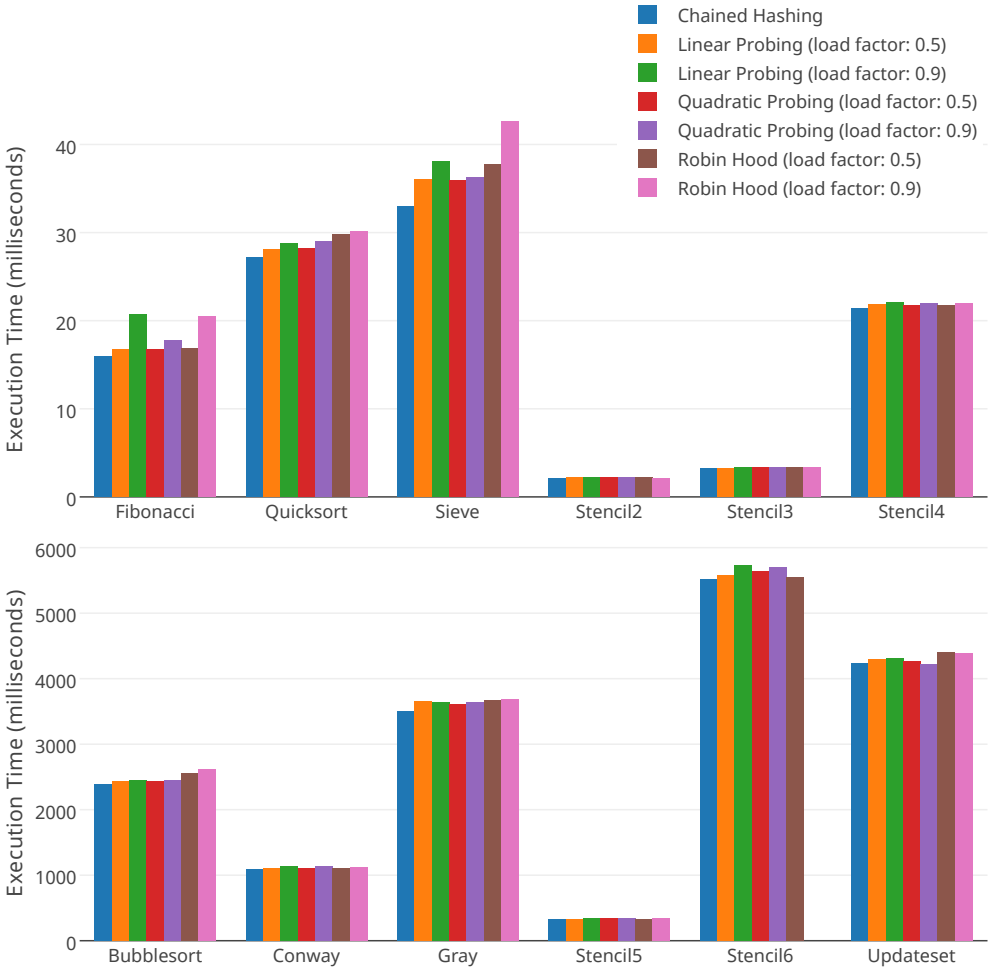


Figure 5.1: Execution Times of Different Hash Maps under Different Load Factors

the smallest maximum PLS in the update set scenario and *robin hood hashing* has the smallest maximum PLS in the function state scenario.

Figure 5.1 shows the execution times of different hash maps using different load factors. The maximum load factor of the hash map using chained hashing is always 1.0, the other hash map types were benchmarked using a maximum load of 50% and 90%. *Chained hashing* is always the fastest one when the number of failed reads of updates is above 20%. All in all the performance of different hash map types are pretty close together and no clear winner could be selected. For these relatively small CASM specifications the *chained hashing* approach is the best one and thus this hash map type has been selected as the default one for the time being. It would definitely be interesting to repeat those benchmarks in future once the IR interpreter is ready and bigger CASM specifications, if possible also from real-world projects, are available.

CHAPTER 6

Conclusion

This thesis presents an improved version of `casmi` based on the work done by Hahn as part of his bachelor thesis [Hah14], Paulweber as part of his master thesis [Pau14] and Lezuo as part of his PhD thesis [Lez14]. Missing proprietary parts have been re-implemented in order to provide `casmi` as free and open source CASM interpreter and a lot of different areas have received significant improvements and clean-ups.

The new update set implementation was one of the major goals of this thesis. It provides a fully tested and reusable component, which can later be reused in the IR interpreter [PZ16]. The design of this component makes it easy to understand and straightforward to use. Furthermore, a set of rules have been shown to predict the number of updates of ASM rules. This prediction can be used to reduce the amount of rehashing in update sets, which occurs when more updates are inserted into the update set than slots were previously reserved.

The overall memory consumption of `casmi` has been drastically reduced as seen in Section 5.4. `casmi` is now able to run industrial-scale CASM specifications with only a relatively small memory footprint, whereas the legacy version ran out of memory in a matter of seconds. Additionally the hashing of function arguments has been greatly improved. `casmi` is now able to easily cope with multi-dimensional functions, especially the performance in case of many function arguments has been significantly increased as seen in Section 5.5.

6.1 Future Work

The *update set* size prediction described in section 3.3 could be implemented as an IR optimization pass. This would make it possible to optimize away needless fork and merge operations and to generate optimized byte code for different update set sizes (array based update set for small numbers vs. hash map based update set for larger numbers).

The predicted number of updates can also be used to reorder the statements of a parallel execution block depending on their expected update set size. Given that the update set of a statement can be merged into its parent update set by simply swapping its internal data structure (see Section 3.4) when the parent update set is empty, the amount rehashing and resizing of update sets could be reduced.

The current value handling of `casmi`, which was implemented as part of the legacy interpreter, still has some memory leaks when using Strings or other non-trivial value objects. This has not be fixed in the current version because the value handling will be replaced by the IR implementation in future, so that builtins and other implementation aspects can be shared more easily between the AST and IR interpreter (see [PZ16] for an architectural overview).

The type checker, which was developed by Hahn as part of his bachelor thesis [Hah14], still has some major problems with type inference. Also *undef* and *symbolic* values are currently un-typed, meaning that the value object loses its original type when assigning *undef* or a symbol to it. This has the drawback that the interpreter treats e.g. an *undef*-Integer the same as an *undef*-String. This can be fixed by replacing the Undef and Symbolic type with boolean values indicating if the value is *undef* or *symbolic*.

Finally, support for first-order logic expressions should be added to `casmi` to make specifications easier to write and understand. CoreASM already has a syntax for existential and universal quantifiers, maybe this syntax can be used in CASM as well. Listing 6.1 shows a railway crossing specification[1] written in CASM. Listing 6.2 shows the same example but with an universal quantifier instead of writing the same logic formula for each member of the Rail domain. The syntax used in Listing 6.2 is equal to the CoreASM syntax.

```
1 derived safeToOpenGate =
2     (status(rail1) = notrain or (now + opendelay) < deadline(rail1)) and
3     (status(rail2) = notrain or (now + opendelay) < deadline(rail2)) and
4     ... and
5     (status(rail3) = notrain or (now + opendelay) < deadline(railn))
```

Listing 6.1: Railway Crossing Example with Current Syntax

```
1 derived safeToOpenGate =
2     forall r in Rail holds
3         status(r) = notrain or (now + opendelay) < deadline(r)
```

Listing 6.2: Railway Crossing Example with Universal Quantifier

[1] https://github.com/CoreASM/coreasm.core/blob/master/org.coreasm.engine/test-rsc/without_test_class/RailroadCrossing.coreasm

Bibliography

[AB12] Jean-Philippe Aumasson and Daniel J. Bernstein. *SipHash: A Fast Short-Input PRF*, pages 489–508. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[App] Austin Appleby. MurmurHash3. `https://github.com/aappleby/smhasher`. [Online; accessed 23-02-2017].

[BS03] Egon Böger and Robert Stark. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003.

[Cel86] Pedro Celis. *Robin Hood Hashing*. PhD thesis, Waterloo, Ont., Canada, Canada, 1986.

[Con70] John Conway. The game of life. *Scientific American*, 223(4):4, 1970.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[FGG07] Roozbeh Farahbod, Uwe Glässer, and Vincenzo Gervasi. CoreASM: An extensible ASM execution engine. *Fundamenta Informaticae*, 77(1-2):71–103, 2007.

[FNV] Glenn Fowler, Landon Curt Noll, and Kiem-Phong Vo. The FNV Non-Cryptographic Hash Algorithm. `https://tools.ietf.org/html/draft-eastlake-fnv-12`. [Online; accessed 31-01-2017].

[Gur95] Yuri Gurevich. Specification and Validation Methods. chapter Evolving Algebras 1993: Lipari Guide, pages 9–36. Oxford University Press, Inc., New York, NY, USA, 1995.

[Hah14] Florian Hahn. *Introducing CASMI, an AST Interpreter for CASM*. 2014. Wien, Techn. Univ., Bac. Thesis.

[Inf13] Dominik Inführ. AST interpreter for CASM. 2013. Wien, Techn. Univ., Bac. Thesis.

[LBK13] Roland Lezuo, Gergö Barany, and Andreas Krall. CASM: Implementing an Abstract State Machine based Programming Language. In *Software Engineering 2013 - Workshopband (inkl. Doktorandensymposium), Fachtagung des GI-Fachbereichs Softwaretechnik, Aachen, Germany, February 26 - March 1, 2013*, pages 75–90, 2013.

[Lez14] Roland Lezuo. *Scalable translation validation; tools, techniques and framework.* 2014. Wien, Techn. Univ., Diss.

[LK12] Roland Lezuo and Andreas Krall. *A Unified Processor Model for Compiler Verification and Simulation Using ASM*, pages 327–330. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[LPK14] Roland Lezuo, Philipp Paulweber, and Andreas Krall. CASM: Optimized Compilation of Abstract State Machines. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '14, pages 13–22, New York, NY, USA, 2014. ACM.

[PA] Geoff Pike and Jyrki Alakuijala. CityHash, a family of hash functions for strings. `https://github.com/google/cityhash`. [Online; accessed 23-02-2017].

[Pau14] Philipp Paulweber. *An optimizing compiler for the abstract state machine language CASM.* 2014. Wien, Techn. Univ., Dipl.-Arb.

[Pik] Geoff Pike. FarmHash, a family of hash functions. `https://github.com/google/farmhash`. [Online; accessed 23-02-2017].

[PZ16] Philipp Paulweber and Uwe Zdun. A Model-Based Transformation Approach to Reuse and Retarget CASM Specifications. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 5th International Conference, ABZ 2016*, Lecture Notes in Computer Science 9675, pages 250–255, May 2016.

[RAD15] Stefan Richter, Victor Alvarez, and Jens Dittrich. A Seven-dimensional Analysis of Hashing Methods and Its Implications on Query Processing. *Proc. VLDB Endow.*, 9(3):96–107, November 2015.

[Sch01] Joachim Schmid. Compiling Abstract State Machines to C++. *Journal of Universal Computer Science*, 7(11):1068–1087, nov 2001.