Model Checking with Multi-threaded IC3 Portfolios

Sagar Chaki[1(✉)] and Derrick Karimi[1,2]

[1] Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA
chaki@sei.cmu.edu
[2] Carnegie Robotics, Pittsburgh, USA
derrick.karimi@gmail.com

Abstract. Three variants of multi-threaded IC3 are presented. Each variant has a fixed number of IC3s running in parallel, and communicating by sharing lemmas. They differ in the degree of synchronization between threads, and the aggressiveness with which proofs are checked. The correctness of all three variants is shown. The variants have unpredictable runtime. On the same input, the time to find the solution over different runs varies randomly depending on the thread interleaving. The use of a portfolio of solvers to maximize the likelihood of a quick solution is investigated. Using the Extreme Value theorem, the runtime of each variant, as well as their portfolios is analyzed statistically. A formula for the portfolio size needed to achieve a verification time with high probability is derived, and validated empirically. Using a portfolio of 20 parallel IC3s, speedups over 300 are observed compared to the sequential IC3 on hardware model checking competition examples. The use of parameter sweeping to implement a solver that performs well over a wide range of problems with unknown "hardness" is investigated.

1 Introduction

In recent years, IC3 [6] has emerged as a leading algorithm for model checking hardware. It has been refined [10] and incorporated into state-of-the-art tools, and generalized to verify software [8,12]. Our interest is that IC3 is amenable to parallelization [6], and promises new approaches to enhance the capability of model checking by harnessing the abundant computing power available today. Indeed, the original IC3 paper [6] described a parallel version of IC3 informally and reported on its positive performance. In this paper, we build on that work and make three contributions.

First, we formally present three variants – IC3SYNC, IC3ASYNC and IC3PROOF – of parallel IC3, and prove their correctness. All the variants have some common features: (i) they consist of a fixed number of threads that execute in parallel; (ii) each thread learns new lemmas and looks for counterexamples (CEXes) or proofs as in the original IC3; (iii) all lemmas learned by a thread are shared with the other threads to limit duplicated effort; and (iv) if any thread finds a CEX, the overall algorithm declares the problem unsafe and terminates.

B. Jobstmann and K.R.M. Leino (Eds.): VMCAI 2016, LNCS 9583, pp. 517–535, 2016.
DOI: 10.1007/978-3-662-49122-5_25

However, the variants differ in the degree of inter-thread synchronization, and the frequency and technique for detecting proofs, making different trade-offs between the overhead and likelihood of proof-detection. Threads in IC3SYNC (cf. Sect. 4.1) synchronize after each round of new lemma generation and propagation, and check for proofs in a centralized manner. Threads in IC3ASYNC (cf. Sect. 4.2) are completely asynchronous. Proof-detection is decentralized and done by each thread periodically. Finally, threads in IC3PROOF are also asynchronous and perform their own proof detection, but more aggressively than IC3ASYNC. Specifically, each thread saves the most recent set of inductive lemmas constructed. When one of the threads finds a new set of inductive lemmas, it checks if the collection of inductive lemmas across all threads form an inductive invariant. Thus, in terms of increasing overhead (and likelihood) of proof-detection, the variants are ordered as follows: IC3SYNC, IC3ASYNC, and IC3PROOF. Collectively, we refer to all three variants as IC3PAR.

The runtime of IC3PAR is unpredictable (this is a known phenomenon [6]). In essence, the number of steps to arrive at a proof (or CEX) is sensitive to the thread interleaving. We propose to counteract this variance using a portfolio – run several IC3PARs in parallel, and stop as soon as any one terminates with an answer. But how large should such a portfolio be? Our second contribution is a statistical analysis to answer this question. Our insight is that the runtime of IC3PAR should follow the Weibull distribution [20] closely. This is because it can be thought of as the *minimum* of the runtimes of the threads in IC3PAR, which are themselves independent and identically distributed (i.i.d.) random variables. According to the Extreme Value theorem [11], the minimum of i.i.d. variables converges to a Weibull. We empirically demonstrate the validity of this claim.

Next, we hoist the same idea to a portfolio of IC3PARs. Again, the runtime of the portfolio should be approximated well by a Weibull, since it is the minimum of the runtime of each IC3PAR in the portfolio. Under this assumption, we derive a formula (cf. Theorem 5) to compute the portfolio size sufficient to solve any problem with a specific probability and speedup compared to a single IC3PAR. For example, this formula implies that a portfolio of 20 IC3PARs has 0.99999 probability of solving a problem in time no more than the "expected time" for a single IC3PAR to solve it. We empirically show (cf. Sect. 6.3) that the predictions based on this formula have high accuracy. Note that each solver in the portfolio potentially searches for a different proof/CEX. The first one to succeed provides the solution. In this way, a portfolio utilizes the power of IC3PAR to search for a wide range of proofs/CEXes without sacrificing performance.

Finally, we implement all three IC3PAR variants, and evaluate them on benchmarks from the 2014 Hardware Model Checking Competition (HMCC14) and the Tip Suite. Using each variant individually, and in portfolios of size 20, we observe that IC3PROOF and IC3ASYNC outperform IC3SYNC. Moreover, compared to a purely sequential IC3, the variants are faster, providing an average speedup of over 6 and a maximum speedup of over 300. We also show that widening the proof search of IC3 by randomizing its SAT solver is not as effective as parallelization. In addition, we evaluate the performance of the parallel version of IC3

reported earlier [6], which we refer to as IC3PAR2010. Experimental results indicate that our parallelization approach is a good complement to IC3PAR2010, and overall outperforms it. Complete details are presented in Sects. 6.1, 6.2 and 6.3.

Next, we note that IC3PAR is paramaterized by the number of threads and SAT solvers. We empirically show that the parameter value affects performance of IC3PAR significantly, and the best parameter choice is located unpredictably in the input space. Thus, for any input problem, the best parameter choice is difficult to compute. However, we show empirically that a "parameter sweeping" [2] solver that executes a randomly selected IC3PAR variant with random parameters is competitive with the best IC3PAR variant with fixed parameters over a range of problems. Complete details are presented in Sect. 6.4.

For brevity, we defer proofs and other supporting material to an extended version of the paper [7]. The rest of the paper is organized as follows. Sect. 2 surveys related work. Sect. 3 presents preliminary definitions. Sect. 4 presents the three variants of parallel IC3. Sect. 5 presents the statistical analysis of the runtime of an IC3PAR solver, as well as a portfolio of such solvers. Sect. 6 presents our experimental results, and Sect. 7 concludes.

2 Related Work

The original IC3 paper [6] presents a parallel version informally, which we call IC3PAR2010, and shows empirically that parallelism can improve verification time. Our IC3PAR solvers were inspired by this work, but are different. For example, the parallel IC3 in [6] implements clause propagation by first distributing learned clauses over all solvers and then propagating them one frame at a time, in lock step. It also introduces uncertainty in the proof search by randomizing the backend SAT solver. Our IC3PAR solvers perform clause propagation asynchronously, and use deterministic SAT solvers. We also present each IC3PAR variant formally with pseudo-code and prove their correctness. In addition, we evaluate the performance of IC3PAR2010 empirically, and show that our parallelization approach provides a good complement to (and overall outperforms) it in terms of speedup. Finally, we go beyond the earlier work on parallelizing IC3 [6] by performing a statistical analysis of the runtimes of both IC3PAR solvers and their portfolios. Experimental results (cf. Sect. 6.1) indicate that a portfolio of IC3PAR solvers is more efficient than a portfolio composed of IC3 solvers with randomized SAT solvers.

A number of projects focus on parallelizing model checking [1,3–5,13,17]. Ditter et al. [9] have developed GPGPU algorithms for explicit-state model checking. They do not report on variance in runtime, nor analyze it statistically like us, or explore the use of portfolios. Lopes et al. [15] do address variance in runtime of a parallel software model checker. However, their approach is to make the model checker's runtime more predictable by ensuring that the counterexample generation procedure is deterministic. They also do not perform any statistical analysis or explore portfolios.

Portfolios have been used successfully in SAT solving [14,16,19,22], SMT solving [21] and symbolic execution [18]. However, these portfolios are composed

of a heterogeneous set of solvers. Our focus is on homogeneous portfolios of IC3PAR solvers and statistical analysis of their runtimes.

3 Preliminaries

Assume Boolean state variables V, and their primed versions V'. A verification problem is (I, T, S) where $I(V)$, $T(V, V')$ and $S(V)$ denote initial states, transition relation and safe states, respectively. We omit V when it is clear from the context, and write S' to mean $S(V')$. Let $Post(X)$ denote the image of $X(V)$ under the transition relation T, i.e., $Post(X) = (\exists V \centerdot X \wedge T)[V' \mapsto V]$. Let $Post^k(X)$ be the result of applying $Post(\cdot)$ k times on X with $Post^0(X) = X$, and $Post^{k+}(X) = \bigcup_{j \geq k} Post^j(X)$. The verification problem (I, T, S) is safe if $Post^{0+}(I) \subseteq S$, and unsafe (a.k.a. buggy) otherwise. A "lemma" is a clause (i.e., disjunction of minterms) over V, and a "frame" is a set of lemmas.

A random variable X has a Weibull distribution with shape k and scale λ, denoted $X \sim \text{WEI}(k, \lambda)$, iff its probability density function (pdf) pdf_X and cumulative distribution function (cdf) cdf_X are defined as follows:

$$pdf_X(x) = \begin{cases} \frac{k}{\lambda}(\frac{x}{\lambda})^{k-1} e^{-(\frac{x}{\lambda})^k} & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases} \qquad cdf_X(x) = 1 - e^{-(\frac{x}{\lambda})^k}$$

Let $X_1, \ldots, X_n$ be i.i.d. random variables (rvs) whose pdfs are lower bounded at zero, i.e., $\forall x < 0 \centerdot pdf_{X_i}(x) = 0$. Then, by the Extreme Value theorem [11] (EVT), the pdf of the rv $X = \min(X_1, \ldots, X_n)$ converges to a Weibull as $n \to \infty$. The "Gamma" function, Γ, is an extension of the factorial function to real and complex numbers, with its argument decreased by 1, and is defined as follows: $\Gamma(t) = \int_{x=0}^{\infty} x^{t-1} e^{-x} dx$.

4 Parallelizing IC3

We begin with a description of the sequential IC3 algorithm. Figure 1 shows its pseudo-code. IC3 works as follows: (i) checks that no state in $\neg S$ is reachable in 0 or 1 steps from some state in I (lines 16–17); (ii) iteratively construct an array of frames, each consisting of a set of clauses, as follows: (a) initialize the frame array and flags (lines 18–19); (b) `strengthen()` the frames by adding new clauses (line 22); if a counterexample is found in this step (indicated by *bug* being set), IC3 terminates (line 24); (c) otherwise, `propagate()` clauses that are inductive to the next frame (line 26); if a proof of safety is found (indicated by an empty frame), IC3 again terminates (lines 27–28); (d) add a new empty frame to the end of the array (line 30) and repeat from step (b). In the rest of this paper we use the term "function" to mean a "procedure", as opposed to a mathematical function. In particular, we use terms "pdf" and "cdf" to mean probability and cumulative distribution functions of random variables, respectively.

```
1 //-- global variables
2 var (I, T, S) : problem (P)
3 var F: frame[] (array of frames)
4 var K: int (size of F)
5 var bug: bool (CEX flag)
6
7 //-- invariants
8 ∀i ∈ [0, K − 1], let f(i) = ⋀_{j∈[i,K−1]} ⋀_{α∈F[j]} α
9 A1 : ∀i ∈ [0, K − 1] . I ⟹ f(i)
10 A2 : ∀i ∈ [0, K − 2] . f(i) ∧ T ⟹ f′(i + 1)
11 A3 : ∀i ∈ [0, K − 3] . f(i) ∧ T ⟹ S′
12 A4 : ∀i ∈ [0, K − 2] . f(i) ∧ T ⟹ S′
13
14 //-- main function.
15 bool IC3 ()
16   if (I ∧ ¬S ≠ ⊥) ∨ (I ∧ T ∧ ¬S′ ≠ ⊥)
17     return ⊥;
18   K := 3; F[0] := I; F[1] := ∅;
19   F[2] := ∅; bug := ⊥;
20   while (⊤)
21     @INV{I1 : A1 ∧ A2 ∧ A3}
22     strengthen(F, K);
23     @INV{I2 : bug ∨ (A1 ∧ A2 ∧ A4)}
24     if (bug) return ⊥;
25     @INV{I3 : A1 ∧ A2 ∧ A4}
26     propagate(F, K);
27     if (∃i ∈ [1, K − 2] . F[i] = ∅)
28       return ⊤;
29     @INV{I3}
30     F[K] := ∅; K := K + 1;
31 //-- add new lemmas to frames. stop
32 //-- with a CEX or when A4 holds.
33 void strengthen (F, K)
34   var PQ : priority queue
35   while (⊤)
36     if (f(K − 2) ∧ T ⟹ S′) return;
37     let m ⊨ f(K − 2) ∧ T ∧ ¬S′;
38     PQ.insert(m, K − 3);
39     while (¬PQ.empty())
40       (m, l) := PQ.top();
41       if (f(l) ∧ T ∧ m′ = ⊥)
42         F[l + 1] := F[l + 1] ∪ {¬m};
43         PQ.erase(m, l);
44       else if (l = 0)
45         bug := ⊤; return;
46       else
47         let m0 ⊨ f(l) ∧ T ∧ m;
48         PQ.insert(m0, l − 1);
49
50
51 //-- push inductive clauses forward.
52 void propagate(F, K)
53   for i : 1 . . . K − 2
54     for α ∈ F[i]
55       if (f(i) ∧ T ⟹ α′)
56         F[i + 1] := F[i + 1] ∪ {α};
57         F[i] := F[i] \ {α};
```

Fig. 1. Pseudo-Code for IC3. Variables are passed by reference, and arrays are indexed from 0. This holds for all the pseudo-code in this article.

Definition 1 (Frame Monotonicity). A function is frame monotonic if at each point during its execution, $\forall i \in [0, K-1] \,.\, f(i) \implies \tilde{f}(i)$ where $\tilde{f}(i)$ is the value of $f(i)$ when the function was called.

Correctness. Figure 1 also shows the invariants (indicated by `@INV`) before and after `strengthen()` and `propagate()`. Since `strengthen()` always adds new lemmas to frames, it is frame monotonic, and hence it maintains A_1 and A_3. It also maintains A_2 since a new lemma $\neg m$ is added to frame $F[l+1]$ (line 42) only if $f(l) \wedge T \implies \neg m'$ (line 41). Finally, when `strengthen()` returns, then either $bug = \top$ (line 45), or $f(K-2) \wedge T \implies S'$ (line 36). Hence $\mathcal{I}_2$ is a valid post-condition for `strengthen()`. Also, `propagate()` is frame monotonic since it always pushes inductive lemmas forward (the order of the two statements at lines 56–57 is crucial for this). Hence, `propagate()` maintains A_1 and A_4 at all times. It also maintains A_2 since a new lemma α is added to frame $F[i+1]$ (line 56) only if $f(i) \wedge T \implies \alpha'$ (line 55). Hence $\mathcal{I}_3$ is a valid post-condition for `propagate()`. Finally, note that $A_4 \equiv A_3 \wedge f[\mathbf{K}-2] \implies S$. Hence, after $\mathbf{K}$ is incremented, A_4 becomes A_3. Also, since the last frame is initialized to $\emptyset$, A_1 and A_2 are preserved. Hence: $\{\mathcal{I}_3\}\mathbf{F}[\mathbf{K}] := \emptyset; \mathbf{K} := \mathbf{K}+1; \{\mathcal{I}_1\}$. The correctness of IC3 is summarized by Theorem 1. Its proof is in the appendix of [7].

Theorem 1. *If* `IC3()` *returns* $\top$*, then the problem is safe. If* `IC3()` *returns* $\bot$*, then the problem is unsafe.*

```
58 //-- global variables
59 var (I, T, S) : problem (P)
60 var ∀i ∈ [1, n] . F_i : frame[]
61 var K: int (size of each F_i)
62 var bug: bool (CEX flag)
63
64 //-- invariants
65 ∀j ∈ [0, K − 1], let
66     f(j) = ⋀_{i∈[1,n]} ⋀_{k∈[j,K−1]} ⋀_{α∈F_i[k]} α
67
68 B_1 : ∀j ∈ [0, K − 1] . I ⟹ f(j)
69 B_2 : ∀j ∈ [0, K − 2] . f(j) ∧ T ⟹ f′(j + 1)
70 B_3 : ∀j ∈ [0, K − 3] . f(j) ∧ T ⟹ S′
71 B_4 : ∀j ∈ [0, K − 2] . f(j) ∧ T ⟹ S′
```

```
72 bool IC3Sync (n)
73   if (I ∧ ¬S ≠ ⊥) ∨ (I ∧ T ∧ ¬S′ ≠ ⊥)
74     return ⊥;
75   K := 3; bug := ⊥;
76   ∀i ∈ [1, n] . F_i[0] := I; F_i[1] := F_i[2] := ∅;
77   while (⊤)
78     @INV{I_4 : B_1 ∧ B_2 ∧ B_3}
79     {strengthen(F_1, K); propagate(F_1, K)}
80                     ‖ ··· ‖
81     {strengthen(F_n, K); propagate(F_n, K)};
82     @INV{I_5 : bug ∨ (B_1 ∧ B_2 ∧ B_4)}
83     if (bug) return ⊥;
84     @INV{I_6 : B_1 ∧ B_2 ∧ B_4}
85     if (∃j ∈ [1, K − 2] . ∀i ∈ [1, n] . F_i[j] = ∅)
86       return ⊤;
87     @INV{I_6}
88     ∀i ∈ [1, n] . F_i[K] := ∅; K := K + 1;
```

Fig. 2. Pseudo-Code for IC3SYNC(n). Functions `strengthen()` and `propagate()` are defined in Fig. 1.

We now present the three versions of parallel IC3 and their correctness (their termination follows in the same way as IC3 [6] – see Theorem 5 in the appendix of [7]).

4.1 Synchronous Parallel IC3

The first parallelized version of IC3, denoted IC3SYNC, runs a number of copies of the sequential IC3 "synchronously" in parallel. Let IC3SYNC(n) be the instance of IC3SYNC consisting of n copies of IC3 executing concurrently. The copies maintain separate frames. However, for any copy, the frames of other copies act as "background lemmas". Specifically, the copies interact by: (i) using the frames of all other copies when computing $f(i)$; (ii) declaring the problem unsafe if any copy finds a counterexample; (iii) declaring the problem safe if some frame becomes empty across all the copies; and (iv) "synchronizing" after each call to `strengthen()` and `propagate()`.

The pseudo-code for IC3SYNC(n) is shown in Fig. 2. The main function is `IC3Sync()`. After checking the base cases (lines 73–74), it initializes flags and frames (lines 75–76), and then iteratively performs the following steps: (i) run n copies IC3 where each copy does a single step of `strengthen()` followed by `propagate()` (lines 79–81); (ii) check if any copy of IC3 found a counterexample, and if so, terminate (line 83); (iii) check if a proof of safety has been found, and if so, terminate (lines 85–86); and (iv) add a frame and repeat from step (i) above (line 88). Functions `strengthen()` and `propagate()` are syntactically identical to IC3 (cf. Fig. 1). However, the key semantic difference is that lemmas from all copies are used to define $f(j)$ (lines 65–66). Global variables are shared, and accessed atomically. Note that even though all IC3 copies write to variable *bug*, there is no race condition since they always write the same value ($\top$).

Correctness. The correctness of IC3SYNC follows from the invariants specified in Fig. 2. To show these invariants are valid, the main challenge is to show that if $\mathcal{I}_4$ holds at line 78, then $\mathcal{I}_5$ holds at line 82. Note that since `strengthen()`

and `propagate()` are frame monotonic, they preserve B_1 and B_3. This means that $B_1 \wedge B_3$ holds at line 82. Now suppose that at line 82, we have $\neg bug$. This means that each `strengthen()` called at lines 79–81 returned from line 36. Thus, the condition $f(\mathbf{K}-2) \wedge T \implies S'$ was established at some point, and once established, it continues to hold due to the frame monotonicity of `strengthen()` and `propagate()`. Since $B_4 \equiv B_3 \wedge (f(\mathbf{K}-2) \wedge T \implies S')$, we therefore know that $B_1 \wedge B_4$ holds at line 82. Also, B_2 holds at line 82 since a new lemma α is only added to frame $F_i[j+1]$ by `strengthen()` (line 42) and `propagate()` (line 56) under the condition $f(j) \wedge T \implies \alpha'$. Note that once $f(j) \wedge T \implies \alpha'$ is true, it continues to hold even in the concurrent setting due to frame monotonicity. Finally, the statement at line 88 transforms $\mathcal{I}_6$ to $\mathcal{I}_4$. The correctness of IC3SYNC is summarized by Theorem 2. Its proof is in the appendix of [7].

Theorem 2. *If* `IC3Sync()` *returns* $\top$*, then the problem is safe. If* `IC3Sync()` *returns* $\bot$*, then the problem is unsafe.*

```
89 //-- invariants
90 ∀j ∈ [0, max(K_1, ..., K_n) − 1], let
91     f(j) = ⋀_{i∈[1,n]} ⋀_{k∈[j,K_i−1]} ⋀_{α∈F_i[k]} α
92
93 C_1 : ∀j ∈ [0, K_i − 1] . I ⟹ f(j)
94 C_2 : ∀j ∈ [0, K_i − 2] . f(j) ∧ T ⟹ f'(j + 1)
95 C_3 : ∀j ∈ [0, K_i − 3] . f(j) ∧ T ⟹ S'
96 C_4 : ∀j ∈ [0, K_i − 2] . f(j) ∧ T ⟹ S'
97
98
99
100 //-- top-level function
101 bool IC3Async (n)
102   if (I ∧ ¬S ≠ ⊥) ∨ (I ∧ T ∧ ¬S' ≠ ⊥)
103     return ⊥;
104   bug := ⊥;
105   IC3Copy(1) ◇···◇ IC3Copy(n);
106   return bug ? ⊥ : ⊤;
```

```
107 //-- global variables
108 var (I, T, S) : problem (P)
109 var ∀i ∈ [1, n] . F_i : frame[]
110 var ∀i ∈ [1, n] . K_i : int (size of F_i)
111 var bug: bool (CEX flag)
112
113 void IC3Copy (i)
114   K_i := 3; F_i[0] := I;
115   F_i[1] := ∅; F_i[2] := ∅;
116   while (⊤)
117     @INV{I_7 : C_1 ∧ C_2 ∧ C_3}
118     strengthen(F_i, K_i);
119     @INV{I_8 : bug ∨ (C_1 ∧ C_2 ∧ C_4)}
120     if (bug) return;
121     @INV{I_9 : C_1 ∧ C_2 ∧ C_4}
122     propagate(F_i, K_i);
123     if (∃j ∈ [1, K_i − 2] . ∀l ∈ [1, n] . F_l[j] = ∅)
124       return;
125     @INV{I_9}
126     F_i[K_i] := ∅; K_i := K_i + 1;
```

Fig. 3. Pseudo-Code for IC3ASYNC(n). Functions `strengthen()` and `propagate()` are defined in Fig. 1.

4.2 Asynchronous Parallel IC3

The next parallelized version of IC3, denoted IC3ASYNC, runs a number of copies of the sequential IC3 "asynchronously" in parallel. Let IC3ASYNC(n) be the instance of IC3ASYNC consisting of n copies of IC3 executing concurrently. Similar to IC3SYNC, the copies maintain separate frames, interact by sharing lemmas when computing $f(i)$, and declare the problem unsafe if any copy finds a counterexample. However, due to the lack of synchronization, proof detection is distributed over all the copies instead of being centralized in the main thread.

Figure 3 shows the pseudo-code for IC3ASYNC(n). The main function is `IC3Async()`. After checking the base cases (lines 102–103), it initializes flags

(line 104), launches n copies of IC3 in parallel (line 105) and waits for some copy to terminate (the $\diamond$ operator), and checks the flag and returns with an appropriate result (line 106). Function `IC3Copy()` is similar to `IC3()` in Fig. 1. The key difference is that lemmas from all copies are used to compute $f(j)$ (lines 90–91).

Correctness. The correctness of IC3ASYNC follows from the invariants specified in Fig. 3. To see why these invariants are valid, note that C_1 and C_3 are always preserved due to frame monotonicity. If `strengthen()` returns with $bug = \bot$, then it returned from line 36, and hence $f(\mathbf{K}_i - 2) \wedge T \implies S'$ was true at some point in the past and continues to hold due to frame monotonicity. Together with C_3, this implies that C_4 holds at line 119. Also, C_2 holds at line 119 since a new lemma α is only added to frame $F_i[j+1]$ by `strengthen()` (line 42) and `propagate()` (line 56) under the condition $f(j) \wedge T \implies \alpha'$. Note that once $f(j) \wedge T \implies \alpha'$ is true, it continues to hold even under concurrency due to frame monotonicity. Hence, $\mathcal{I}_8$ holds at line 119. Since *bug* is never set to $\bot$ after line 104, this means that $\mathcal{I}_9$ holds at line 121 even under concurrency. Finally, the statement at line 126 transforms $\mathcal{I}_9$ to $\mathcal{I}_7$. The correctness of IC3ASYNC is summarized by Theorem 3. Its proof is in the appendix of [7].

Theorem 3. *If* `IC3Async()` *returns* $\top$*, then the problem is safe. If* `IC3Async()` *returns* $\bot$*, then the problem is unsafe.*

```
127 //-- global variables
128 var (I, T, S) : problem (P)
129 var ∀i ∈ [1,n].F_i, P_i: frame[]
130 var ∀i ∈ [1,n].K_i: int (size of F_i and P_i)
131 var bug, safe: bool (CEX and proof flags)
132
133
134 void IC3PrCopy (i)
135   K_i := 3; F_i[0] := I;
136   F_i[1] := ∅; F_i[2] := ∅;
137   while (⊤)
138     @INV{I_7 : C_1 ∧ C_2 ∧ C_3}
139     strengthen(F_i, K_i);
140     @INV{I_8 : bug ∨ (C_1 ∧ C_2 ∧ C_4)}
141     if (bug) return;
142     @INV{I_9 : C_1 ∧ C_2 ∧ C_4}
143     propProof(F_i, K_i);
144     if (safe) return;
145     @INV{I_9}
146     F_i[K_i] := ∅; K_i := K_i + 1;
```

```
147 bool IC3Proof (n)
148   if (I ∧ ¬S ≠ ⊥) ∨ (I ∧ T ∧ ¬S' ≠ ⊥)
149     return ⊥;
150   bug := ⊥; safe := ⊥;
151   IC3PrCopy(1) ◇···◇ IC3PrCopy(n);
152   return bug ? ⊥ : ⊤;
153
154 void propProof(F, K)
155   for j : 1...K − 2
156     for α ∈ F[j]
157       if (f(j) ∧ T ⟹ α')
158         F[j+1] := F[j+1] ∪ {α};
159         F[j] := F[j] \ {α};
160         if (F[j] = ∅)
161           P_j := ⋃_{j<k≤K−1} F[k];
162           Π := ⋃_{{i|1≤i≤n∧j<K_i}} P_i;
163           if (Π ∧ T ⟹ Π')
164             safe := ⊤; return;
```

Fig. 4. Pseudo-Code for IC3PROOF(n). Function `strengthen()` is defined in Fig. 1. Formulas $f(j), \mathcal{I}_7, \mathcal{I}_8$, and $\mathcal{I}_9$ are defined in Fig. 3.

4.3 Asynchronous Parallel IC3 with Proof-Checking

The final parallelized version of IC3, denoted IC3PROOF, is similar to IC3ASYNC, but performs more aggressive checking for proofs. Let IC3PROOF(n) be the instance of IC3PROOF consisting of n copies of IC3 executing concurrently. Similar to IC3ASYNC, the copies maintain separate frames, interact by sharing lemmas

when computing $f(i)$, and declare the problem unsafe if any copy finds a counterexample. However, whenever a copy finds an empty frame, it checks whether the set of lemmas over all the copies for that frame forms an inductive invariant.

The pseudo-code for IC3PROOF(n) is shown in Fig. 4. The main function is `IC3Proof()`. After checking the base cases (lines 148–149), it initializes flags (line 150), launches n copies of IC3 in parallel (line 151) and waits for at least one copy to terminate, and checks the flag and returns with an appropriate result (line 152). Function `IC3PrCopy` is similar to `IC3` in Fig. 1, but calls `propProof()` instead of `propagate()` where, once an empty frame is detected (line 160), we check whether a proof has been found by collecting the lemmas for that frame (lines 161–162), and checking if these lemmas are inductive (line 163).

Correctness. The correctness of IC3PROOF follows from the invariants (whose validity is similar to those for IC3ASYNC) specified in Fig. 4. It is summarized by Theorem 4. The proof of the theorem is in the appendix of [7].

Theorem 4. *If* `IC3Proof()` *returns* $\top$*, then the problem is safe. If* `IC3Proof()` *returns* $\bot$*, then the problem is unsafe.*

5 Parallel IC3 Portfolios

In this section, we investigate the question of how a good portfolio size can be selected if we want to implement a portfolio of IC3PARs. We begin with an argument about the pdf of the runtime of IC3ASYNC(n).

Conjecture 1. The runtime of IC3ASYNC(n) converges to a Weibull rv as $n \to \infty$.

Argument: Recall that each execution of IC3ASYNC(n) consists of n copies of IC3 running in parallel, and that IC3ASYNC(n) stops as soon as one copy finds a solution. We can consider the runtime of each copy of IC3 to be a rv. Specifically, let rv X_i be the runtime of the i-th copy of IC3 under the environment provided by the other $n-1$ copies. Recall that the pdf of X_i has a lower bound of 0, since no run of IC3 can take negative time. Also, for the sake of argument, assume that $(X_1, \ldots, X_n)$ are i.i.d. since the interaction between the copies of IC3 is logical and symmetric. Finally, let X be the random variable denoting the runtime of IC3ASYNC(n). Note that $X = \min(X_1, \ldots, X_n)$. Hence, by the EVT, $X \sim \text{WEI}(k, \lambda)$ for large n. □

A similar argument holds for IC3SYNC and IC3PROOF, and therefore their runtime should follow Weibull as well. Indeed, despite the assumption of $(X_1, \ldots, X_n)$ being i.i.d., we empirically find that the runtime of IC3PAR(n) follows a Weibull distribution closely for even modest values of n. Specifically, we selected 10 examples (5 safe and 5 buggy) from HWMCC14, and for each example we:

1. Executed IC3ASYNC(4) "around" 3000 times (we actually ran each example 3000 times but some timed out – the exact number of timeouts varied across examples);
2. Measured the runtimes;

	IC3SYNC (4)				IC3ASYNC (4)				IC3PROOF (4)			
Example	k	λ	μ, μ^*	σ, σ^*	k	λ	μ, μ^*	σ, σ^*	k	λ	μ, μ^*	σ, σ^*
6s286	4.07	1119	1015,1015	280,274	4.44	990	902,903	230,220	4.35	980	892,892	232,228
intel026	2.71	49.0	43.6,44.2	17.3,14.6	3.70	50.2	45.3,46.2	13.6,10.1	3.70	50.1	45.2,46.1	13.6,10.3
6s273	3.80	26.1	23.6,23.6	6.93,6.57	4.11	23.5	21.3,21.4	5.85,5.36	4.17	23.3	21.2,21.3	5.73,5.29
intel057	6.58	16.0	14.9,15.1	2.66,2.11	7.31	17.2	16.1,16.1	2.60,2.46	7.52	17.8	16.7,16.9	2.63,2.07
intel054	7.82	24.3	22.8,23.0	3.46,2.94	8.69	26.1	24.6,24.8	3.38,2.84	9.26	26.1	24.7,24.8	3.20,2.92
6s215	2.38	7.69	6.82,7.03	3.05,2.34	4.71	6.75	6.17,6.21	1.49,1.34	4.72	6.38	5.84,5.90	1.41,1.21
6s216	1.95	35.1	31.1,31.0	16.6,16.9	3.56	27.5	24.8,24.9	7.74,6.97	2.78	28.1	25.0,25.1	9.74,9.05
oski3ub1i	5.98	54.9	50.9,51.4	9.90,7.90	7.02	52.3	48.9,49.2	8.20,6.71	4.78	54.8	50.2,50.8	11.9,9.53
oski3ub3i	5.71	52.4	48.5,48.9	9.84,8.00	5.51	52.2	48.2,48.6	10.1,8.51	5.66	52.2	48.2,48.5	9.87,8.39
oski3ub5i	5.08	66.8	61.4,61.9	13.8,11.6	4.94	67.2	61.6,62.0	14.2,12.4	4.93	66.2	60.7,61.1	14.0,12.1
SAFE	5.00	246	224,224	62.1,60.2	5.65	221	202,202	51.1,48.3	5.80	219	200,200	51.4,49.7
BUG	4.22	43.4	39.7,40.0	10.6,9.37	5.15	41.2	37.9,38.2	8.36,7.20	4.58	41.5	38.0,38.3	9.42,8.07
ALL	4.61	145	131,132	36.4,34.7	5.40	131	120,120	29.7,27.7	5.19	130	119,119	30.4,28.9

Fig. 5. Fitting IC3PAR(4) runtime to Weibull. First 5 examples are safe, next 5 are buggy; SAFE, BUG, ALL = average over safe, buggy, and all examples; μ, μ^* = predicted, observed mean; σ, σ^* = predicted, observed standard deviation.

3. Estimated the k and λ values for the Weibull distribution that best fits these values (using maximum likelihood estimation and the R statistical tool); and
4. Computed the observed mean and standard deviation from the data, and the predicted mean and standard deviation from the k and λ estimates.

We repeated these experiments with IC3SYNC and IC3PROOF. The results are shown in Fig. 5. We see that in all cases, the observed mean and standard deviation is quite close to the predicted ones, indicating that the estimated Weibull distribution is a good fit for the measured runtimes. IC3ASYNC and IC3PROOF have similar performance, are and slightly faster overall than IC3SYNC, indicating that additional synchronization is counter-productive. The estimated k and λ values vary widely over the examples, indicating their diversity. Note that smaller values of λ mean a smaller expected runtime.

Determining Portfolio Size. Consider a portfolio of IC3PARs. In general, increasing the size of the portfolio reduces the expected time to solve a problem. However, there is diminishing returns to adding more solvers to a portfolio in terms of expected runtime. We now express this mathematically, and derive a formula for computing a portfolio size to achieve an runtime with a target probability. Consider a portfolio of m IC3PAR solvers run on a specific problem. Let Y_i denote the runtime of the i-th IC3PAR. From previous discussion we know that $Y_i \sim \text{WEI}(k, \lambda)$ for some k and λ. Therefore, the cdf of Y_i is: $cdf_{Y_i}(x) = 1 - e^{-(\frac{x}{\lambda})^k}$. Note that Y_i refers to an instance of IC3PAR, whereas X_i, used before, referred to a single thread (executing a copy of IC3) within an instance of IC3PAR.

Let Y be the rv denoting the runtime of the portfolio. Thus, we have $Y = \min(Y_1, \ldots, Y_m)$. More importantly, the cdf of Y is:

$$\begin{aligned} cdf_Y(x) &= 1 - (1 - cdf_{Y_1}(x)) \times \cdots \times (1 - cdf_{Y_m}(x)) \\ &= 1 - (e^{-(\frac{x}{\lambda})^k})^m = 1 - e^{-m(\frac{x}{\lambda})^k} = 1 - e^{-(\frac{x m^{\frac{1}{k}}}{\lambda})^k} \end{aligned}$$

Note that this means Y is also a Weibull rv, not just when $m \to \infty$ (as per the EVT) but for all m. More specifically, $Y \sim \text{WEI}(k, \frac{\lambda}{m^{\frac{1}{k}}})$. Recall that if $m = 1$, then the expected time to solve the problem by the portfolio is $E[Y_1]$. We refer to this time as t^*, the expected solving time for a single IC3PAR. Recall the Gamma function Γ. Since $Y_1 \sim \text{WEI}(k, \lambda)$, it is known that $t^* = \lambda\Gamma(1 + \frac{1}{k})$. Now, we come to our result, which expresses the probability that a portfolio of m IC3PARs will require no more than t^* to solve the problem.

Theorem 5. *Let $p(m)$ be the probability that $Y \leq t^*$. Then $p(m) > 1 - e^{-\frac{m}{e^\gamma}}$ where $\gamma \approx 0.57721$ is the Euler-Mascheroni constant.*

Proof. We know that:

$$p(m) = cdf_Y(t^*) = 1 - e^{-m(\Gamma(1+\frac{1}{k}))^k} = 1 - (\alpha(k))^m, \text{ where } \alpha(k) = e^{-(\Gamma(1+\frac{1}{k}))^k}$$

Next, observe that $\alpha(k)$ increases monotonically with k but does not diverge as $k \to \infty$. For example, Fig. 11 in the appendix of [7] shows a plot of $\alpha(k)$. Since we want to prove an lower bound on $p(m)$, let us consider the limiting case $k \to \infty$. It can be shown that (see Lemma 2 in the appendix of [7]): $\lim_{k\to\infty} \alpha(k) = e^{-\frac{1}{e^\gamma}}$. In practice, as seen in Fig. 11 in the appendix of [7], the value of $\alpha(k)$ converges quite rapidly to this limit as k increases. For example, $\alpha(5) > 0.91 \cdot e^{-\frac{1}{e^\gamma}}$, and $\alpha(10) > 0.95 \cdot e^{-\frac{1}{e^\gamma}}$. Since $\forall k \,.\, \alpha(k) < e^{-\frac{1}{e^\gamma}}$, we have our result:

$$p(m) > 1-(e^{-\frac{1}{e^\gamma}})^m = 1-e^{-\frac{m}{e^\gamma}} \qquad \square$$

Achieving a Target Probability. Now suppose we want $p(m)$ to be greater than some target probability p. Then, from Theorem 5, we have:

$$p = 1 - (e^{-\frac{1}{e^\gamma}})^m \iff 1 - p = e^{-\frac{m}{e^\gamma}} \iff \ln(1-p) = -\frac{m}{e^\gamma}$$
$$\iff \ln(\tfrac{1}{1-p}) = \tfrac{m}{e^\gamma} \iff m = e^\gamma \ln(\tfrac{1}{1-p})$$

For example, if we want $p = 0.99999$, then $m \approx 20$. Thus, a portfolio of 20 IC3PARs has about 0.99999 probability of solving a problem at least as quickly as the expected time in which a single IC3PAR will solve it. We validated the efficacy of Theorem 5 by comparing its predictions with empirically observed results on the HWMCC14 benchmarks. Overall, we find the observed and predicted probabilities to agree significantly. Further details are presented in Sect. 6.3.

Speeding Up the Portfolio. To reduce the portfolio's runtime below t^*, we must increase m appropriately. In general, for any constant $c \in [0, 1]$, the probability that a portfolio of m IC3PAR solvers will have a runtime $\leq c \cdot t^*$ is given by:

$$p(m, c, k) = 1 - e^{-m(c\cdot\Gamma(1+\frac{1}{k}))^k}$$

For $c < 1$ we do not have a closed form for $\lim_{k\to\infty} p(m, c, k)$, unlike when $c = 1$. However, the value of $p(m, c, k)$ is computable for fixed m, c and k. Figure 6(a) plots $p(m, c, 4)$ for $m = \{1, \ldots, 100\}$ and $c = \{0.4, 0.5, 0.6\}$. Figure 6(b) plots

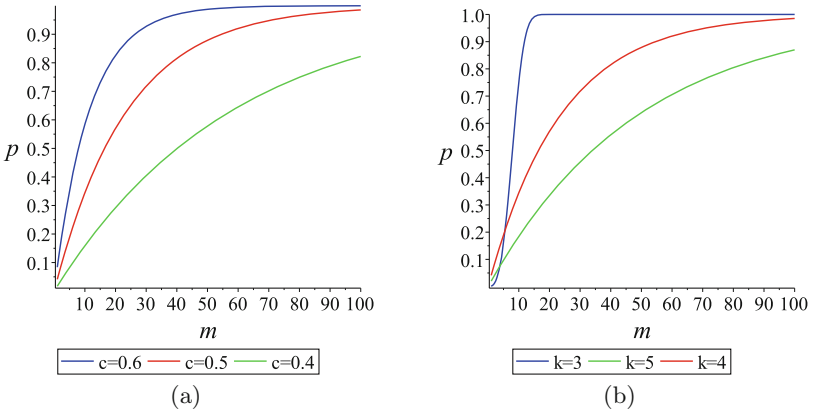


Fig. 6. (a) $p(m, c, 4)$ for different values of c; (b) $p(m, .5, k)$ for different values of k.

$p(m, .5, k)$ for $m = \{1, \ldots, 100\}$ and $k = \{3, 4, 5\}$. As expected, $p(m, c, k)$ increases with: (i) increasing m; (ii) increasing c; and (iii) decreasing k. One challenge here is that we do not know how to estimate k for a problem without actually solving it. In general, a smaller value of k means that a smaller portfolio will reach the target probability. In our experiments – recall Fig. 5 – we observed k-values in a tight range (1–10) for problems from HWMCC14. These numbers can serve as guidelines, and be refined based on additional experimentation.

6 Experimental Results

We implemented IC3SYNC, IC3ASYNC and IC3PROOF by modifying a publicly available reference implementation of IC3 (https://github.com/arbrad/IC3ref), which we call IC3REF. All propositional queries in IC3 are implemented by calls to MINISAT. We refer to the variant of IC3REF that uses a randomized MINISAT as IC3RND. We use IC3RND to introduce uncertainty in IC3 purely by randomizing the backend SAT solver. We performed two experiments – one to evaluate the effectiveness of the IC3PAR variants, and another to validate our statistical analysis of their portfolios. All our tools and results are available at http://www.andrew.cmu.edu/~schaki/misc/paric3.tgz.

Benchmarks. We constructed four benchmarks. The first was constructed by pre-processing the safe examples from HWMCC14 (http://fmv.jku.at/hwmcc14cav) with IIMC (http://ecee.colorado.edu/wpmu/iimc), and selecting the ones solved by IC3REF within 900 s on a 8 core 3.4 GHz machine with 8GB of RAM. The remaining three benchmarks were constructed similarly from the buggy examples from HWMCC14, and the safe and buggy examples (without pre-processing) from the TIP benchmark suite (http://fmv.jku.at/aiger/tip-aig-20061215.zip). We refer to the four benchmarks as HWCSAFE, HWCBUG, TIPSAFE and TIPBUG, respectively.

SAT Solver Pool. The function f (cf. Fig. 1–4) is implemented in IC3REF by a SAT solver (MINISAT). A separate SAT solver S_i is used for each $f(i)$. Whenever $f(i)$ changes due to the addition of a new lemma to a frame, the corresponding solver S_i is also updated by asserting the lemma. To avoid a single SAT solver from becoming the bottleneck between competing threads in IC3PAR, we use a "pool" of MINISAT solvers to implement each S_i. The solvers are maintained in a FIFO queue. When a thread requests a solver, the first available solver is given to it. When a lemma is added to the pool, it is added to all available solvers, and recorded as "pending" for the busy ones. When a busy solver is released by a thread, all pending lemmas are first asserted to it, and then it is inserted at the back of the queue. We refer to the number of solvers in each pool as $SPSz$.

6.1 Comparing Parallel IC3 Variants

These experiments were carried on a Intel Xeon machine with 128 cores, each running at 2.67 GHz, and 1TB of RAM. For each solver selected from $\{\text{IC3ASYNC}(4), \text{IC3SYNC}(4), \text{IC3PROOF}(4), \text{IC3RND}\}$ and each benchmark $\mathcal{B}$, and with $SPSz = 3$, we performed the following steps:

1. extract all problems from $\mathcal{B}$ that are solved by IC3REF in at least 10 s; call this set $\mathcal{B}^*$; the cutoff of 10 s was a tradeoff between problem complexity and benchmark size; our goal was to avoid evaluating our approach on very simple examples to limit measurement errors, and also to have enough examples for statistically meaningful results;
2. solve each problem in $\mathcal{B}^*$ with IC3REF and also with a portfolio of 20 solvers, compute the ratio of the two runtimes; this is the speedup for the specific problem;
3. compute the mean and max of the speedups over all problems in $\mathcal{B}^*$.

Figure 7(a) shows the results obtained. In all cases, we see speedup. On this particular run, IC3PROOF performs best overall, with an average speedup of over 6 and a maximum speedup of over 300. As in the non-portfolio case

		IC3SYNC		IC3ASYNC		IC3PROOF		IC3RND	
$\mathcal{B}$	$\lvert\mathcal{B}^*\rvert$	Mean	Max	Mean	Max	Mean	Max	Mean	Max
HWCSAFE	31	1.30	5.61	1.58	5.47	1.60	4.08	1.17	4.64
HWCBUG	14	2.49	18.7	14.3	151	25.1	309	1.07	1.49
TIPSAFE	14	1.28	4.50	2.61	11.1	2.29	12.8	1.37	3.80
TIPBUG	9	2.23	5.35	2.82	7.32	3.50	12.1	1.16	2.17
SAFE	44	1.30	5.61	1.93	11.1	1.83	12.8	1.24	4.64
BUG	23	2.38	18.7	9.58	151	16.3	309	1.11	2.17
ALL	67	1.67	18.7	4.74	151	6.79	309	1.19	4.64

(a)

		IC3PAR2010	
$\mathcal{B}$	$\lvert\mathcal{B}^+\rvert$	Mean	Max
HWCSAFE	20	2.67	14.40
HWCBUG	15	1.62	3.91
TIPSAFE	14	.89	1.82
TIPBUG	7	1.32	1.67
SAFE	34	1.94	14.40
BUG	22	1.52	3.91
ALL	56	1.77	14.40

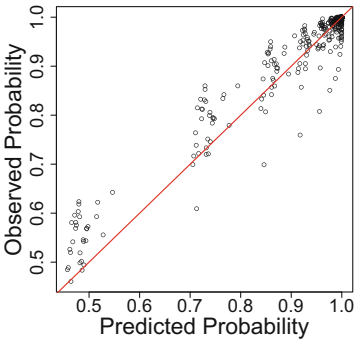
(b)

Fig. 7. (a) Speedup of IC3SYNC, IC3ASYNC, IC3PROOF and IC3RND compared to IC3REF; (b) Speedup of IC3PAR2010 compared to IC3REF2010.

(cf. Fig. 5) IC3PROOF and IC3ASYNC have similar performance, and are better than IC3SYNC. The pattern is followed for both safe and buggy examples. Finally, IC3RND provides mediocre speedup (not just on the whole, but across all examples) indicating that parallelization enables better search for shorter proofs/CEXes compared to randomizing the SAT solver.

	ρ - IC3ASYNC		ρ - IC3SYNC		ρ - IC3PROOF	
Example	Mean	StDev	Mean	StDev	Mean	StDev
6s286	1.0000	0.0016	1.0010	0.0046	0.9996	0.0032
intel026	1.0042	0.0233	1.0028	0.0163	1.0027	0.0163
6s273	1.0025	0.0122	1.0031	0.0149	1.0030	0.0154
intel057	0.9968	0.0214	0.9855	0.0381	1.0002	0.0136
intel054	1.0029	0.0162	0.9998	0.0076	0.9994	0.0080
6s215	1.0001	0.0057	0.9988	0.0099	0.9991	0.0058
6s216	1.0038	0.0204	1.0025	0.0163	1.0034	0.0182
oski3ub1i	1.0063	0.0321	1.0055	0.0293	1.0049	0.0274
oski3ub3i	1.0042	0.0230	1.0049	0.0259	1.0053	0.0272
oski3ub5i	1.0061	0.0312	1.0070	0.0358	1.0069	0.0357

(a)



(b)

Fig. 8. Validating Theorem 5; (a) mean and standard deviation of ratios of predicted and observed probabilities; (b) scatter plot of predicted and observed probabilities.

6.2 Comparing IC3PAR2010

We compared the parallel version of IC3 reported in the original paper [6], which we refer to as IC3PAR2010, with our IC3PAR variants. We first downloaded the source code[1] of IC3PAR2010. It comes with its own version of IC3 implemented in Ocaml, which we refer to as IC3REF2010. The parallelization in IC3PAR2010 is implemented via three Python scripts that invoke the IC3 binary. We modified these scripts to implement a solver with four copies of IC3 running in parallel. This was done for a fairer comparison with our IC3PAR results presented earlier which also used four copies of IC3 per solver. In addition, we made other changes to the scripts to make the solver more robust (e.g., replacing hard coded TCP/IP port numbers with dynamically selected ones). All experiments were done on the same machine as in Sect. 6.1.

While IC3REF was quite deterministic in its runtime, IC3REF2010 demonstrated random behavior in this respect. One source of this randomness is that IC3REF2010 randomizes the backend SAT solver. However, there could be other sources of randomness due to the management of the priority queue during `strengthen()`. We were unable to eliminate the randomness satisfactorily via command line options. Instead, we accounted for it by running experiments multiple times and computing the average. We computed the speedup of IC3PAR2010 using a similar process as for the IC3PAR variants. Specifically, we performed the following steps:

[1] http://ecee.colorado.edu/~bradleya/ic3/ic3.tar.gz.

1. extract all problems from $\mathcal{B}$ that are solved by IC3REF2010 in at least 10s; call this set $\mathcal{B}^+$; note that $\mathcal{B}^+$ differs from $\mathcal{B}^*$ since the underlying solvers – IC3REF2010 and IC3REF – have different solving capability.
2. solve each problem in $\mathcal{B}^+$ with IC3REF2010 twenty times and compute the average runtime (call this t_s) and also with IC3PAR2010 twenty times and compute the average runtime (call this t_p); compute the ratio $\frac{t_s}{t_p}$; this is the speedup with IC3PAR2010 for that specific problem;
3. compute the mean and max of the speedups over all problems in $\mathcal{B}^+$.

Figure 7(b) shows the results obtained. Comparing with Fig. 7(a), we see that all three of our IC3PAR invariants provided considerably better speedups compared to IC3PAR2010 on the three benchmark groups HWCBUG, TIPSAFE and TIPBUG. Indeed, for the TIPSAFE group as a whole, IC3PAR2010 does not provide a speedup. However, for the HWCSAFE group, IC3PAR2010 provided better speedups. If we look at all the safe examples, then IC3PAR2010 edges out IC3PAR marginally. In contrast, for unsafe examples IC3PAR provides much better speedups. Overall, IC3PROOF performs best. In summary, portfolios of IC3PAR variants appear to be a good complement to IC3PAR2010, and a better option for unsafe examples.

6.3 Portfolio Size

These experiments were done on a cluster of 11 machines, each with 16 cores at 2.4 GHz, and between 48 GB and 190 GB of RAM. To validate Theorem 5, we compared its predictions to empirically observed results as follows (again using $SPSz = 3$):

1. Process each problem from Fig. 5 as follows.
2. Solve the problem b (≈ 3000) times using IC3PAR(4). This gives a set of runtimes $t_1, \ldots, t_b$. Fit these runtimes to a Weibull distribution to obtain the estimated value of k (the same as the second column of Fig. 5).
3. Compute $\tilde{t} = \text{mean}(t_1, \ldots, t_b)$. This is the estimated average time for IC3PAR(4) to solve the problem.
4. Pick a portfolio size m. Start with $m = 1$.
5. Divide $t_1, \ldots, t_b$ into blocks of size m. Let $B = \lfloor \frac{b}{m} \rfloor$. We now have B blocks of runtime $T_1, \ldots, T_B$, each consisting of m elements. Thus, $T_1 = \{t_1, \ldots, t_m\}$, $T_2 = \{t_{m+1}, \ldots, t_{2m}\}$, and so on. For $i = 1, \ldots, B$, compute $\mu_i = \min(T_i)$. Note that each μ_i is the runtime of a portfolio of m IC3PAR(4) solvers.
6. Let $n(m)$ be the number of blocks for which $\mu_i \leq \tilde{t}$, i.e., $n(m) = |\{i \in [1, B] \mid \mu_i \leq \tilde{t}\}|$. Compute $p^*(m) = \frac{n(m)}{B}$. Note that $p^*(m)$ is the estimate of $p(m)$ based on our experiments. Compute $p(m) = 1 - (\alpha(k))^m$ (use k from Step 2). Compute $\rho(m) = \frac{p^*(m)}{p(m)}$. We expect $\rho(m) \approx 1$.
7. Repeat steps 5 and 6 with $m = 2, \ldots, 100$ to obtain the sequence $\rho = \langle \rho(1), \ldots, \rho(100) \rangle$. Compute the mean and standard deviation of ρ.

Figure 8(a) shows the results of the above steps for each IC3PAR variant. We see that for each example, the mean of ρ is very close to 1 and its standard

deviation is very close to 0, indicating that $p(m)$ and $p^*(m)$ agree considerably. Figure 8(b) shows a scatter plot of all $p^*(m)$ values computed against their corresponding $p(m)$. Note that most values are very close to the (red) $x = y$ line, as expected.

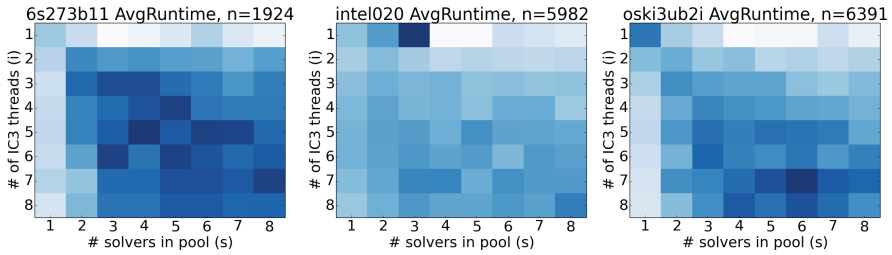


Fig. 9. Heatmap of IC3PROOF runtimes for three examples. Deeper color of cell (i, s) indicates that IC3PROOF(i, s) solves the benchmark faster; n = total number of runs of IC3PROOF over all 64 values of (i, s).

6.4 Parameter Sweeping

IC3PAR has two parameters: number of IC3 threads and $SPSz$. We write IC3PAR (i, s) to mean an instance of IC3PAR with i IC3 threads and $SPSz = s$. Thus, IC3PAR$(4, 3)$ was used is all previous experiments. We observed in Sect. 6.1 that different IC3PAR variants perform the best for different benchmarks. We now evaluate the performance of IC3PROOF by varying i and s. These experiments were also done on our cluster (cf. Sect. 6.3). We begin with a conjecture about the relationship of runtime and parameter values.

Conjecture 2. The parameter value affects performance of IC3PAR significantly, and the best parameter choice is located unpredictably in the input space.

To investigate Conjecture 2, we measured the runtime of IC3PROOF(i, s) for each $(i, s) \in I \times S$ where $I = S = \{1, \ldots, 8\}$. We selected 16 examples from $\mathcal{B}$. For each example η, and each $(i, s) \in I \times S$, we executed IC3PROOF(i, s) on η "around" 3000 times (again, the exact number varied across examples due to timeouts) and computed the average runtime. This gives us the entry at (i, s) for the "heatmap" for η. The heatmaps in Fig. 9 summarize our results for three of the benchmarks that we found to be representative. They support Conjecture 2, as average runtimes (indicated by the color depth of cells in the heatmaps) across the parameter space are varied. The depth of cells show no discernable pattern (e.g., do not increase with i or s), and the deepest cells are significantly more so than the lightest ones. This implies that: (i) selecting the best parameters for IC3PROOF would be quite beneficial; but (ii) this is a non-trivial problem.

As a preliminary step to address this challenge, we ran portfolios of a solver that uses parameter sweeping [2]. Specifically, the solver (denoted IC3SWEEP) executes a randomly selected IC3PAR variant with a random (i, s) selected from

	Time	Speedup			
Example	IC3REF	Sync	Async	Proof	Sweep
6s286	947.6	1.54	1.57	1.66	1.77
intel026	78.33	2.61	2.77	2.85	2.58
6s273	31.06	1.84	1.88	1.90	1.65
intel057	31.33	2.45	2.49	2.49	2.66
intel054	55.89	3.52	3.51	3.52	3.92
Mean		2.39	2.44	2.48	2.52

	Time	Speedup			
Example	IC3REF	Sync	Async	Proof	Sweep
6s215	12.20	2.47	2.57	2.61	2.36
6s216	67.24	4.33	4.35	4.30	4.29
oski3ub1i	83.64	1.90	1.97	1.94	1.96
oski3ub3i	79.41	1.90	1.94	1.99	1.94
oski3ub5i	127.3	2.66	2.65	2.67	2.76
Mean		2.65	2.70	2.70	2.66

Fig. 10. Parameter sweeping; Sync, Async, Proof, Sweep = average speedups over IC3REF for portfolios of 20 IC3SYNC (4,3), IC3ASYNC (4,3), IC3PROOF (4,3), and IC3SWEEP, respectively.

$I \times S$. We compared the average speedup (over 50 runs) of a portfolio of 20 IC3SWEEPs with the average speedup (over 50 runs) of portfolios of 20 of each of the three IC3PAR variants with fixed $(i, s) = (4, 3)$. Figure 10 summarizes our results. We observe that in general IC3SWEEP is competitive with each of the IC3PAR variants (indeed, it performs best for the hardest examples from the safe and buggy categories). We believe that parameter sweeping shows promise as a strategy for real-life problems where good parameters would be difficult (if not impossible) to compute.

7 Conclusion

We present three ways to parallelize IC3. Each variant uses a number of threads to speed up the computation of an inductive invariant or a CEX, sharing lemmas to minimize duplicated effort. They differ in the degree of synchronization and technique to detect if an inductive invariant has been found. The runtime of these solvers is unpredictable, and varies with thread-interleaving. We explore the use of portfolios to counteract the runtime variance. Each solver in the portfolio potentially searches for a different proof/CEX. The first one to succeed provides the solution. Using the Extreme Value theorem and statistical analysis, we construct a formula that gives us a portfolio size to solving a problem within a target time bound with a certain probability. Experiments on HWMCC14 benchmarks show that the combination of parallelization and portfolios yields an average speedups of 6x over IC3, and in some cases speedups of over 300. We show that parameter sweeping is a promising approach to implement a solver that performs well over a wide range of problems of unknown difficulty. An important area of future work is the effectiveness of parallelization and portfolios in the context of software verification via a generalization of IC3 [12].

Aknowledgment. We are grateful to Jeffery Hansen and Arie Gurfinkel for helpful comments and discussions. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution DM-0002752.

References

1. Albarghouthi, A., Kumar, R., Nori, A.V., Rajamani, S.K.: Parallelizing top-down interprocedural analyses. In: Vitek, J., Lin, H., Tip, F. (eds.) Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation (PLDI 2012), pp. 217–228. Association for Computing Machinery, Beijing, China, June 2012
2. Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U., Amarasinghe, S.P.: OpenTuner: an extensible framework for program autotuning. In: Amaral, J.N., Torrellas, J. (eds.) Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT 2014), pp. 303–316. Association for Computing Machinery, Edmonton, AB, Canada, August 2014
3. Barnat, J., et al.: DiVinE 3.0 - an explicit-state model checker for multithreaded C & C++ programs. In: Sharygina, N., Veith, H. (eds.) CAV. Lecture Notes in Computer Science, vol. 8044, pp. 863–868. Springer, Saint Petersburg (2013)
4. Bingham, B., Bingham, J., Erickson, J., de Paula, F.M., Reitblatt, M., Singh, G.: Industrial strength distributed explicit state model checking. In: Proceedings of the 9th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2010), Twente, The Netherlands, September-October 2010
5. Blom, S., van de Pol, J., Weber, M.: LTSmin: distributed and symbolic reachability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 354–359. Springer, Heidelberg (2010)
6. Bradley, A.R.: SAT-Based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)
7. Chaki, S., Karimi, D.: Model Checking with Multi-Threaded IC3 Portfolios (2016), Extended version of this paper. http://www.contrib.andrew.cmu.edu/~schaki/publications/VMCAI-2016-Extended.pdf
8. Cimatti, A., Griggio, A.: Software model checking via IC3. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 277–293. Springer, Heidelberg (2012)
9. Ditter, A., Ceska, M., Lüttgen, G.: On parallel software verification using boolean equation systems. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 80–97. Springer, Heidelberg (2012)
10. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: Proceedings of the 11th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2011), pp. 125–134. IEEE Computer Society, Austin, TX, October-November 2011
11. de Haan, L., Ferreira, A.: Extreme Value Theory: An Introduction. Springer, New York (2006)
12. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 157–171. Springer, Heidelberg (2012)
13. Holzmann, G.J.: Parallelizing the spin model checker. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 155–171. Springer, Heidelberg (2012)
14. Kadioglu, S., Malitsky, Y., Sabharwal, A., Samulowitz, H., Sellmann, M.: Algorithm selection and scheduling. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 454–469. Springer, Heidelberg (2011)

15. Lopes, N.P., Rybalchenko, A.: Distributed and predictable software model checking. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 340–355. Springer, Heidelberg (2011)
16. Malitsky, Y., Sabharwal, A., Samulowitz, H., Sellmann, M.: Boosting sequential solver portfolios: knowledge sharing and accuracy prediction. In: Nicosia, G., Pardalos, P. (eds.) LION 7. LNCS, vol. 7997, pp. 153–167. Springer, Heidelberg (2013)
17. Melatti, I., Palmer, R., Sawaya, G., Yang, Y., Kirby, R.M., Gopalakrishnan, G.C.: Parallel and distributed model checking in eddy. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 108–125. Springer, Heidelberg (2006)
18. Palikareva, H., Cadar, C.: Multi-solver support in symbolic execution. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 53–68. Springer, Heidelberg (2013)
19. Ppfolio website. http://www.cril.univ-artois.fr/~roussel/ppfolio
20. Weibull, W.: A statistical distribution function of wide applicability. ASME J. Appl. Mech. **18**(3), 293–297 (1951)
21. Wintersteiger, C.M., Hamadi, Y., de Moura, L.: A concurrent portfolio approach to SMT solving. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 715–720. Springer, Heidelberg (2009)
22. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: Satzilla: Portfolio-based algorithm selection for SAT. J. Artif. Intell. Res. (JAIR) **32**, 565–606 (2008)