Efficient and Proven Verification of Unreliable Hardware Executions of Classic Algorithms

by

Yoana G. Gyurova

S.B., Massachusetts Institute of Technology (2014)

Submitted to the
Department of Electrical Engineering and Computer Science in
Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

June 2015

Copyright 2015 Yoana Gyurova. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole and in part in any medium now known or hereafter created.

Author_____
Department of Electrical Engineering and Computer Science
May 26, 2015

Certified by_____
Saman Amarasinghe, Professor, Thesis Supervisor
May 26, 2015

Accepted by_____
Prof. Albert R. Meyer, Chairman, Masters of Engineering Thesis Committee

Efficient and Proven Verification of Unreliable Hardware Executions of Classic Algorithms

by Yoana G. Gyurova
Submitted to the Department of Electrical Engineering and Computer Science

May 26, 2015
In Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

Lowering voltage and frequency guardbands of CPU, DRAM, cache, or interconnect lowers power and latency, but increases the risk of silent data corruptions in even formally verified hardware and software.
Researchers have been developing systems that use unreliable hardware, combined with software checkers executed on reliable hardware, to gain high performance with no risk.
Deterministic checkers for many important algorithms are asymptotically and practically more efficient than the original problem solvers, e.g. for systems of linear equations, satisfiability, linear programming, sorting, 3SUM, graph matching and others.
Writing a correct checker is hard, since often intricate corner cases get overlooked. Our system, SoundCheck, helps reduce burden on programmers by automatically proving soundness and completeness of checkers with bounded verification in the Sketch program synthesis language. Verified checkers are emitted as efficient C++ code and shown to have low overhead which results in a net performance improvement with no risk.

Thesis Supervisor: Saman Amarasinghe
Title: Professor

Acknowledgments

I would like to sincerely thank my mentor, Vladimir Kiriansky, for his support and guidance throughout my experience as a Masters student. He was always there to provide advice and to help me through the learning process. I am deeply grateful for his patience and his willingness to share his knowledge with me. When I joined the research group, he was already researching on systems combining unreliable hardware and reliable software checkers, explained in detail in Section 3. He implemented the guided solver routine, showed in Section 3.2. He also had substantial contribution in the implementation and proof of some algorithms we evaluated in this research, mostly SAT, systems of linear equations and linear programming. He found the hardware errors described in Section 6. My mentor and I jointly worked on a paper submission for a conference and an extended version of the paper turned into this thesis.

I would also like to express my deepest gratitude to my thesis supervisor, Saman Amarasinghe, for giving me directions and providing me with a clear idea of the project scope and difficulties. He was always available to discuss and help with any issues that I came across at all stages of the project. His excitement about this research and the results we obtained made me motivated to complete my part of it to the best of my ability.

I also give thanks to Shoaib Kamil, Rohit Singh, and Armando Solar-Lezama for their guidance in using Sketch, and Ivan Kuraj for helping with our exploration of Leon.

Last but not least, I cannot be more grateful to my family for their constant encouragement and belief in me. The infinite support and love I received from them kept me going trough good and bad times and it was truly invaluable to me.

Chapter 1

Introduction

We demonstrate we can construct efficient and reliable checkers for programs executed on non-trusted or unreliable machines and automatically prove them correct.

Voltage and frequency guardbands added to hardware designs are predicted to reduce performance by 50% at 10nm [1, 2]. This performance loss can be regained by intentionally overclocking machines beyond hardware specifications, which adds the difficulty of maintaining reliability of the program results.

Today, it is possible to have unreliable hardware that is more efficient and faster than reliable hardware. While executing on unreliable hardware can give better performance, it can lead to unreliable results, due to the introduction of silent errors. Our goal is to show that for many of the most important algorithms, reliability guarantees can be provided by validating the results from unreliable executions with efficient software *checkers*, executed on separate reliable machines. Checkers should be faster in practice compared to the checked programs to maximize net gain. In many cases, checking the final result or intermediate results is much faster than producing them. We take advantage of this runtime asymmetry by running unreliably the target program while a slower reliable machine can check the result. In this thesis, we use the term *solver* to refer to the target program executed on unreliable machine. The

main algorithms we evaluate are systems of linear equations, linear programming, and satisfiability problems, sorting, graph matching and 3SUM.

In this thesis work, we focus on helping programmers create correct deterministic checkers and maintain checkers correctness over multiple optimization iterations. The most important requirement for a checker is to be sound, i.e. it must detect any errors produced by an unreliable execution of the target program. A checker should also be complete, i.e. it should accept every possible correct output. An efficient checker may accept even more outputs for a given input than those produced by a given solver as long as they are valid. Proving soundness for such checkers, however, is more difficult. We demonstrate checkers and novel proof strategies for checker soundness for several important solvers. We use the synthesis language Sketch [3] to prove checkers correct, and to emit efficient C++ code from the same source code.

To illustrate that writing a good checker is not trivial, consider sorting. The first property that most programmers consider to check is whether the output is sorted in the specified order. They rarely think of the second property of sorting – the permutation property. For example, for an unsorted input $\langle 1, 5, 2 \rangle$ a checker should accept only $\langle 1, 2, 5 \rangle$ and it must reject an output like $\langle 1, 2, 4 \rangle$. Even though this output is sorted and has the same length as the input, the element 5 from the original array got corrupted to 4 after a bit flip. A sound checker has to make sure that the numbers in the output are a permutation of the numbers in the input. A faster checker might check only if each element in the output is in the input but this checker is not correct as it would accept the invalid outputs $\langle 1, 2, 2 \rangle$, $\langle 1, 5 \rangle$ and some others. Even examples in formal method tools show postconditions that only verify that the input elements as a set match the output elements set, instead of correctly using a multiset to count all occurrences of input elements. We will discuss in Section 5.6 details of our sound and efficient sorting checker, and show in Section 6 that the majority of errors we observed are bad permutations.

We implement a prover to automatically verify a checker is sound and complete in relation to a *reference solver*. The reference solver may be the target program that will be executed unreliably, or a simpler implementation that is a *de facto* specification to a given problem. The reference solver produces the expected output for every input, though it may be much less efficient than the target solver. For example, a reference solver may be a textbook implementation of merge sort, instead of its TimSort variant which only recently was shown amenable to formal analysis [4].

Our runtime checkers can also guarantee correctness of unreliable software, or software that is beyond the capabilities of current formal method tools. Better performance can be achieved with more complex algorithms or data structures, or parallel or distributed optimized implementations, or optimizing compilers and language runtimes. If these more efficient alternative solvers are expected to produce outputs equivalent to the reference solver, a runtime checker can guarantee they are correct.

We discuss related work in Section 2, our system overview and runtime model in Section 3, soundness and completeness proof strategies in Section 4, algorithm specific challenges in Section 5, evaluation of checker effectiveness in detecting hardware errors, and prover effectiveness in detecting checker bugs in Section 6, and conclusion in Section 7.

Chapter 2

Related Work

Previous systems using unreliable hardware or software make a trade-off between fast execution and reliability. SoundCheck provides both with a formal guarantee.

Program checking for many important problems has been investigated over the years [5], though often as "a theoretical curiosity" [6]. For some applications probabilistic guarantees are sufficient, for example, a sorting checker may check the permutation property in $O(n)$ by comparing the sum of input with sum of output [7]. Our sorting checkers are also linear but deterministic.

Slow checkers, even much slower than the original program are acceptable during software development. In our system we are only interested in solvers and checkers with low overhead to offer performance gains in production. For example, we only look at SAT but we are not addressing UNSAT instances. While SAT satisfiability solvers can produce proofs of unsatisfiability that can be checked, e.g. with resolution or reverse unit propagation proofs, often either the augmented solver or the proof-checker is slower than the original solver. UNSAT checkers have been proposed with low SAT solver overhead[8], and even formally proven [9] albeit slow.

Formal methods have been used with sort algorithms starting with Quicksort[10], up to very recent successful application of formal methods to unroot a bug in Open-

JDK's TimSort [4]. Precondition and postcondition assertions that hold on reliable hardware, however, may not hold on unreliable hardware. Therefore, traditional formal method proof invariants about software executing on unreliable hardware cannot be based on Hoare logic[11].

The problem of verifying checker correctness maps closely to the problem of automated equivalence checking [12]. Yet checkers are not equivalent to solvers, and proving soundness is harder since for a given problem an efficient checker may accept outputs that are valid but not produced by the target program. In SoundCheck we build a system and develop strategies for proving checkers sound and complete.

Hardware designs, like Diva [13], pair unreliable CPUs with simpler more reliable CPUs and compare the results on each for every executed instruction. Our checkers incur much less overhead, especially when only final results are checked.

This thesis extends the work in [14], with the added guarantees for soundness and completeness of the checkers.

Chapter 3

System Overview

Figure 3-1 shows the runtime components of our system. We call a *solver* the original program, while a *checker* is a program that dynamically checks whether an output produced by the solver is acceptable.

An input is given to both the solver and the checker. Since the solver is executed unreliably its output may be incorrect. The checker will accept all valid outputs for the given problem instance and reject all invalid ones. In case an output is rejected, the solver will have to be re-executed reliably.

3.1 Runtime system

3.1.1 Unreliable and Reliable Execution

Execution on unreliable nodes offers advantages over reliable nodes in performance, power, and cost. Our unreliable nodes use hardware configured beyond specifications – e.g. overclocked CPUs, caches, memory, or interconnect networks.

Reliable nodes must not be affected by errors on unreliable nodes. Our reliable and unreliable nodes run on separate machines communicating over a network.

Depending on the checker performance advantage over the solver, a single slower

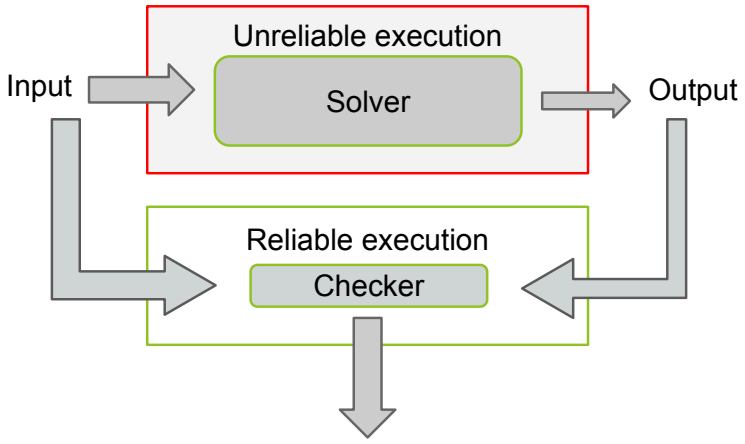

Figure 3-1: Runtime system

reliable node can check the results of multiple unreliable nodes. Our sorting checker has only $O(\log n)$ advantage over solver, therefore constant factors in runtime are critical for overall efficiency. For all other algorithms, checker to solver advantage on practical sized problems is $10^3 - 10^6\times$. Table 3.1 summarizes the performance advantage of checkers over typical solvers for each problem, expressed as ratios of running time of the problem size in asymptotic terms, and as actual time ratios for practical instances.

Problem	**Asymptotic Ratio**	**Practical Size Ratio**
LINPACK	$O(n)$	$10^4\times$
SAT	$O(2^n/n)$	$10^5\times$
LP	$O(2^n/n)$	$10^6\times$
Perfect Matching	$O(E/\sqrt{V})$	$10^5\times$
Maximum Matching	$O(\sqrt{V})$	$10^3\times$
Sort	$O(\log n)$	$35\times$

Table 3.1: Ratio of solver vs checker runtime on practical instances.

3.1.2 The solver – the original program

We target only deterministic algorithm solvers for both decision and optimization problems. Decision problems are only concerned with feasibility of a given solution, e.g. SAT. Proving non-feasibility is often harder, e.g. UNSAT.

Proving optimality may be impossible for some problems e.g. MaxSAT. With algorithm knowledge, however, problems in which local optimality guarantees global optimality can be efficiently checked – e.g. convex problems like linear programming, maximum matching, least squares, etc.

3.1.3 The checker

To catch solver errors due to unreliable execution, we implement a solver-specific checker. This checker runs reliably, therefore we are only concerned about algorithm and software implementation errors.

The theoretical groundwork proving the original solver sound, complete, and terminating can often be reused to understand how to write a checker to determine if a result is feasible and optimal. For many iterative algorithms, a checker needs a single iteration of the original algorithm to ascertain optimality after starting with an unreliable solution as a hint.

High-level algorithm insights depend on human creativity, but low level challenges of software engineering can be addressed by automatic tools. Writing checkers from scratch is challenging, and even if portions of solver code are reused, this may break poorly documented or partially understood invariants. SoundCheck helps automatically determine if both the algorithm and implementation of a checker are correct.

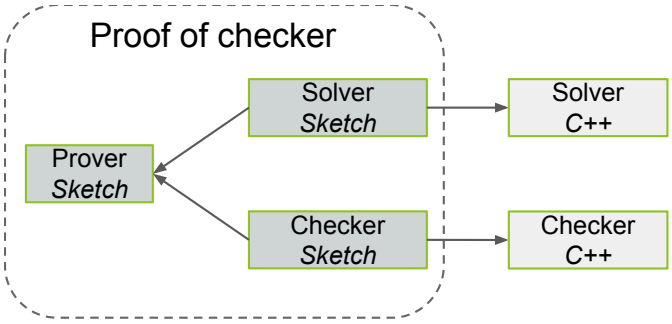

Figure 3-2: Automatic proof and efficient code generation from Sketch sources

3.2 SoundCheck offline system

3.2.1 Checker Soundness and Completeness

A checker *must* be *sound* – for a given input it must never accept an invalid output, and it *should* be *complete* with respect to the original solver – it should accept all valid outputs. A checker may be incomplete if some valid outputs are too hard to check, but this will only affect performance since the system will fall back to re-executing the solver on a reliable node. We focus on checkers that are both sound and complete, and we verify these properties with our *prover*.

3.2.2 Prover

The SoundCheck prover is an offline component that automatically proves soundness and completeness for a given checker with respect to a reference solver. Once a checker is validated by SoundCheck, an unreliable solver paired with it will have soundness properties equivalent to a reliable solver. A checker can then be emitted to efficient code for use in the runtime system.

Figure 3-2 shows the offline SoundCheck system. In our current implementation its three components – solver, checker, and prover are all written in Sketch [3]. After a successful proof Sketch allows us to emit a checker in efficient C++ without further

manual intervention.

The prover must ascertain that for all ⟨input, output⟩ pairs a checker rejects all invalid pairs for soundness and accepts all valid pairs for completeness. Valid ⟨input, output⟩ pairs are only those for which the output may be produced by a reliably executed solver given the input. A straightforward prover can exhaustively search the universe of all pairs for a counterexample. In Section 4 we will replace Algorithm 1 with more efficient verification tools.

Algorithm 1 Exhaustive "prover" of checker soundness and completeness

```
1 procedure ExhaustiveProver(f, check)
2     for ∀in ∀out do
3         Prove(in, out, f, check)
4     end for
5 end procedure
```

The crux of proving soundness is in the Prove routine. It must use different proof strategies depending on the number of valid outputs for every input.

3.2.3 Unique Valid Outputs

When for every input there is at most one output that should be accepted by the checker, e.g. stable sort, Algorithm 2 directly follows from the definitions of soundness and completeness. The prover asserts that only the output produced by a reliable reference solver $f(in)$ is accepted, and any other output is rejected.

Algorithm 2 Prover of checker correctness for problems with a single valid output per input

```
1 procedure ProveUnique(in, out, f, check)
2     if f(in) = out then
3         assert check(in, out)                    ▷ Completeness
4     else
5         assert not check(in, out)                ▷ Soundness
6     end if
7 end procedure
```

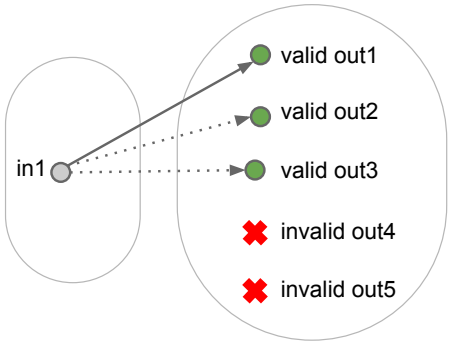

Figure 3-3: Canonical output, alternative valid outputs and invalid outputs

3.2.4 Multiple Valid Outputs with Guided Solver

In Figure 3-3, we show the more general case when for a given input of the original problem there may be multiple valid outputs. We assume a deterministic reference solver, and we will call its output the *canonical valid output*. In addition to this output, *alternative valid outputs* may exist for this problem. Consider the graph perfect matching problem: even though a deterministic solver will return only one matching, alternative solutions may exist. All other outputs are *invalid*.

If a checker accepts the alternative valid outputs, then Algorithm 2 can not prove its soundness. If alternative valid outputs are rejected, then the checker would also not be complete with respect to other solvers.

Artificially constraining the problem so that solvers produce a unique result can be easily proven sound but may dramatically reduce solver or checker efficiency – e.g. if a solver must give the first output in lexicographic order, a checker may have to enumerate all solutions to accept only the first.

For a more efficient runtime, we may require the reference solver to generate the set of all valid outputs instead of only the canonical one to the prover.

The approach we took and found effective for our provers is to create a minimal modification to the reference solver – a *guided* version, that given a possible output, makes any non-binding decisions such that a desired alternative output is reached

instead of the canonical. A guided solver given an invalid output will return a valid output instead.

For example, a guided `for` loop may be traversed in a different order – a desired index may be processed first or last, or other indices may not need to be processed. Here we show the one line change needed to guide a routine from a graph matching solver. Instead of extending a matching with the first feasible vertex, it is extended with the vertex from the desired matching.

```
1 extend(Vertex u, Vertex[] match)
2   for (int v = 0; v < N; v++)
3     if (guided() && guided_match[v]==u)
```

We call the above loop transformation a **guided_for** that can be used to mark loops with non-deterministic choice. Iterations can be reordered or skipped when any feasible solution is acceptable, e.g. perfect matching. Only reordering is allowed when optimality is needed, e.g. for maximum matching, as skipping loop iterations may force the solver to ignore the best solution. With a *guided solver* we can now reach all alternative outputs.

Algorithm 3 Prover of checker correctness with guided deterministic choice in solver

```
 1 procedure ProveGuided(in, out, f, check)
                          ▷ Validate checker(in, out) with f guided towards out
 2     if f(in) = out then
 3         assert check(in, out)                                  ▷ Completeness
 4     end if
 5     if guided(f)(in, out) = out then
 6         assert check(in, out)                          ▷ Extended Completeness
 7     else
 8         assert not check(in, out)                              ▷ Soundness
 9     end if
10 end procedure
```

In Algorithm 3 we use a guided solver (line 5) which allows us to prove soundness (line 8) which was impossible before. If checker completeness should be extended

in relation to all other solvers of the same problem, line 6 asserts that alternative outputs are accepted.

The prover does not have the specifications of the solver, but it can execute it on any input.

In general, we cannot have a checker for any sort algorithm, we talk about a checker that checks non-stable sort which might not be good for stable sort since it will not necessarily check stability.

If, we somehow know for sure the original program is correct implementation of sort, then a checker can be used to check other sort programs with the same specifications.

Chapter 4

Verification Implementation

The SoundCheck assertions can be proved by using existing tools for bounded or unbounded verification. We present the tools we considered, together with summary of advantages and disadvantages each one had for our use case. We decided on using Sketch [3] for bounded verification after comparing to other tools such as Java and Coq. We give an analysis of differences between Java and Sketch and compare the two based on running time of the prover for an implementation of merge sort.

In this thesis, we also present findings on the system Leon[15], which we hoped could extend our research to also cover unbounded verification. However, due to some limitations of the system, Leon was not able to give formal guarantees of soundness and completeness of any checkers, especially when we knew they were implemented correctly.

4.1 Bounded verification with Sketch

Sketch [3] is a program synthesis tool which allows programmers to write code with missing parts that the Sketch synthesizer can fill in to satisfy all assertions.

Even though we do not use the full functionality of the synthesiser, we take advantage of the fact that Sketch is essentially a SAT solver on the backend and this

underlying mechanism allows for some verification problems to be solved in a fast manner. In our research we take advantage of the efficiency with which the tool can prove or disprove assertions in large spaces of possible inputs. This is crucial property for us because we need a fast and deterministic way to run the prover, which takes too long to complete in Java even for small inputs.

The usage of Sketch involves placing assertions about the result from the checker in the prover code and based on whether or not these assertions are satisfied concluding if the checker is correct or not. The Sketch underlying logic will check the assertions for the universe of all possible inputs to the prover which are the possible ⟨ input, output ⟩ pairs to the original program. These assertions are written within *assert* statements and all the assert statements go into a *harness* function which is executed when the prover is called. Whenever some assertion cannot be satisfied because the function can find a counterexample, Sketch prints the message UNSATISFIABLE ASSERTION together with the place in the program where the assertion occurs. In addition, Sketch can provide the counterexample that caused the unsatisfiability of the assertion which makes it easier to debug and fix the issue.

In our scenario, we have two assertions, one proving the soundness and one - the completeness of the checker. The Sketch prover determines which property is not satisfied by displaying which of the assertions were not satisfied during the execution. It is of course possible that the checker is neither sound nor complete in which case the assertion for which a counterexample is found first will break and cause the prover to stop.

4.1.1 Prover in Sketch

We envision checker writers can take advantage of synthesis and write checker sketches, but we primarily depend on Sketch to replace the naïve Algorithm 1. Its integrated SAT solver back-end and verification system can disprove assertions by providing

counter-examples, or prove all assertions in large input spaces efficiently.

The prover **assert** statements and pseudocode directly map to Sketch code. We need the checker (after optional synthesis) and a reference solver (with no preconditions or postconditions required) as Sketch functions. Whenever an assertion cannot be satisfied, Sketch offers a counter-example showing when a checker is not sound or not complete. This is usually a bug in the checker, if the prover and the reference solver are expected to be correct.

4.1.2 Sketch Limitations

The main limitation of Sketch is that it guarantees correctness only up to a fixed input size. The solver, checker, and prover may all be incorrect on larger inputs.

Interactive Verification

There are Sketch back-end options to control input bit range, intermediate values range, function inlining and loop unrolling. Reducing input ranges and intermediate values improves proof times, however, correctly setting the maximum intermediate value range depends on the target solver. For some solvers, the limits on bounded function inlining, recursion, and loop unrolling also need to be scaled to handle larger input range and size.

Increasing Coverage

We explicitly control input sizes in our test harness independently of input bit range. This allows us to explore separately each dimension - i.e. small input sizes with large value range, vs large input size with small range. Making an informed decision whether a bounded proof will be correct for larger input sizes and ranges is still an art.

When running the prover in our system, we restrict the size of the input in addition

to the range of possible values. Sketch requires certain flags to be set to bound the space the underlying solver searches in when running the prover. In our work, we mainly define three different flags.

In the next paragraphs, we explain in detail the significance and the syntax for each flag that we needed to set while using Sketch.

Input bits bound

By setting the flag –**bnd-inbits numBits** in Sketch, the solver searches through values of the input and output that are up to $2^{numBits}$-1.

Input size limitations

Even if we set the number of bits to a very small number, increasing the size of the input will make Sketch break or timeout. For example, when doing sorting, we can set the array size to 4 and try increasing the range of the numbers we are sorting by increasing the inbits flag. We can also do the opposite, set the range of the integers and increase how many numbers we are sorting until we get to a point where the prover takes too long to come up with a result.

Intermediate value range

In some scenarios, when many computations are involved in the original program or the checker the sketch backend solver assumes the execution will produce very large intermediate values and that is why if left by default it searches a very big range. When explicitly using the flag **-bnd-int-range interRange** we force the solver to consider executions where the intermediate values can go up to interRange and ignore all other branches where bigger values are assumed to appear. This way we are essentially helping Sketch know what cases not to consider at all.

Function inlining and unrolling

Sketch assumes that certain number of inlining can be made for a function call, or in other words a function can appear up to that many times on the stack. If the code requires a function to be inlined more than the default number of times, the programmer needs to set the flag –**bnd-inline-amnt** to the required value. Even though there is not a formally set limit to how big the amount can be, the increased number makes the prover slower which is normal with recursion in general.

Similarly, Sketch provides a flag for unrolling number which controls loops that the code introduces. If the default is not enough, the programmer needs to set a flag controlling the unrolling called –**bnd-unroll-amnt**.

4.1.3 Proposed Sketch Extensions

Integer Type System

Problems surface once Sketch code is emitted to C++ if variable types are extended to the richer native integer types, e.g. handling corner cases of integer promotion and vagaries of truncation in 32/64-bit signed/unsigned integers.

Our experience with scaling our hashtable-based sort checker to real world sizes up to 64GB illustrates these issues. We were confident that even a bounded proof up to 2^3 should hold up to 2^{30} before intermediate values enter undefined `int` ranges. Cache and memory efficiency required using a combination of 32-bit and 64-bit integers. However, manual modifications of the emitted code to allow this and to handle larger input sizes uncovered more latent problems at each of 2^{31}, 2^{32}, and 2^{33} sizes. Effectively modeling C++ standard defined integer promotion and truncation in Sketch would help detect these boundary conditions using types with independent bit-range control, e.g. a combination of 4-bit signed, and 8-bit unsigned integers. Integrating these in the front-end will allow checkers to use native operators for expressiveness,

and eliminate impact on emitted code performance.

Floating Point

Floating point imprecision and non-associativity bring about further problems to checker completeness, soundness, and soundness proof. Modeling these complexities in fixed point representation in Sketch is difficult.

Floating point imprecision may allow checkers to accept multiple valid outputs, which makes proving soundness difficult. On the other hand, tightening precision to limit a checker to accept a single solution would affect completeness, since a target solver may have lower numerical accuracy.

Non-associativity and floating point truncation bring up real soundness concerns. For example, using a standard implementation of matrix-vector multiplication $A \times x$ is technically not sound in our checker of systems of linear equations. A proper Kahan summation [16] or pair-wise summation is needed to accumulate small additions. Modeling an IEEE 754-like representation with tiny mantissa and exponent, e.g. "quarter precision" would catch imprecision errors, but is too complex. Alternatively, checkers must add dynamic checks for the assumed number ranges. We are considering a simpler extension to Sketch to statically detect unsound numeric methods while still using fixed point integers. Every floating point variable should be paired with a shadow variable evaluated at a higher precision. If assertion outcomes disagree between comparisons with shadow variables vs comparisons with lower precision variables, we would be able to detect losses of accuracy.

`guided_for`

We have added `guided_for` only as syntactic sugar in the Sketch front-end to mark all iterations where guided choice is acceptable. Manual instrumentation is still needed to add shadow variables for every output and pass-by-reference variable. Sketch

synthesis can be used to determine the conditional expressions to guide iteration choice so that the desired return output is reached.

Parallel Checkers

Many of our checkers are efficiently parallelizable into subproblem checks. Although Sketch has support for concurrent data structures[17], our checkers avoid concurrent writes to minimize contention. Sub-checkers communicate only to signal whether their subproblems are accepted, but we do not prove correctness of this barrier protocol. We only need to prove sequential execution correctness of input partitioning and subpartition checkers.

4.1.4 Java vs. Sketch prover implementations

Deciding what software to use to write the components of our system was not easy. We considered many different options with the main goal being to find a tool that allows writing of fast checkers and proving them correct for as many values as possible. While unbounded verification, which we describe in the next Section, turned out to be even more challenging, bounded verification could use approaches as simple as brute force exhaustive search in the universe of all possible values. We evaluated Java and Sketch to decide which of the two to use for our research.

The Java programming language, being one of the most popular languages and with its support for many data structures, seemed to be a good candidate for writing efficient checkers. In Sketch, we take advantage of the backend solver to get faster execution and search in the universe of ⟨ input, output ⟩ pairs. Since Sketch uses a SAT solver, it can find counterexamples, pairs that are accepted by the checker when not valid or vice versa, faster than Java. In addition, in Sketch, the programmer does not need to generate all possible ⟨ input, output ⟩ pairs since the solver does that automatically. This way we are sure that the checker will be tested on all

pairs. Generating the pairs manually, as Java requires, introduces the additional risk that some special corner cases could be omitted, for example arrays with only equal elements might not be correctly generated and the checker will not be run on those pairs as a result.

Once the prover proves a checker, Sketch allows automatic emission of C++ code. The programmer can then use this C++ for bigger inputs and outputs but Sketch, just like Java, provides no guarantees beyond the input size it verified. While the prover is run only once and its speed is not the main concern, we discovered that the Sketch prover could verify bigger value ranges and hence created a checker with better guarantees than the one produced by Java.

We evaluated logically similar implementations of a prover for merge sort in Sketch and Java. First, we generated all possible arrays of certain pre-specified size and range for the Java implementation of brute-force exhaustive prover. This prover essentially runs Algorithm 1 on all generated pairs aftwerwards. The prover checks if the checker accepts only he valid ones and rejects the invalid ones. This algorithm is exponential on the size of the input array and the running time of the prover quickly grows, limiting the completion of the proof phase to very small inputs.

In Sketch, we run the prover from Algorithm 1. However, Sketch is able to run faster compared to Java for fixed input size and range of the numbers and thus allows us to get verification for more values than Java does. Sketch does not require a generator of all the $\langle$ input, output $\rangle$ pairs since the backend solver does that automatically based on the signature of the harness function and the given inputBits and inputRange parameters.

On Figure 4-1 we compare the time it takes for the Java prover to complete proving a checker, also written in Java. We use an input array of size 4 and compare Java and Sketch for various ranges of the numbers in the input.

The actual running times of the provers in seconds are summarized in Table 4.1.

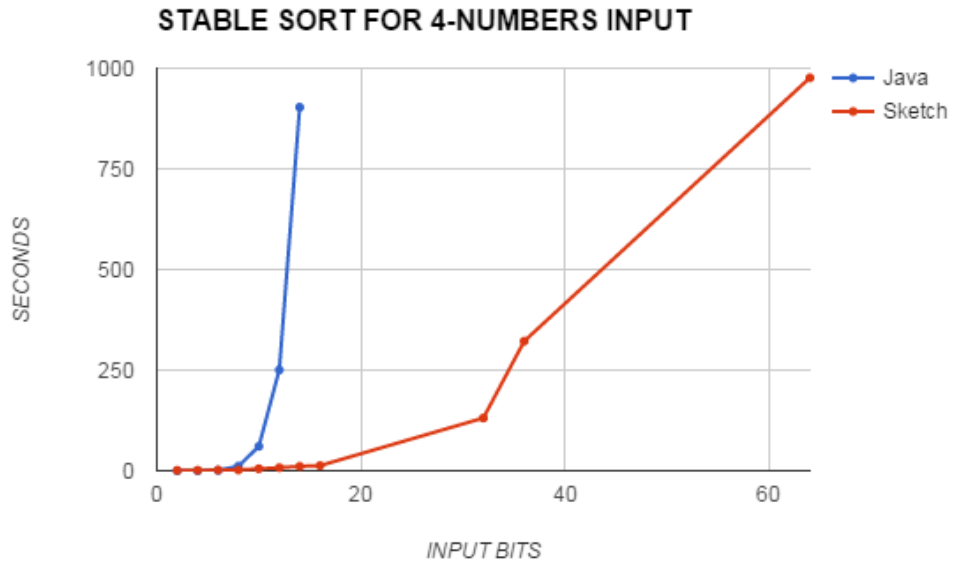

Figure 4-1: Sorting prover running time for input size 4

A range of N means, the prover was run on all possible inputs of size 4 consisting of the numbers in the range from 0 to N. In Java, we successfully prove the checker for 4-bit numbers, while in Sketch we could prove it for 6-bit range in approximately the same amount of time. For more than 4-bit numbers, we do not have information for the prover in Java, since it took too long for the execution to complete. Clearly, even for 4-bit inputs Java becomes slow and practically unusable while Sketch continues to be able to prove the checker correct.

In Table 4.2, we compare the running time of a prover written in Java versus that of a prover written in Sketch for sorting arrays of size 3. We use the same merge sort solver as we did for the previous results and the setup is generally the same. However, having a smaller input array size allows us to get better distinction between Java and Sketch since it takes more bit increases before the prover starts to timeout. In Java we could only verify correctness by the prover for up to 5-bit numbers. In Sketch, we managed to verify inputs of bit size 6 without any issues and we think we could potentially be able to verify even bigger ranges with some optimizations of the solver

Range	Java	Sketch
2	0.01	1.12
4	0.14	1.27
6	1.22	1.9
8	11.44	2.04
10	60.61	5.01
12	250.6	7.55
14	903.18	11
16		12.75
32		131.09
36		321.82
64		976.21

Table 4.1: Stable sort for input of size 4.

and checker code. The difference of 2 bits we expect in favor of the Sketch prover could be actually quite substantial for bounded verification as it extends the set of problems we could verify and the checkers we could trust.

Range	Java /seconds/	Sketch /seconds/
8	0.15	1.259
12	1.15	1.65
16	4.35	2.11
20	17.5	3.99
24	47.9	4.58
28	118.5	8.28
32	223	8.54
36	464	25.5
40	1941	29.1
64		45.95
128		351.64

Table 4.2: Stable sort for input of size 3

In summary, some advantages we found that Sketch has to Java are related to the larger space of $\langle$ input, output$\rangle$ pairs that the prover can search in which the checker can be then considered sound and complete. In Java, the prover for sorting took a lot of time for inputs as small as 4 to 6 bits. While in Sketch the runtime also

increases fast as the range of the input numbers grows, we could prove successfully the checker for inputs between 6 and 8 bits for sorting. In addition, we were able to get the results faster than in Java.

Apart from speed and better coverage of the space of input values, by using Sketch we also take advantage of the simplicity of the prover. As mentioned above, there is no need to generate the possible inputs or outputs since the prover does that automatically. The prover is hence very easy to write and consists of 4 lines of code in the trivial non-guided version. Even the guided version is relatively concise and easy to understand. The prover is generic and it works for any solver and checker, since the code only depends on the $\langle$ input, output $\rangle$ pairs that are valid.

Since Sketch only provides bounded verification, we needed to consider a tool that would extend this techniques to the unbounded case. We considered Leon with the hope it could give some more guarantees than Sketch and Java. Figure 4-2 summarized the three tools we researched the most and the space of values for which they could verify the checker is sound and complete. Leon is represented in dotted line since in the current state Leon does not provide unbounded verification either even though it provides some advantages we explain in the next subsection.

4.2 Unbounded verification and limitations

We considered proving the SoundCheck assertions for a given checker using unbounded verification systems [18, 15] backed by SMT solvers.

We focused a little more on the promising Leon[15] system which offers verification and synthesis, and can emit code to Scala.

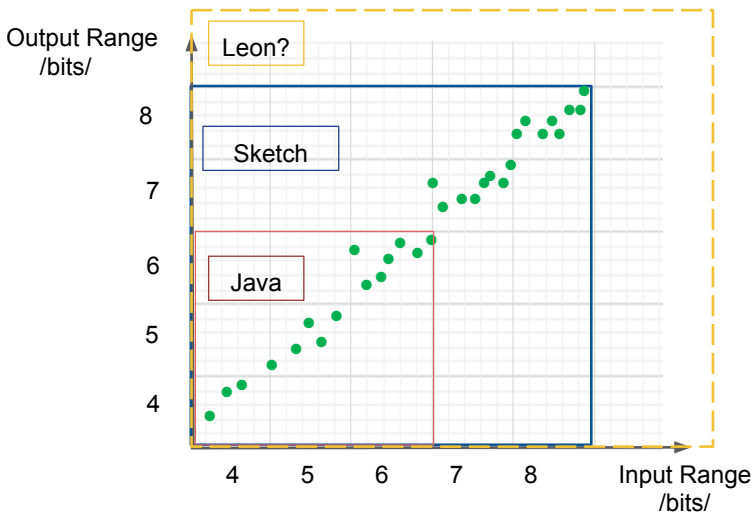

Figure 4-2: Bounded verification with Java and Sketch and possible unbounded verification with Leon

4.2.1 Leon

Leon has an interactive environment where the programmer defines methods using preconditions and postconditions. Leon is based on an SMT solver, similarly to a lot of other verification tools. It verifies the postcoditions for each method in a Scala file for the universe of all possible inputs. For each postcondition that is verified successfully, Leon outputs the time it took for the verification and for each condition that fails to be verified, Leon returns a counter-example that helps in quickly diagnosing what checker bug caused the failure.

In Figure 4-3 we show a screenshot of the Leon interactive simulator, where Leon is trying to verify a 3SUM-checker example As we can see from the message, Leon provides information about which function failed, in this case – the prover. We can also see what kind of failure it was, where the most common ones are *invalid postconditions* or failed *match exhaustiveness*. The later one shows that the code we provided does not take any action for some subset of the possible inputs. The simulator also outputs the input and output that caused the postcondtion to be

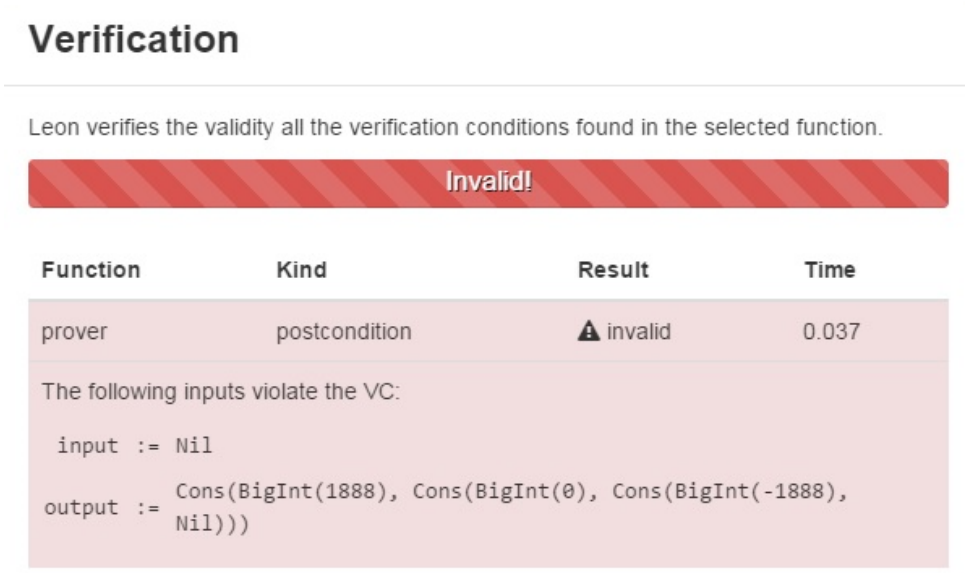

Figure 4-3: Leon simulator

violated and in how much time this counter-example was found. We explain next in more detail some checker bugs the Leon system was able to find as well as the specific bug in Figure 4-3. We also talk about the limitations of Leon, some of which we knew about before we started using the system, and some that we encountered while fixing the checker bugs.

Checker bugs discovered by Leon

We created a simple solver and checker for 3SUM and we wanted to prove the checker checked that solver using a prover, also written in Scala and verified in the Leon simulator. We intentionally made the checker wrong and gradually fixed all the bugs the checker had that Leon was able to find until we got a correct checker, as far as we could tell, which caused Leon to time out. We still think it is interesting to provide the bugs we discovered with Leon in this paragraph.

As explained above, a checker for 3SUM needs to make sure the returned numbers add up to zero and all three of them come from the original input array. The first wrong checker we implemented in Leon only checked if the first condition was satisfied,

i.e. the numbers returned from the solver added up to zero. This made Leon output a failed verification message for the prover function of the code. The message was precisely the one displayed in Figure 4-3. Clearly the numbers from the output: -1888, 0 and (-1888) sum up to zero but the input is the empty list, hence for this invalid pair $\langle$ input, output $\rangle$ the checker said it was valid. This is therefore an unsound checker.

Next, we partially fixed the bug by adding a new check in the checker which made sure that the input contained each of the numbers in the output. However, this simple check still did not search correctly since a repeated element from the output is considered found in the input even if the element is in the input only once. Leon found an example where the checker failed and returned:

The following inputs violate the VC:

input := Cons(BigInt(0), Nil)

output := Cons(BigInt(0), Cons(BigInt(0), Cons(BigInt(0), Nil)))

As expected, the output is determined by the checker to be a subset of the input. This is not the case since zero is found three times in the output but only once in the input.

We finally fixed the checker to search the input for each element in the output, by marking an element in the input after it is found so it cannot be double-counted. This final version of the checker caused Leon to time out. Since we believe this checker was actually a correctly implemented one and Leon could not verify it for us, unfortunately, we had to conclude that with the current limitations, Leon is still not enough for reaching our unbounded verification goal. We discuss in more detail the limitations of Leon next.

Limitations of Leon

The biggest disadvantage in using Leon that we discovered is that even though for some incorrect checkers we implemented, Leon was able to catch the errors and return a counterexample for most of them, Leon timed out on some correct checkers without showing any message. This showed that the backend solver is conservative and does not have the certainty to verify a correct checker when the implementation of this checker is a bit more complex.

Unfortunately, we also found examples where we gave the Leon system a non-trivial, more complex but incorrect checker for which the system could not produce a counterexample. Instead, it timed out just as when given a correct more complex checker.

The second limitation of Leon is that, like most unbounded verification systems we know about, it does not support array access and has only linked-list data structures. Our efficiency requirements for both checkers and solvers crucially depend on fast operations on hashtables, and dense matrices. These runtime performance limitations are a big reason why we decided Leon was not fully suitable for our purposes in its current version.

Despite the timeouts and the data structure limitations, we found some usages of Leon in finding counter-examples for wrong checkers. We were also pleased to discover that we never found an example where Leon verified as valid an actually incorrect checker. Such false positives are our biggest concern since we need to trust a verified checker completely. Not being able to verify a checker correct and timing out is inefficient and it requires that the programmer finds an alternative tool or method to do the full verification but does not provide false guarantees.

Chapter 5

Targeted Algorithms

We evaluate several important problems for which efficient dynamic checkers exist and show strategies used to prove the checkers sound and complete. We discuss the problems of solving systems of linear equations (LINPACK), boolean formula satisfiability (SAT), finding perfect and maximum graph matching, linear programming (LP) and integer linear programming (ILP), and sorting.

We present for each problem its instance types, checker overview, prover strategy, reference solver, running time analysis, and any challenges and design patterns it exemplifies.

For all problems, an unreliable solver can crash or timeout without returning a value. Each checker also needs a reliable parser of unreliable outputs, but these are not discussed here.

5.1 Systems of linear equations

An important classic linear algebra problem is solving a system of M linear equations with N unknowns, e.g. $Ax = b$. Standard textbook solvers[19] run in $O(N^2M + M)$, while a checker can verify a solution in $O(NM)$.

5.1.1 Instance types

A linear system of equations may have a unique solution, infinitely many solutions or no solution. This depends on the number of independent equations compared to the number of unknowns, i.e. the matrix rank. For square matrices we will focus on full rank, i.e. non-singular matrices, and check only instances with unique solutions.

5.1.2 Checker

The checker needs to plug in the unreliable solution vector $x = x_1, ..., x_N$ into the system of equation to verify the output is indeed valid up to a desired precision, i.e. $|Ax - b| < \epsilon$.

5.1.3 Prover

A simple prover should be able to prove soundness as long as we restrict the instances that need to be checked only to inputs with a unique output. Determining whether a matrix is full-rank is by itself more challenging and expensive. A solver that can reveal the matrix rank, e.g. LU or QR decomposition, or mark singular matrices is needed for the soundness proof.

5.1.4 Solver

For a reference solver we implement in Sketch a Gaussian elimination based on code from [20], which can detect singular matrices (at target precision).

5.1.5 Running time analysis

The time complexity of Gaussian elimination for a square $N \times N$ matrix is $O(N^3)$. The running time of a checker when a solution exists is $O(N^2)$.

5.2 SAT Satisfiability

Proving satisfiability of a 3-CNF boolean formula (SAT) is a classic NP-complete problem. Yet many real world instances can be efficiently solved with modern SAT solvers.

5.2.1 Instance types

Any given 3-CNF formula instance could be unsatisfiable, satisfiable with a unique assignment or satisfiable with multiple assignments.

5.2.2 Checker

A checker for satisfiable instances simply evaluates the formula with the given variable assignment and accepts if all clauses are satisfied. If there is no valid assignment, the checker always rejects.

5.2.3 Solver

Since there may be multiple valid outputs, we use a guided reference solver that normally generates the first satisfiable output it finds. When used by the prover with a guided output, it skips alternative variable assignments and returns the target output if it is satisfiable, or marks invalid.

5.2.4 Running time analysis

Checking whether an assignment is satisfiable takes linear time, while finding an assignment (SAT) or proving that no such assignment exists (UNSAT) are $\Omega(2^n)$ problems.

5.2.5 Variants – UNSAT and MAX-SAT

If UNSAT instances are a common workload, unsatisfiability proofs would need to be checked. In our system when using unreliable hardware, we should shift most of the overhead to the unreliable but faster UNSAT solver. An alternative formulation of SAT as an optimization problem is to find the maximum number of satisfiable clauses MAX-SAT. We believe it cannot be efficiently checked. These two variants demonstrate that slight changes to a problem or output specification can dramatically change checker time complexity.

5.3 Perfect Matching

A graph problem with many practical applications is finding a perfect matching in a bipartite graph. This problem is related to many problems such as matching people to jobs or positions that they qualify for, or house-seekers to houses they can afford and like. The problem can also be generalized to maximizing happiness among those people if they also assigned some scores to the houses based on their desire to buy them and this is a common maximization of utility problem that we are not dealing with in this paper.

We define the problem as follows - consider a graph $G = \langle U, V, E \rangle$ where U, V are disjoint vertex sets, and E is a set of edges with one vertex in U and another in V. A *perfect matching* is a subset of edges $M \subseteq E$ such that each vertex in U is connected to exactly one vertex in V.

5.3.1 Instance types

A bipartite graph G can have a unique perfect matching, multiple perfect matchings, or no perfect matchings.

5.3.2 Checker

Given a matching M the checker ensures that every vertex in G is connected to exactly one other vertex. Since the checker also receives a reliable copy of the input, it can assume that G is bipartite. However, it also checks if every vertex in M is part of the original graph G, if all the edges in M are also valid in the original graph G and if M has the same number of nodes as the original graph G. In other words, it must check all edge vertices in M are valid in G, and $M \subseteq E$.

5.3.3 Solver

Since we can have multiple perfect matchings for a problem instance, proving checker soundness depends on a guided solver to reach all alternative valid outputs.

Our reference solver chooses, with a `guided_for`, among the available edges connecting a non-matched vertex, the one from the provided output instead of the first possible. If the guided solver is not able to produce the considered output, then it is not a valid perfect matching.

5.3.4 Running time analysis

The checker validates each edge in M, and terminates after at most V edges on invalid output, and after exactly V edges on valid output. With an efficient edge access in $O(1)$ the total checker running time is $O(V)$.

The fastest practical algorithms[21] that run in $O(EV^{1/2})$ also solve the more general problem of maximum matching.

5.4 Maximum matching

In a general graph $G = \langle V, E \rangle$, a *matching* is a set of edges $M \subseteq E$, where no two edges share a vertex. A maximum matching is a matching with maximum number of edges.

5.4.1 Instance types

For a given graph G there may be one or many matchings of maximum cardinality. The maximum matching problem always has a solution, even if the solution is the empty edge set, in case G has no edges.

5.4.2 Solver

We recall that we trust that the solver is trustworthy if run on a reliable machine. The reason why we need to verify its output is that we assume we run it on an overclocked machine. However, we could use parts of the solver in our checker, which we assume is run on a reliable hardware and we can trust this routine gives valid results. In particular, we could use the routine, which takes a current maximum matching and looks for alternating paths based on it. Our checker would only need to verify one single matching, namely the output from the solver.

For our reference solver we use the classic and elegant Edmonds Blossom Algorithm [22]. It works by iterating over the vertices of G looking for *an alternating path* on each iteration.

Definition. Given a graph G with a matching M, consider a path P. The path is considered an *alternating path* if the edges in the path alternate between edges in M and edges not in M.

By Berge's lemma[21], a matching is maximum iff an alternating path between unmatched vertices does not exist.Otherwise, if an alternating path is found, the matching is extended.

Berge's Lemma. *Given a matching M for a bipartite graph G, M is a maximum matching if G contains no alternating paths.*

We call the method that looks for alternating paths *findAlternatingPath(G, M)* where G is the original graph and M is the output of the solver. If the method call results in null, we know there are no alternating paths in G, so we can be sure the matching the checker got M is the maximum one. If, however, the method call returns some path instead, we deduce that there is an alternating path, which based on the main theorem means that M is not the greatest matching we could create from G. Below is a snippet of the *findAlternatingPath(G, M)* routine we are going to use in the checker as well as in the solver:

```
1 struct UndirectedGraph{...};
2 UndirectedGraph maximumMatching(UndirectedGraph g) {
3  if (g.isEmpty())
4    return new UndirectedGraph();
5  UndirectedGraph result= new UndirectedGraph();
6    for (node in g){
7      addNode(node, result);
8    }
9  while (true){
10 //look for an alternating path
11   T[] path = findAlternatingPath(g, result)
12   //if no alternating path found, return result
13   if (path==null) return result;
14   //if alternating path found, update the result with it
```

```
15      updateMatching(path, result);
16        }
17      }
```

We not that the Edmonds Blossom algorithm terminates after at most $V/2$ iterations.

5.4.3 Checker

Our checker depends on the above theoretical backing to verify optimality, and also reuses the reference solver routines for a single iteration.

For a given output M, the checker first checks feasibility – i.e. all edges in M are valid in the input graph G, and every vertex is matched to exactly one other one, and optimality – M is maximum. The maximality check looks for an alternating path in M and G. If there are no alternating paths, then M is indeed the maximum matching. Otherwise, there exists a matching with bigger cardinality than M, therefore M is rejected.

5.4.4 Running time analysis

The cost of one iteration of reference solver using Edmonds is $O(VE)$, and total time is $O(V^2E)$. The checker runs only a single iteration, and depending on the cardinality of the maximum matching, it is up to $V/2$ times faster than the solver.

After a series of theoretical breakthroughs (including incorrect proofs) as surveyed in [21], best algorithms today use only $\sqrt{V/2}$ phases of searching for augmenting paths, with each phase taking $O(E)$. A checker for a best practice solver would then be only $\sqrt{V/2}$ times faster.

5.5 Linear Programming

Given a linear program $max\ c^T x : s.t. Ax \leq b, x \geq 0$, a solver must find a feasible and optimal x, i.e. satisfying the constraints and maximizing the objective function.

5.5.1 Instance Types

A linear program problem instance can be either feasible or infeasible, with feasible instances being either bounded or unbounded with no finite optimal value solution.

5.5.2 Checker

Our checkers accept only feasible bounded instances whenever all constraints are satisfied ($Ax \leq b$, $x \geq 0$).

Optimality

We can check optimality for continuous instances by verifying that the given solution is a local optimum. The termination condition of the simplex method gives us an optimality check. According to the *strong duality* theorem [19] if the primal linear programming problem has an optimal solution x^*, then its corresponding dual problem also has an optimal solution, y^*, and $c^T x^* = b^T y^*$.

We start the simplex method with an unreliable solution as an initial vector and execute for a single iteration. If the unreliable solution is indeed optimal, the simplex solver would terminate without improving the objective function value. Note that any solver method (e.g. interior-point methods) can be used for the unreliable solver, while the checker can use a simpler, and formally verified solver for this single iteration.

Integer Linear Programming

Even finding feasible solutions to Integer Linear Programming (ILP) is NP-hard. However, when optimality is also needed, no ILP end-result checker exists. ILP solvers that iterate using an LP solver, e.g. branch-and-cut, can be accelerated by using a checked unreliable LP solver subroutine.

5.6 Sorting

The problem of creating efficient sorting is important because of the many places where sorting is used as a subroutine. It has usage in various fields, including computer science, but also some less known applications of sorting are found in computational geometry, computational biology, supply chain management and others. It is also a component in the execution of many database queries and hence practically all systems using databases need algorithms for efficient sort. Therefore, coming up with a new fast approach to sorting can have a lot of impact. We think we have a novel approach for sorting which could lead to speedup without sacrificing reliability.

The main idea behind our approach, just as before, is to run some already efficient sorting algorithm, for example merge sort, on an unreliable machine and then verify the result is sorted array.

Given an input sequence and a comparison function, a sorted output must include all elements in the input and no other, and each element must be less than or equal to the next. Sorting integers in natural total order results in a unique output, and no concerns about *stability*. If stability is required then order of equal elements must also be preserved. Stability and uniqueness issues arise in sorting primitive types only if a partial order is used for comparison, e.g. ordering by last digit.

5.6.1 Instance Types

For stable sort there is a unique valid output for a given input. For non-stable sort there may be multiple valid outputs for the same input array.

5.6.2 Checker

A stable sort checker is very easy to prove correct as the unique stable sort output for any given input should be the only one accepted. The sort routine, however, has to sort keys paired with their original index positions, (which is usually already done when sorting large records). A checker given a set of indices can check when comparing two equal elements in the output, that the later one has a greater original index.

A non-stable sort checker does not need to track indices and instead checks directly the permutation property, however that is expensive in either time or space. Previous sorting checkers resorted to unsound methods, e.g. comparing the sums of the elements of the input and output arrays [5, 7]. We use $O(n)$ space to deterministically check permutation using a hashtable-backed multiset which tracks number of occurrences per element. First, the output elements are added to the multiset. Then, input elements are removed and if any element is not found the output is rejected. The multiset should be empty at the end if the output and input match.

5.6.3 Solver

A non-stable checker would accept all possible permutations of equal element subsequences in the output. Proving a non-stable checker for non-primitive type correct, requires a solver that can either enumerate all outputs, or can be *guided* to reach a desired output. Selection sort is a simple sort algorithm, which can be formally proven, but most importantly can be easily "guided" as a reference solver. We use

merge sort for a stable sort solver.

5.6.4 Running time analysis

Any comparison sort takes $\Omega(n \log n)$ to sort n integers. A checker that uses hashing for checking permutation takes $O(n)$. We limit the expected maximum collision chain, and abort the checker on an unexpectedly bad output, therefore hashtable access has a guaranteed amortized $O(1)$ time.

5.6.5 Divide and Conquer on Subproblems

Our runtime system is currently targeting unreliable systems with low error rates, on which problems of practical sizes have a high probability of successful completion. In higher error regimes, the probability of completing a larger problem successfully will quickly drop. Divide and conquer sorting algorithm strategies, as illustrated by merge sort, allow a checker to validate subproblems. A top-down merge sort will have fully sorted subarrays that can be checked and then merged reliably.

5.6.6 Input size vs. Input Range

One of the main discoveries we made through testing and timing of the sorting program and checkers we implemented is that the run time of the checker depends highly on how random the numbers in the input array are, what range we are using to get the numbers from and how big is the hash table used to check permutation. We noticed several interesting patterns with increasing range and hash size that we summarize in Figure 5-1. This is a heat map where the colors represent how good is the ratio solver to checker running time. The smallest ratios, where the checker is in fact slower than the solver, are depicted in red, with the darkest red being the worst. The good ratios are colored in green with darker green meaning better results. One

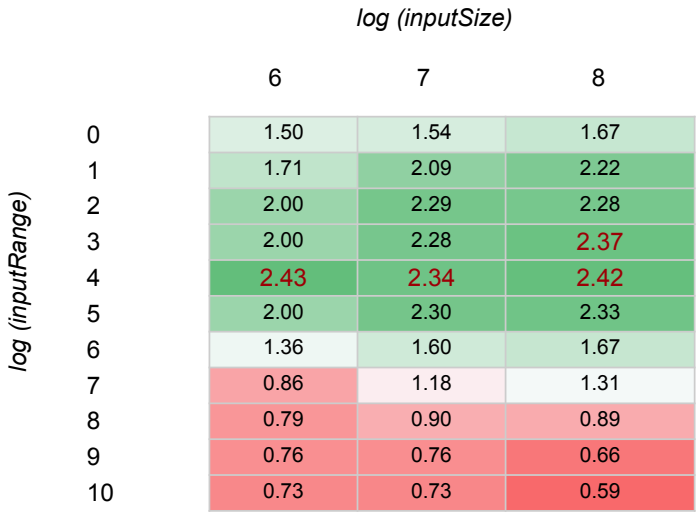

Figure 5-1: Program to checker ratio - green cells represent best ratios, whereas red cells show worst ratios

interesting observation is that for fixed input size, the increasing range decreases the ratio program to program checker. For input sizes 10^6, 10^7 and 10^8, we see that the best ratio is obtained when the range is 10^4. For this experiment we fixed the hash size so we can compare the other two dimensions - range and input size.

5.7 3-SUM Problem

The 3-SUM problem in general is defined as the problem of finding three numbers that sum up to 0 in an array of numbers that are not necessarily sorted. This is a problem that is a good candidate for our verification techniques since the solver for it is slow but checking whether three numbers sum up to zero is very fast and intuitively requires no effort.

The decision version of the problem takes as an input the list of numbers, and looks for any three numbers that sum up to 0. When it finds them, the problem outputs the numbers, either as values or as indices in the array. In the more general

version of the problem, we ask for the algorithm to find all triples of numbers that sum up to 0. Clearly, this problem is harder to solve. It is also harder to verify, since verifying we are given all the possible triple is not at all trivial without using the original solver.

We concentrate on three different checkers, based on our attempts in optimizing for both time and space complexity. We analyze the checkers together with the solver that we use to produce the verifiable triple. The reason the solver is important is because we care about the form in which the output is presented - in other words, we have two main cases - the output of the solver is indices of the three numbers that sum up to 0 in the original list or the output is the values of the numbers themselves.

5.7.1 Instance types

In general, a 3SUM problem can have no solutions, one solution or many solution. For example, an input array consisting on only positive numbers will be an instance with no solutions since all sums of three numbers will be positive.

5.7.2 Solver

There are two main checks we need to make when verifying 3-SUM and we design the solver to make the checks as easy as possible since this way we take advantage of the unreliable execution. The first condition that needs to be satisfied is that the returned numbers sum up to zero. However, this is not enough. We also need to make sure the numbers come from the original list. If the solver returned indices, we have to check what the values are by simply looking up the numbers corresponding to those indices in the original array and check for a zero sum. If the solver returned values, we have to first look for them and find them in the original array. It seems like it is always better to have a solver that returns indices since look-up is a cheap operation. We are going to analyse the methods since they are different and it is to

our advantage to consider both. We specify the two types of solver:

i. Solver , that returns the indices of the numbers - i, j, k, such that $a[i] + a[j] + a[k] = 0$.

ii. Solver that returns the numbers - a, b, c, such that $a + b + c = 0$.

As we mentioned, we are working with the solver and the checker together. The solver is important in many ways, so we will present the overall idea behind each algorithm.

Solver 1

This solver creates each possible triple of numbers and for each one check whether it gives a sum of 0. It then outputs the indices of the numbers *i, j, k* once it find them. However, if outputting values is better for the checker, the solver could be implemented to output values $a[i], a[j], a[k]$ instead.

Solver 2

The second solver sorts the array first and the uses the fact that the numbers are already sorted to determine which numbers to include or exclude from the current sum. After the initial sorting, for each i, it selects $a[i]$ and two indices l, k. If the sum $a[i] + a[l] + a[k]$ is greater than 0, it adjusts the bigger index and if the sum is smaller than 0, it adjusts the lower index. This is an algorithm that is similar to the binary search technique. It outputs the values a, b, c such that $a + b + c = 0$. Note that it is not possible for this solver to give indices since the array has been modified after the sorting and the indices do not match the original ones.

Solver 2'

This solver does the same as solver 2. However, when a, b, c are found, it also looks for them in the original list and returns their indices. This way we are using the

solver, instead of the checker, to do more work since it runs faster and we will lose performance if we run this expensive search on our reliable checker.

Solver 3

Finally, we create a third solver which does not use additional space to do the sorting. The third solver uses extra space by creating a hash table where it stores the numbers. This solver then takes additional $O(n)$ space. The reason we use a hash table is because it provide a constant look-up time of the numbers.

The algorithm starts by going through the input array and hashing all the numbers in a hash table. Then it creates all sums of two numbers from the list and for each sum it checks if the negation of the sum is in the hash table. If it is, the solver outputs the values of the numbers a, b, c.

This solver could output the indices ofa, b and c in the original list without extra work if the hash table keeps as (key, value) pairs the pairs (number, index of the number in the original list).

5.7.3 Checker

Depending on input the solver provides, the checker either gets as input the original list $a[]$ and indices i, j, k, where the numbers $a[i], a[j], a[k]$ sum up to 0, or it gets the numbers a, b, c themselves where we suppose $a + b + c = 0$.

Checker 1

This checker takes as input the output of the solver - the values a, b, c and the input array. Since in this case the values are given, the checker cannot assume they are valid numbers from the original list. Therefore, the checker has access to the list and needs to check if the numbers are a subset of the original set. The checker then needs two things: check if $a + b + c = 0$ and whether a, b and c are all in the input array.

The checker rejects if any of these conditions is not satisfied.

Checker 2

The second checker takes as input the indices i, j, k of the numbers rather than the numbers themselves. If the numbers $a[i]$, $a[j]$, and $a[k]$ sum up to zero the checker returns true, otherwise it returns false. The indices are given this time and the checker can easily find the numbers at positions i, j, k. It is clear that an additional check that i, j, k are smaller than the input size and larger than 0 is helpful and it eliminates invalid out of bounds indices if placed in the beginning.

5.7.4 Running time analysis

The trivial solver 1 is the easiest in terms of implementation, it consists of three nested **for** loops. The running time, though, is very large, in the worst case it could be as large as $O(n^3)$.

Solver 2 takes $O(n^2)$ since it takes $O(nlogn)$ for sorting and additional $O(n^2)$ for finding the numbers that add up to 0.

Solver 2' needs additional iteration compared to solver 2 in which it looks for a, b and c in the original list and outputs there indices.The running time of the slightly modified solver 2, is bigger because of the last check but it is only linear on n, therefore the running time seems to not change but in practice we might see a small increase.

Solver 3, which uses the hash table approach, takes $O(n^2)$ time since it needs to create all the sums of two numbers in case it runs long enough. Thus, it uses $O(n)$ space and $O(n^2)$ time to compute both the result in the form of indices and values.

For checker 1, notice that it takes constant time to verify i. But it takes longer to check if a, b and c are entries in $a[]$. This could be done by either going through $a[]$ and checking for the elements or sorting the list first and then finding them using binary search. Therefore, the best possible running time is $O(n)$ for simply scanning

the list since sorting takes longer. Overall, the running time of the checker is in this case $O(n)$.

The checker 2 runs in constant time since it check one sum only and three simple inequalities. Thus, this is clearly the most efficient checker in time and space.

5.7.5 Optimization version of the 3-SUM

The 3-SUM problem could be generalized in many ways, one of them being of course looking for sums of more numbers such as 4-SUM for four numbers, 5-SUM for five numbers and so on. The general problem subset-sum is hard and it could be solved using dynamic programming.

In this research we considered only finding one single triple of numbers that add up to zero but we could have many more and it would be interesting to ask the solver to give us all triples (a, b, c) of the numbers such that $a + b + c = 0$. Even though for some of the presented solvers this would not be a problem since the set up is already here, the checker complexity is increased. It is not hard to verify all the triples, simply by running some of the checkers we presented on each triple, but it is hard to verify there are not more then the ones the checker got.

Chapter 6

Evaluation

We *stimulated* real hardware errors on production processors and DRAM parts by configuring them out of specifications.

6.1 Hardware Errors Detected

6.1.1 Unreliable Memory Configuration

In this experiment we used an Intel i7-4770K processor, which has 8MB cache, and officially supports memory up to DDR3-1600 with 10-10-10 latency parameters. Decreasing memory controller timing beyond DIMM specifications improves bandwidth and latency but also quickly increases the error rate. We focused on lowering the second timing parameter tRCD (RAS# to CAS# Delay) from the reliable 12.5ns setting, since it improves random row access latency but doesn't affect reliability of most sequential accesses. This allowed us to quickly detect errors on an otherwise mostly stable system running Ubuntu 14.04.2. We first overclocked the memory controller to a seemingly stable DDR3-2000 9-10-9, i.e. at 25% higher bandwidth and 10ns tRCD latency. When tRCD was overclocked further to 9ns our sorting checker started detecting errors, that quickly grew once the working set no longer fit in last

level cache.

6.1.2 LINPACK

In this unreliable configuration we used Intel's Optimized LINPACK [23] to solve a system of linear equations of size 5000×5000. Out of 100 runs, only 69% passed a residual check on the output. Notably 9 of the "passing" tests reached a different answer but were within the official benchmark acceptable residual tolerance.

6.1.3 Sorting Checker

Using unreliable memory while sorting 2^{23} 32-bit integers with STL `std::stable_sort`, out of 1000 runs, only 48% produced a correct output. Of the failing ones, 10% were not sorted, while 90% were sorted but the output was not a permutation of the input.

6.2 Prover Effectiveness

We experimented with a few planted bugs to verify the prover is effective. Yet, we were delighted it discovered some unintentional bugs before having to run even a single test on the emitted C++ code.

We demonstrate detected bugs in the open-address hashtable implementation used in the sorting checker. The remove() method initially tried to mark a no longer needed entry as available. In an open-address hashtable, creating a gap in a collision chain may leave entries inaccessible. Properly implementing Knuth's *Algorithm R* is tricky[19], and the simplest correct implementation is to mark entries as *tombstones*. Not removing entries is in fact best, since the sorting checker never inserts entries once removal phase starts.

Several incorrect versions of the add() method follow

```
1     int i = hash(x);
```

```
2     while (ht[i] != x) i++;   // Version 1
3     while (ht[i] != x)        // Version 2
4         i=(i+1) % table_size;
```

Line 2 omits the boundary condition test and will cause an unsafe memory write – SoundCheck rejects this wrong and insecure checker with a completeness assert and a counter-example of two elements colliding for the last entry.

Line 4, adds a wrap-around guard with modulo hashtable size. However, if x does not appear in the hashtable the checker will enter an infinite loop. Bounded loop unrolling in Sketch identifies this bug. The third correction needed is to ensure that the loop will terminate if the table is full.

Overall, we have never observed SoundCheck accept any known incorrect checkers which reinforces our trust in Sketch and our prover.

Chapter 7

Conclusion

The SoundCheck system proves that a result checker is sound, and accepts only valid outputs of a target program whether using unreliable hardware or unreliable software. We evaluated a novel soundness proof strategy with the `guided_for` primitive for expressing guided non-deterministic choice, and demonstrated efficient checkers for several important applications.

A checker proven by SoundCheck, executing on reliable hardware, can be trusted to harvest throughput and latency improvements from execution on unreliable hardware without any risk.

Bibliography

[1] V. Reddi, S. Kanev, W. Kim, S. Campanoni, M. Smith, G.-Y. Wei, and D. Brooks, "Voltage noise in production processors," *Micro, IEEE*, vol. 31, no. 1, pp. 20 –28, Jan.-Feb. 2011.

[2] X. Zhang, T. Tong, S. Kanev, S. K. Lee, G.-Y. Wei, and D. Brooks, "Characterizing and evaluating voltage noise in multi-core near-threshold processors," in *Low Power Electronics and Design (ISLPED), 2013 IEEE International Symposium on*, Sep. 2013, pp. 82–87.

[3] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," in *ASPLOS XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 2006, pp. 404–415.

[4] S. de Gouw, J. Rot, F. de Boer, R. Bubel, and R. H. and, "OpenJDK's java.utils.Collection.sort() is broken: The good, the bad and the worst case," ser. CAV'15, 2015.

[5] H. Wasserman and M. Blum, "Software reliability via run-time result-checking," *J. ACM*, vol. 44, no. 6, pp. 826–849, Nov. 1997.

[6] J. Thaler, M. Roberts, M. Mitzenmacher, and H. Pfister, "Verifiable computation with massively parallel interactive proofs," *CoRR*, vol. abs/1202.1350, 2012.

[7] N. R. Saxena and E. J. McCluskey, "Linear complexity assertions for sorting," *Software Engineering, IEEE Transactions on*, 1994.

[8] M. J. Heule, W. A. Hunt Jr, and N. Wetzler, "Verifying refutations with extended resolution," in *Automated Deduction–CADE-24*. Springer, 2013, pp. 345–359.

[9] N. Wetzler, M. J. H. Heule, and W. A. Hunt, "Mechanical verification of SAT refutations with extended resolution," in *Proceedings of the 4th International Conference on Interactive Theorem Proving*, ser. ITP'13, 2013, pp. 229–244.

[10] M. Foley and C. A. R. Hoare, "Proof of a recursive program: Quicksort," *The Computer Journal*, vol. 14, no. 4, pp. 391–395, 1971.

[11] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969.

[12] D. A. Ramos and D. R. Engler, "Practical, low-effort equivalence verification of real code," in *Computer Aided Verification*. Springer, 2011, pp. 669–685.

[13] T. Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design," in *Microarchitecture. MICRO-32. Proceedings 32nd Annual International Symposium on*, 1999.

[14] V. Kiriansky and S. Amarasinghe, "Reliable computation on unreliable hardware: Can we have our digital cake and eat it?" *WACAS:Workshop on Approximate Computing Across the System Stack*, 2014.

[15] P. Suter, A. S. Köksal, and V. Kuncak, "Satisfiability modulo recursive programs," in *Proceedings of the 18th International Conference on Static Analysis*, ser. SAS'11, 2011, pp. 298–315.

[16] W. Kahan, "Pracniques: Further remarks on reducing truncation errors," *Commun. ACM*, vol. 8, no. 1, p. 40, Jan. 1965.

[17] A. Solar-Lezama, C. G. Jones, and R. Bodik, "Sketching concurrent data structures," in *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, 2008.

[18] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, ser. LPAR'10, 2010, pp. 348–370.

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms, 3rd ed.* Cambridge, MA: MIT Press, 2009.

[20] R. Sedgewick and K. Wayne, *Algorithms, 4th ed.* Addison-Wesley, 2011.

[21] H. Alt, N. Blum, K. Mehlhorn, and M. Paul, "Computing a maximum cardinality matching in a bipartite graph in time $O(n^{1.5}m/logn)$," *Inf. Process. Lett.*, vol. 37, no. 4, pp. 237–240, Feb. 1991.

[22] J. Edmonds, "Paths, trees, and flowers," *Canad. J. Math.*, vol. 17, pp. 449–467, 1965.

[23] "Intel Optimized LINPACK Benchmark for Linux OS," https://software.intel.com/en-us/node/528615.