JuniKhyat (UGC Care Group I Listed Journal)
ISSN: 2278-4632 Vol-8 Issue-02 Oct 2018

Scrutinizing the Validity of Software Verification Outcomes on Hardware Employing Approximation Techniques

Prasant Sethi
College of Engineering Bhubaneswar

***Abstract*—An new paradigm for computation that uses less energy is approximate computing, or AC. The fundamental principle of AC is to let hardware perform "approximately correct" calculations, sacrificing high precision for cheap energy consumption. This is a significant obstacle for software quality assurance since programs that have been successfully confirmed to be accurate may not be on approximate hardware. We describe a new method in this letter for figuring out when an approximation hardware software verification result is valid. We calculate the permitted tolerances for AC hardware from successful verification tests in order to do this. In other words, we establish a set of constraints that, if satisfied by the AC hardware, ensure that the verification result will transfer over to AC. Practically speaking, we also demonstrate: 1) how to use predicate abstraction as a verification methodology to extract tolerances from verification runs; and 2) how to verify such limits on hardware designs. We have put all of the strategies into practice and used many recently suggested approximate adders as well as example C programs to demonstrate them..**

I. Introduction

APPROXIMATE computing (AC) [1], [2] is a newcomputing paradigm which aims at reducing energy con- sumption at the cost of computation *precision*. A numberof application domains can tolerate AC because they areinherently resilient to imprecision (e.g., machine learning, big data analytics, image processing, and speech recogni-tion). Computation precision can be reduced by either directlymanipulating program executions on the algorithmic level(e.g., by loop perforation [3]) or by employing approximate hardware for program execution [4].

For software verification, the use of approximate hard- ware challenges soundness, and raises the question of whetherthe achieved verification result will really be valid when the program is being executed. So far, correctness in the context of AC has either studied *quantitative reliability*, i.e., the probability that outputs of functions have correct val- ues [5], [6], or differences between approximate and precise executions [7], [8] (applying differential program verification).Alternatively, some approaches plainly use types and type checking to separate the program into precise and approximate

```
int arr[1000];
for(int j:=0;j<990;) {
  j:=j+10;
  if (!(j>=0 && j<1000))
     ERR: ;
  arr[j]:=0; }
```

Fig. 1. Program *array*.

parts [4]. All of these techniques take a hardware-centric approach: take the (non-)guarantees of the hardware, and develop new analysis methods working under such weak guar-antees. The opposite direction, namely use standard program analysis procedures and let the verification impose constraints on the allowed approximation, has not been studied so far. Ranjan *et al.* [9] also checked constraints on AC hardware designs, however, these are general (not verification specific) quality constraints.

In this letter, we propose a new strategy for making soft- ware verification reliable for AC. We start with a verification run proving safety properties or termination of a program.Our approach derives from this verification run requirements (called *tolerance constraints*) on the hardware executing the program. A tolerance constraint acts like a pre/postcondition pair, and describes properties of the expected output of a hard-ware design when supplied with specific inputs. The derived tolerance constraints capture the assumptions the verification run has made on the executing hardware. Thus, they are spe- cific to the program and safety property under consideration. Typically, tolerance constraints are much less restrictive than the precise truth table of a hardware operation. The hardware design can then be checked against tolerance constraints. The outcome is a *qualitative* result (as opposed to the quantitative results of [5]): the hardware either meets the constraints or does not meet them.

We have developed a general theory for tolerance constraintextraction based on abstract interpretation (see [10] for a more complete treatment). To see our technique in practice, we haveinstantiated the framework with predicate abstraction [11]. In this case, tolerance constraints are pairs (p, q) of predicates on inputs and expected outputs of a hardware operation. As a firstexample, take a look at the program in Fig. 1. The program writes to an array within a for-loop. The safety property to be checked (encoded as an error state `ERR` which should not be reachable) is an array-index-inside-bounds check. Using x and$y$ as inputs and z as output (i.e., $z\ x\ y$), the tolerance con- straint on addition ( ) derived from a verification run showingcorrectness is

$$(x \geq 0 \wedge x \leq 989 \wedge y = 10 \Rightarrow z \geq 0 \wedge z \leq 999).$$

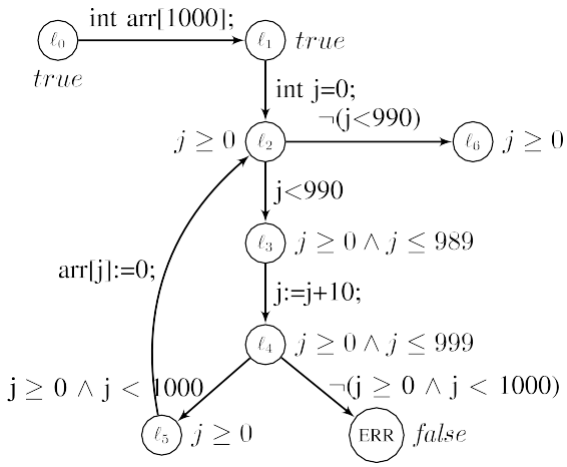

Fig. 2. Abstract transition system of program array. Fig. 2 shows the abstract transition system of program Array constructed during a safety proof via predicate abstraction.It states that the hardware adder should guarantee that adding 10 to a value in between 0 and 989 never leavesthe range [0, 999], and thus the program never crashes withan index-out-of-bounds exception. Using the analysis tool CPACHECKER [12] for verification runs, we implemented the extraction of tolerance constraints from abstract transi-tion systems constructed during verification. The constraints will be in SMT-Lib format [13]. To complete the picture,we furthermore implemented a procedure for *tolerance check-ing* on hardware designs. This technique constructs a specific checker circuit out of a given hardware design (in Verilog)and tolerance constraint. We evaluated our overall approach onexample C programs using as AC hardware different approx- imate adders from the literature. It shows that in particular termination is a fragile property for AC: programs involving standard iteration over arrays which can easily be shown to terminate on precise hardware might not terminate anymoreon AC.

II. CONSTRAINT EXTRACTION

In the following, we describe our technique along the exam-ple of Fig. 1. Our objective is to show that a program isfree of errors, i.e., the location marked with `ERR` is not reachable. Using the technique employed in [14], we can also encode proofs of program termination this way. For verifying that a program is free of errors, verification tools fre-quently employ the technique of *abstract interpretation* [15]. Abstract interpretation constructs an abstract version of the state space of a program on which the safety property is then checked. An instance of an abstract interpretation includes 1) an *abstract domain* and 2) an *abstract semantics* of pro- gram statements. The abstract program semantics has to *safely approximate* the real, concrete semantics. For our example,we use predicate abstraction: our abstract domain are pred- icates on program variables (predicates being incrementally constructed as needed for the proof of error-freedom) and the abstract semantics fixes how program statements change thesepredicates. Predicate abstraction is a frequently employed ver-ification technique, and we can simply use it off-the-shelf, including tool support.

Such a proof establishes the nonreachability of the program location marked `ERR`. An abstract transition system is a graph in which the edges are labeled with program statements and the nodes with elements of the abstract domain (next to the cir-cles), here predicates, and program locations (inside the circle).We see that the verification run has determined the necessity ofusing three predicates: $j \geq 0$, $j \leq 989$, and $j \leq 999$ (plus true and false). The error state `ERR` is not reachable as it is labeled with false. The general theory of abstract interpretation allowsus to transfer this result to the program itself: if the abstract transition system is free of errors (i.e., no locations marked `ERR` reachable) and the abstract semantics safely approximates the concrete semantics, than the program is free of errors.

Such an abstract transition system can be automatically con-structed by a verification tool. This forms the basis for our subsequent constraint extraction. The objective is to extract socalled *tolerance constraints* from the abstract transition system s.t. the following holds: whenever the approximate hardware meets these constraints, then the verification result is also validfor the program being run on this hardware.

Definition 1: A *tolerance constraint* for a program state- ment *stm* is a pair of abstract states (a_1, a_2).

A tolerance constraint tells us what property of the statement*stm* the verification tool used in its proof: an execution of *stm* on a state satisfying a_1 (precondition) should lead to a state satisfying a_2 (postcondition). Such constraints are now being extracted from the abstract transition system. As an example, consider the statement `j := j+10`. In the abstract transition system, it only occurs on the edge connecting node 4_3 (with predicate $j \geq 0 \wedge j \leq 989$) with node 4_4 (with predicate $j \geq 0 \wedge j \leq 999$). The tolerance constraint for `j := j+10` is thus$(j \geq 0 \wedge j \leq 989, j \geq 0 \wedge j \leq 999)$. Such constraints must be systematically extracted from the abstract transition system forall statements incorporating operations which will be executedon approximate hardware. If we are to use an approximate adder, we thus need the constraint for `j := j+10`. Whenever the hardware meets these constraints, the verification result will be valid for the AC hardware as well.

Theorem 1: If the abstract transition system is free of errors, the abstract semantics safely approximates the concretesemantics and all tolerance constraints are valid for the approx-imate hardware, then the program is free of errors when runon this hardware.

Proof: The proof can be found in [10].

III. IMPLEMENTATION

We implemented the above sketched constraint extraction technique as well as the constraint checking on hardware.The constraint extraction first collects all program statements using a particular operator *op* which is to be executed on AC hardware. For this, we assume all program statements to take the form of three-address code, i.e., to occur in statements`v := a op b`. The checking of the constraint on a given hardware design op^{AC} with inputs *x, y* and output *z* (in our case specified in Verilog) proceeds in three steps.

1) *Mapping:* The tolerance constraint (a_1, a_2) extracted from the abstract transition system in the form of SMT- Lib code speaks about the program variables, not the inputs and outputs of the circuit. The first step consists of replacing these variables with the appropriate inputsTABLE I

RESULTS OF EXPERIMENTS

program	#+	#stm	#tc	RCA	ACA-I	ACA-II	ETAII	GDA	GeAr
AddOne	1	22	1	✓ 5.88	× 5.97	× 6.10	× 6.06	× 6.39	× 5.99
Array	1	15	1	✓ 6.23	✓ 6.63	✓ 6.24	✓ 6.13	✓ 6.38	✓ 6.12
Attach/Detach	1	26	1	✓ 5.83	✓ 7.28	✓ 6.64	✓ 6.01	✓ 5.72	✓ 5.23
EvenSum	2	24	4	✓ 5.84	× 5.97	× 6.17	× 6.91	× 7.42	×6.82
MonotonicAdd	1	20	1	✓ 7.80	× 9.08	× 8.37	× 6.85	× 7.04	×6.79
SpecificAdd	1	13	1	✓ 6.45	✓ 6.00	× 5.67	✓ 5.92	✓ 5.92	✓5.26
Mirror_Matrix	2	42	2	✓ 8.59	× 8.36	× 9.22	× 10.49	× 10.78	× 11.47
Quotient	2	35	2	✓ 6.26	× 7.13	× 8.71	× 9.24	× 7.74	× 6.63
Sum	2	26	2	✓ 6.48	× 7.15	× 6.62	× 6.78	× 6.97	× 6.73
locks_5	5	114	31	✓ 24.32	✓ 24.51	✓ 24.19	✓ 24.53	✓ 24.85	✓ 26.63
locks_8	8	171	255	✓ 979.60	✓ 1037.94	✓ 1029.37	✓ 966.06	✓ 956.63	✓ 1004.57
cdaudio	13	1888	23	✓ 22.85	✓ 21.74	✓ 21.85	✓ 21.06	✓ 21.43	✓ 22.29
diskperf	19	981	12	✓ 17.27	✓ 17.58	✓ 18.91	✓ 17.88	✓ 19.38	✓ 17.15
floppy4	31	1370	34	✓ 31.16	✓ 30.73	✓ 30.53	✓ 31.19	✓ 29.76	✓ 33.51
kbfiltr2	11	759	15	✓ 12.59	✓ 13.61	✓ 15.96	✓ 12.09	✓ 11.12	✓ 11.51
minepump_s5_p64	2	741	3	✓ 12.88	✓ 13.85	✓ 13.08	✓ 13.56	✓ 13.58	✓ 13.19
minepump_s5_simulator	2	811	3	✓ 18.32	✓ 19.50	✓ 21.03	✓ 19.09	✓ 19.24	✓ 19.02
clnt_4	13	575	18	✓ 919.36	✓ 922.24	✓ 917.07	✓ 924.78	✓ 913.66	✓920.44
srvr_8	19	668	14	✓ 436.64	✓ 431.97	✓ 423.97	✓ 400.12	✓ 412.14	✓ 417.40
col2gray	5	367	5	✓ 9.09	✓ 8.91	✓ 8.80	✓ 8.69	✓ 9.42	✓9.53
resize	22	443	24	✓ 27.71	✓ 23.50	✓ 18.00	✓ 20.68	✓22.68	✓ 25.57
genann	61	1534	52	✓ 10.74	× 10.72	× 10.71	× 8.65	× 8.28	× 8.84

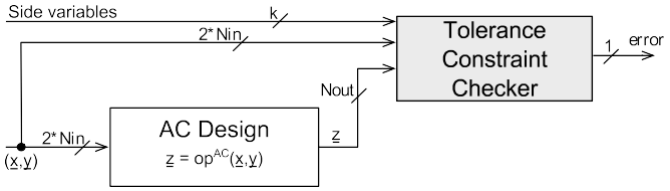

Fig. 3. Adherence checker combining AC design with tolerance constraintchecker.

and the output of the circuit, thereby getting a constraint $(\hat{a}_1, \hat{a}_2)$ (plus getting some left over side variables).

2) *Transformation:* The mapped constraint is transformed into Verilog code giving a *checker circuit*. The checker circuit is created in two steps. First, the logical formulasof the tolerance constraints are compiled to Verilog code(see [16]). We then fix a single output of the checker called *error* by setting *error* : $(a_1\ a_2)$.
3) *Combination:* The generated tolerance constraint checker is afterward combined with the hardware design of op^{AC} into an *adherence checker*. For our examples, the AC hardware designs are also given in Verilog. The combination is done using a top module that contains and wires the design of op^{AC} and the tolerance checker as submodules. The wiring is doneas depicted in Fig. 3.

The resulting circuit is afterward checked for safety, i.e., thatfor no combinations of values on the primary inputs the error flag is raised. This step can be done using standard hardware verification techniques (unsatisfiability checking of logical for-mulas derived from the combination of constraint checker andAC hardware design). An example for program *Array* can be found in [10].

IV. EXPERIMENTS

In our experiments, we used the software analysis tool CPACHECKER to verify safety of a program. We added a constraint

extraction algorithm which outputs the constraints in the SMT-Lib format. On this, the mapping is carried outand afterward the checker circuit is constructed. We employedthe tools *Yosys* [17] and *ABC* [18] for synthesis and genera- tion of a CNF formula that encodes the value of the errorflag in dependence on all the inputs to the circuit. UsingPicoSAT [19], we checked the unsatisfiability of the for- mula, denoting that the error flag is never raised, in whichcase the tolerance constraints are met by the approximatehardware design.

In the following, we give the results of our experiments.In our experiments we studied tolerance constraints for *addi- tion* (since we did not find any other publicly available approximate hardware designs). We extracted tolerance con- straints from the verification of a number of handcrafted programs (including the example given here), some programs from the subcategory `ControlFlow` and `ProductLines`of a software verification competition (SV-COMP) [20], two programs manipulating images and `genann`,[1] a library for neural networks.[2] We chose our programs as to get toler-ance constraints for a variety of verification problems. The handcrafted programs `AddOne`, `EvenSum`, `SpecificAdd`, and `MonotonicAdd` are constructed as to examine the addition of positive numbers. Programs `Sum`, `Quotient`,and `Mirror_Matrix` are programs for which termination needs to be checked. The programs from the SV-COMP (the ten programs after `Sum` in Table I) check protocol prop- erties, e.g., correct locking behavior. The image programsmust compute valid rgb colors and for `genann`, we checked proper set-up.

We checked the tolerance constraints on a standard, nonap- proximate ripple carry adder (RCA) and a set of approximate adders provided by the Karlsruhe library of [21] (called ACA-I [22], ACA-II (ACA_II_N16_Q4) [23], ETAII [24], GDA [25], and GeAr). Table I shows our results. For each

[1]https://github.com/codeplea/genann/blob/master/genann.c

[2]Some additions first had to be brought in three-address code form and in some programs we replaced some constant assignments by proper addition.

program, we show the number of additions $\#_+$, the number of program statements *#stm*, the number of constraints extracted *#tc*, whether an adder meets the extracted tolerance constraints$\mathbb{C}$ or does not , and the total checking time (including verification, extraction, circuit construction, and checking) in seconds.

Our first obser$\times$ation is that except for program `SpecificAdd`, which we created to show that the behavior between the approximate adders differs, either all approximate adders meet the extracted tolerance constraint or none of them. This is because all approximate adders use the same principle: reduction of the carry chain. In their addition, they use a set of subadders and the carry bit of the previous subadder is either dropped or imprecisely predicted. The effect of this reduction only shows off for specific numbers, which differ among the approximate adder. Interestingly, the approximate adders meet the extracted tolerance constraintsfor all of the SV-COMP programs. Note, however, that none of these required a proof of termination. On the one hand,not all additions in the programs have an effect on the correctness of the program (and thus verification imposes no constraints at all). On the other hand, typically those additions considered during verification, which had an effect, increasea variable value in the range $[0, 9]$ by one which can be computed precisely by the first subadder of all approximate adders. Finally, the image programs can handle imprecise propagation, but the neural network set-up does not.

For our own programs, one can see that all sorts of cases occur: all approximate adders satisfy the extracted constraints (as is the case for program `Array`), some do and some donot (on program `SpecificAdd`), and all do not. Imprecise carry propagation is the reason why the approximate adders cannot guarantee termination of programs `Mirror_Matrix`,`Quotient`, and `Sum`. For termination all three programs relyon an addition which is strongly monotonic up to a certain threshold (maximal int value). However, due to the impre-cise carry propagation an addition of two positive integersmay result in value zero. We conjecture that this is a general problem for termination on approximate hardware using AC adders: all programs which contain iterations over, e.g., arraysare potentially threatened to not terminate on AC.

Looking at the checking times, neither the program size (*#stm*) nor the number of additions ($\#_+$) directly influence them. In practice, the checking times are dominated by the software verification, which depends on program and property.

V. Conclusion

We have put forth a novel method in this letter for strengthening software verification against approximation hardware. The fundamental idea behind it is to use verification runs to derive limitations on AC hardware. By demonstrating that the verification result transfers to an AC hardware setting when the hardware meets the given constraints, we have demonstrated the validity of our approach. Initial experimental findings indicate that while the verification result frequently carries over, it does not always do so. Specifically, the experiments suggest that termination may be the most important factor. However, until further AC implementations of operations—aside from approximation adders—become available, more research is required. References

[1] L. Kugler, "Is 'good enough' computing good enough?" *Commun. ACM*,vol. 58, no. 5, pp. 12–14, 2015.

[2] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *Proc. 18th IEEE Eur. Test Symp.*, Avignon, France, 2013, pp. 1–6.

[3] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard, "Verified integrity properties for safe approximate program transformations," in *Proc. Workshop Partial Eval. Program Manipulation*, Rome, Italy, 2013, pp. 63–66.

[4] A. Sampson *et al.*, "EnerJ: Approximate data types for safe and general low-power computation," in *Proc. PLDI*, San Jose, CA, USA, 2011,pp. 164–174.

[5] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware," in *Proc. OOPSLA*, Indianapolis, IN, USA, 2013, pp. 33–52.

[6] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: Reliability-and accuracy-aware optimization of approximatecomputational

kernels," in *Proc. OOPSLA*, Portland, OR, USA, 2014, pp. 309–328.
[7] S. K. Lahiri and Z. Rakamarić, "Towards automated dif- ferential program verification for approximate computing," in *Proc. Workshop Approximate Comput. Across Stack (WAX)*, Portland, OR, USA, 2015. [Online]. Available: http://sampa.cs.washington.edu/wax2015/papers/lahiri.pdf
[8] S. He, S. K. Lahiri, and Z. Rakamarić, "Verifying relative safety, accu- racy, and termination for program approximations," in *NASA Formal Methods* (LNCS 9690), S. Rayadurgam and O. Tkachuk, Eds. Cham, Switzerland: Springer, 2016, pp. 237–254.
[9] A. Ranjan, A. Raha, S. Venkataramani, K. Roy, and A. Raghunathan, "ASLAN: Synthesis of approximate sequential circuits," in *Proc. DATE*,Dresden, Germany, 2014, pp. 1–6.
[10] T. Isenberg, M.-C. Jakobs, F. Pauck, and H. Wehrheim, "Deriving approximation tolerance constraints from verification runs," *CoRR*, vol. abs/1604.08784, 2016.
[11] S. Graf and H. Saidi, "Construction of abstract state graphs with PVS," in *Computer Aided Verification* (LNCS 1254), O. Grumberg, Ed. Heidelberg, Germany: Springer, 1997, pp. 72–83.
[12] D. Beyer and M. Keremoglu, "CPAchecker: A tool for configurable software verification," in *Computer Aided Verification* (LNCS 6806). Heidelberg, Germany: Springer, 2011, pp. 184–190.
[13] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB standard: Version2.5," Dept. Comput. Sci., Univ. Iowa, Iowa City, IA, USA, Tech. Rep., 2015. [Online]. Available: http://www.SMT-LIB.org
[14] B. Cook, A. Podelski, and A. Rybalchenko, "Termination proofs for systems code," in *Proc. PLDI*, Ottawa, ON, Canada, 2006, pp. 415–426.
[15] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation offixpoints," in *Proc. POPL*, Los Angeles, CA, USA, 1977, pp. 238–252.
[16] F. Pauck, "Generierung von Eigenschaftsprüfern in einem hardware/software-co-verifikationsverfahren," Bachelor thesis, Dept. Comput. Sci., Paderborn Univ., Paderborn, Germany, 2014.
[17] C. Wolf. *Yosys Open Synthesis Suite*. Accessed: Sep. 27, 2017. [Online]. Available: http://www.clifford.at/yosys/
[18] (2005). *ABC: A System for Sequential Synthesis and Verification*. [Online]. Available: https://people.eecs.berkeley.edu/ alanmi/abc/
[19] A. Biere. (2013). *Picosat*. [Online]. Available: http://fmv.jku.at/picosat
[20] D. Beyer, "Software verification and verifiable witnesses," in *Tools and Algorithms for the Construction and Analysis of Systems* (LNCS 9035), C. Baier and C. Tinelli, Eds. Heidelberg, Germany: Springer, 2015,pp. 401–416.
[21] M. Shafique, W. Ahmad, R. Hafiz, and J. Henkel, "A low latency genericaccuracy configurable adder," in *Proc. DAC*, San Francisco, CA, USA, 2015, pp. 1–6.
[22] A. K. Verma, P. Brisk, and P. Ienne, "Variable latency speculative addi-tion: A new paradigm for arithmetic circuit design," in *Proc. DATE*, Munich, Germany, 2008, pp. 1250–1255.
[23] A. B. Kahng and S. Kang, "Accuracy-configurable adder for approxi- mate arithmetic designs," in *Proc. DAC*, San Francisco, CA, USA, 2012,pp. 820–825.
[24] N. Zhu, W. L. Goh, and K. S. Yeo, "An enhanced low-power high-speedadder for error-tolerant application," in *Proc. Int. Symp. Integr. Circuits*,Singapore, 2009, pp. 69–72.
[25] R. Ye, T. Wang, F. Yuan, R. Kumar, and Q. Xu, "On reconfiguration- oriented approximate adder design and its application," in *Proc. CAD*, San Jose, CA, USA, 2013, pp. 48–54.