Quantitative Synthesis for Concurrent Programs[⋆]

Pavol Černý[†], Krishnendu Chatterjee[†], Thomas A. Henzinger[†], Arjun Radhakrishna[†], and Rohit Singh[*]

IST Austria[†] IIT Bombay[*]

Abstract. We present an algorithmic method for synthesizing the optimal placement of synchronization constructs in concurrent programs. The input consists of a partial program with possibly some synchronization constructs missing, and a performance model. The quantitative synthesis problem is to automatically complete the program by transforming and adding synchronization constructs so that both correctness is guaranteed and worst-case (or average-case) performance is optimized. As is standard for shared memory concurrency, correctness is formalized "specification free", in particular, as race freedom and deadlock freedom. To capture different system architectures, we use a parametric performance model, specified as a weighted automaton that assigns different costs to actions such as locking, context switching, and memory and cache accesses.
For worst-case performance, we show that the problem is equivalent to 2-player graph games with quantitative limit-average objectives; and for average-case performance, it is equivalent to $2\frac{1}{2}$-player graph games (i.e., games with probabilistic transitions). In both cases, the optimal correct program is derived from an optimal strategy in the corresponding quantitative game. While the respective games are computationally expensive (NP-complete), we present an algorithmic method and implementation that works efficiently for concurrent programs and performance models of practical interest. We have implemented a prototype tool and used it successfully for synthesizing finite-state concurrent programs that exhibit different paradigmatic patterns in concurrency: optimistic execution, producer-consumer cooperation, and work sharing. In each case, the tool automatically synthesized the optimal correct program for several different performance models representing different architectures.

1 Introduction

Developing concurrent programs that fully harness the power of modern multi-core machines is a difficult and error-prone task, as witnessed by a number of errors found in published algorithms [3, 16], and in production code (see for example [4]). A very promising approach to the development of correct concurrent programs is *partial program synthesis*. The goal here is to allow the programmer to specify a part of her intent declaratively, by saying what needs to be done or what conditions need to be maintained. The synthesizer then constructs a program that satisfies the specification (see for example [21, 20, 23]). However, quantitative considerations, so far, have been largely missing from previous frameworks for partial synthesis. Thus, there is no way for a programmer to ask the synthesizer for a program that is not only correct, but also *efficient* with respect to a performance model. We claim that this considerably hinders the potential usability of synthesis. We support the claim with the following examples.

Motivating Examples: *Example 1.* Consider a *producer-consumer* program, where there are k producer and k consumer threads accessing a buffer of n cells. The threads can call the `store` procedure which checks if one of the cell of the buffer is empty, and if so, stores the input value into it. Similarly, the procedure `load` finds a nonempty cell, and returns a value in this cell. The programmer writes a partial program that implements the procedures sequentially, and specifies that at each control location a global lock or a cell-local lock can be

[⋆] Full version with appendices available at http://pub.ist.ac.at/∼aradha/QuantSynthFull.pdf taken. The task of the synthesizer is to construct a correct program (that is, a program with no data races). It is easy to see that there are at least two different ways of implementing correct synchronization. The first is to use a global lock, which locks the whole buffer. The second is to use cell-local locks, with each thread locking only the cell it is currently accessing. The second program allows more concurrent behavior, and could be better in many settings. However, if the cost of locks is high (relatively to the other operations), the program that uses global locking might be more efficient. In our experiments on a desktop machine, the fine-grained implementation was out-performed by the coarse-grained implementation by a factor of 3 for very natural parameters. Thus in order to construct an efficient program, the synthesizer has to have at least a rudimentary performance model representing the architecture of the system on which the program is to be run.

```
1: while(true) {
2:   lver:=gver;
3:   ldata=gdata;
4:   n  = choice(1..10);
5:   i = 0;
6:   while (i < n) {
7:    work(ldata); i++;
8:   }
9:   if (trylock(lock)) {
10:    if (gver==lver) {
11:      gdata = ldata;
12:      gver = lver+1;
13:      unlock(lock);
14:    } else { unlock(lock) }
15:} }
```

Fig. 1. Partial program — optimistic

Example 2. Second, consider the program in Figure 1. It uses the locking mechanism from the transactional memory manager TL2, and illustrates optimistic concurrency. The shared variables are `gdata` on which some operation (given by the function `work()`) is to be performed repeatedly, and `gver`, the version number of the data. Each thread has local variables `ldata` and `lver` that store local copies of these two variables. The example uses a standard pattern where data is read (line 3) and operated on (line 7) without acquiring any locks. When the data has to be written back, the shared data is locked (line 10) and it is checked that no other thread has changed the data since it had been read. This check is performed using the version number (line 11). If the global version number has not changed while the thread has been working on the local copy of the data, the new value of data is written back to the shared memory (line 12), and the global version number is increased (line 13). If the global version number has changed, the whole procedure has to be retried. The number of operations (calls to `work`) performed optimistically without writing back to shared memory can influence the performance significantly. For very optimistic approaches which perform many operations before writing back, there are a lot of retries and the performance drops. On the other hand, when only a few operations are performed optimistically, the data has to be written back often, which also might lead to a performance drop. Thus, the programmer would like to leave the task of finding the optimal number of operations to be performed optimistically to the synthesizer. This is done via the choice statement in line 4.

The partial program resolution problem. Our aim is to synthesize concurrent programs that are both correct and optimal with respect to a quantitative performance model. As quantitative performance requirements were not present in previous approaches, we introduce a flexible framework for specifying performance models for concurrent programs. The input for synthesis consists of (1) a finite-state partial program, (2) a performance model, (3) a model of a scheduler, (4) a correctness condition. A *partial program* is a finite-state concurrent program which includes non-deterministic choices which the synthesizer has to resolve. We say that a program is *allowed* by a partial program if it can be obtained by resolving the nondeterministic choices. The second input to the synthesis problem is a *performance model*, given by a weighted automaton. The automaton assigns different costs to actions such as locking, context-switching, or memory and cache access. It is a flexible model that allows assigning costs based on past sequences of actions. For instance, if a context-switch happens soon after the preceding one, then its cost might be lower due to cache effects. Similarly, we can use the model to specify complex cost models for memory and cache access. Note that the performance model can be fixed for a particular architecture and hence, need not be constructed separately for every partial program. The fourth input is a *scheduler*. Our schedulers are state-based models, and hence support flexible scheduling schemes (e.g., a thread waiting for long may be scheduled with higher probability). In performance analysis, the average-case analysis is as natural as worst-case analysis. For the average-case (randomized) analysis, a probabilistic scheduler is needed. The fourth input, the *correctness condition* is a safety condition. We use "specification-free" conditions such as data-race freedom or deadlock-freedom.

The output of synthesis is a program that is (a) allowed by the partial program, (b) correct with respect to the safety condition, and (c) has the best performance of all the programs satisfying (a) and (b) with respect to the performance and scheduling models.

Methods: Quantitative Games. We show that the partial program resolution problem is equivalent to *imperfect information* stochastic graph games with quantitative (limit-average or mean-payoff) objectives. Traditionally, imperfect information games have been studied to answer the question of existence of general *history-dependent* optimal strategies, and then the problem becomes undecidable for quantitative objectives [9]. We show that the partial program resolution problem gives rise to a new game theoretic question that asks for the existence of *memoryless* optimal strategies (strategies that are independent of the history) in imperfect information games. We establish that the memoryless problem for imperfect information stochastic games is NP-complete, and show that the partial program resolution problem is NEXPTIME-complete for both average and worst-case performance based synthesis.

We present several techniques that overcome the theoretical difficulty of NEXPTIME-hardness in cases of programs of practical interest. We present three practical techniques for the partial program resolution problem: (1) First, we use a lightweight static analysis technique for efficiently eliminating parts of the strategy tree for Player 1. This reduces the number of strategies to be examined significantly. For each strategy that we need to examine, we obtain a (perfect information) Markov decision process (MDP). For MDPs, efficient strategy improvement algorithms exist that require solving Markov chains. (2) Second, Markov chains obtained for concurrent programs typically satisfy certain progress conditions, and we exploit that to develop a forward propagation technique along with Gaussian elimination to solve Markov chains efficiently. (3) The third technique is to use an abstraction that preserves the value of the quantitative (limit-average) objective. An example of such an abstraction is the classical data abstraction.

Experimental results. In order to evaluate our synthesis algorithm, we have implemented a prototype tool and applied it to four finite-state examples that illustrate basic patterns in concurrent programming. In each case, the tool automatically synthesized the optimal correct program for various performance models that represent different architectures. The running time of the tool was under a minute for all cases but one, where the running time was under five minutes.

For the producer-consumer example, we synthesized from a partial program where two producer and two consumer threads are accessing a buffer with four cells. The most important parameters of the performance model are the cost of locking/unlocking l and the cost c of copying data from/to shared memory. If the cost c was higher than l (by a factor 100:1), then the fine-grained locking approach is better (by 19 percent), and would be the result of synthesis. If the cost l is equal to c, then the coarse-grained locking approach was found to perform better (by 25 percent), and thus the coarse-grained program would be the result of the synthesis.

Referring back to the code in Figure 1, for the optimistic concurrency example and a particular performance model, the analysis found that increasing n improves the performance initially, but after a small number of increases (3) the performance started to decrease. We have measured the running time of the program on a desktop machine, and observed the same phenomenon.

Summary. To summarize, our main contributions are: (1) we develop a technique for synthesizing concurrent programs that are both correct and *optimal*; (2) we introduce a parametric performance model, and thus provide a flexible framework for specifying performance characteristics of architectures; (3) (a) we show how to use a imperfect information game theoretic framework to model the synthesis of concurrent programs (to the best of our knowledge, this is the first application of imperfect information games for synthesis of concurrent programs), and (b) establish optimal complexity results for new game-theoretic problems; (4) (a) we present practical techniques to efficiently solve partial program synthesis, and (b) implement a prototype and apply it to several case studies which illustrate common patterns in concurrent programming.

Related work. The problem of synthesis from specifications was originally posed by Church [7]. Synthesis for synchronization constructs is also an old problem and the celebrated paper [8] presented an algorithm for synthesis of synchronization skeletons. Synthesis of reactive systems was considered in [18]. In contrast to our work, all these works focused on qualitative synthesis without any performance measure. Recent works have considered quantitative synthesis [2, 5]; however the focus of these works has been the synthesis of sequential systems from temporal logic specifications. Moreover, all these works consider perfect information games. Neither imperfect information games nor quantitative objectives were considered before for synthesis for concurrent programs. We require imperfect information due to concurrency and quantitative objectives for performance measures.

Sketching is a technique where a partial implementation, with some nondeterministic choices, of a program is given and a correct program is generated automatically. Sketching [21, 19] for concurrent programs has been studied in [20]. However, none of the above works consider performance-aware algorithms for sketching. Abstract interpretation based synthesis was presented in [23], and is only optimal with respect to the number of interleavings, which might not translate into being optimal in performance. Automated lock placement has also been considered in literature. In [10], no user-specified performance objectives are considered. The paper [6] optimizes lock placement always in favor fine-grained locking, which again might not be optimal for performance for all programs on all architectures. We show that the synthesis problem for concurrent programs in general is equivalent to imperfect information games. However, none of the above works consider the general framework of games for synthesis, or the parametric performance model.

2 The Quantitative Synthesis Problem

2.1 Partial Programs

In this section we define threads, partial programs, programs and their semantics. We start with the definition of guards and operations.

Guards and operations. Let L, G, and I be finite sets of variables (representing local, global (shared), and input variables, respectively) ranging over finite domains. A *term* t is either a variable in L, G, or I, or is defined by $t_1\ op\ t_2$, where t_1 and t_2 are terms and op is an operator. Formulas are defined by the following grammar, where t_1 and t_2 are terms and rel is a relational operator: $e\ :=\ t_1\ rel\ t_2 \mid e_1\ \wedge\ e_2 \mid \neg e$. *Guards* are formulae over L, G, and I. *Operations* are simultaneous assignments to variables in L and G, where each variable is assigned a term over L, G, and I.

Threads. A *thread* is a tuple $\langle Q, L, G, I, \delta, \rho_0, q_0 \rangle$, where:

- Q is a finite set of control locations and q_0 is an initial location;
- L, G and I are finite sets of local, global, and input variables respectively, ranging over finite domains;
- ρ_0 is the initial valuation of variables in L, G, and I; and

- δ is a set of tuples of the form (q, g, a, q'), where q and q' are locations from Q, and g and a are *guards* and *operations* over variables in L, G and I.

The set of *sketch locations* $Sk(C)$ of a thread $C = \langle Q, L, G, I, \delta, \rho_0, q_0 \rangle$ is the subset of Q containing exactly locations where δ is non-deterministic, i.e., locations q where for a valuation of variables in L, G and I, there is more than one transition whose guard evaluates to true.

Partial programs and programs. A *partial program* M is a set of threads that have the same set of global variables G and whose initial valuation of variables in G is the same. Informally, the semantics of a partial program is a parallel composition of threads. The set G represents the shared memory. A *program* is a partial program, where the set $Sk(c)$ of each thread c is empty. A program P is *allowed* by a partial program M if it can be obtained by removing the outgoing transitions from sketch locations of all the threads of M, so that at most one transition from every location is enabled for a given valuation of variables from L, G and I, i.e., for a program all component threads are deterministic.

The guarded operations allow us to easily model basic concurrency constructs such as lock (where lock is modeled as a variable in G and locking/unlocking is done using guarded operations) and compare-and-set. The control structures inside a thread are not limited. A partial program is a collection of threads, thus thread creation is not supported.

Semantics. A *transition system* is a tuple $\langle S, A, \Delta, s_0 \rangle$ where S is a finite set of states, A is a finite set of actions, $\Delta \subseteq S \times A \times S$ is a set of transitions and s_0 is the initial state. The semantics of a partial program M is given in terms of a transition system, and we denote the transition system of a partial program M as $\mathsf{Tr}(M)$. Given a partial program M with n threads, let $\mathcal{C} = \{1, \dots, n\}$ represent the set of threads of M.

- *State space.* A state s in the set of states S of $\mathsf{Tr}(M)$ contains a location per thread in $\mathcal{C}$, a valuation of all input and local variables of each thread in $\mathcal{C}$, and a valuation of the global variables. In addition, the state s contains a value σ from $\mathcal{C} \cup \{*\}$, indicating which (if any) thread is currently scheduled. The initial state contains the initial location for each thread in $\mathcal{C}$, the initial valuation ρ_0 of all variables, and the value $*$ indicating that no thread is currently scheduled.
- *Transition.* The transition function Δ defines interleaving semantics for the partial program. There are two types of transitions: thread transitions, which model one step of a scheduled thread, and environment transitions, that model input from the environment and the scheduler. Let c be a thread in $\mathcal{C}$, and let (q, g, a, q') be a transition of c. There is a thread transition labeled by c from a state s to a state s' if and only if (i) the value σ in s is c (indicating that the thread c is scheduled) and the value σ in s' is $*$, (ii) the location of C in s is q and the location of C in s' is q', (iii) the guard g evaluates to true in s, and (iv) the valuation of local and input variables of C in s, and of global variables in s is obtained from the valuation of variables in s' by performing the operation a. There is an environment transition labeled by c from state s to state s' in $\mathsf{Tr}(M)$ if and only if (i) the value σ in s is $*$ and the value σ in s' is c and (ii) the valuations of variables in s and s' differs only in input variables of the thread c.

2.2 The performance model

We define a flexible and expressive performance model via a weighted automaton that specifies costs of actions. A *performance automaton* W is a tuple $W = (Q_W, \Sigma, \delta, q_0, \gamma)$, where Q_W is a set of states, Σ is a finite alphabet, $\delta \subseteq Q_W \times \Sigma \times Q_W$ is a transition relation, q_0 is an initial location and γ is a cost function $\gamma : Q_W \times \Sigma \times Q_W \to \mathbb{Q}$. The labels in Σ represent (concurrency-related) actions that incur costs, while the values of the function γ specify these costs. The symbols in Σ are matched with the actions (edge symbols) performed by the system to which the performance measures are applied. There is a special symbol in Σ, denoted by ot, that signifies that no action of the ones we are tracking occurred. The costs that can be specified in this way include for example the cost of locking, the access to the (shared) main memory or the cost of context switches.

An example specification that uses the costs mentioned above is the automaton W in Figure 2. The automaton describes the costs for locking (l), context-switching (cs), and main memory access (m). Specifying the costs via a weighted automaton is more general than only specifying a list of costs. For example, the fact the automaton based specification enables us to model a cache, and the cost of reading from a cache versus reading from the main memory, as shown in Figure 5 in Section 5. Note that the performance model is fixed for a particular architecture. This eliminates the need to construct a performance model for the synthesis of each partial program.

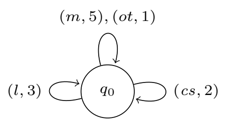

Fig. 2. Perf. Aut.

2.3 The partial program resolution problem

Weighted probabilistic transition system (WPTS). A *probabilistic transition system* (PTS) is a generalization of a transition system with probabilistic transition function. Formally, let $\mathcal{D}(S)$ denote the set of probability distributions over S. A PTS consists of a tuple $\langle S, A, \Delta, s_0\rangle$ where S, A, s_0 are defined as for transition systems, and $\Delta : S \times A \to \mathcal{D}(S)$ is probabilistic, that is, given a state and an action, it returns a probability distribution over successor states. A WPTS consists of a PTS and a weight function $\gamma : S \times A \times S \to \mathbb{Q} \cup \{\infty\}$ that assigns costs (given as rational values or infinite cost) to transitions. An *execution* of a probabilistic weighted transition system is an infinite sequence of the form $(s_0 a_0 s_1 a_2 \ldots)$ where s_i is in S, a_i is in A, and $\Delta(s_i, a_i)(s_{i+1}) > 0$, for all $i \geq 0$. We now define boolean and quantitative objectives for WPTS.

Safety objectives. A *safety objective* Safety_B is defined by a set B of "bad" states. It requires that states in B are never reached by a program execution. An execution $e = (s_0 a_0 s_1 a_2 \ldots)$ is *safe* (denoted by $e \in \mathrm{Safety}_B$) if $s_i \notin B$, for all $i \geq 0$.

Limit-average and Limit-average safety objectives. The *limit-average* objective is a quantitative objective that assigns a real-valued number to every infinite execution e. For an execution $e = (s_0 a_0 s_1 a_1 s_2 \ldots)$, we have $\mathrm{LimAvg}_\gamma(e) = \limsup_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n} \gamma((s_i, a, s_{i+1}))$ if there is no infinite cost transition, and ∞ otherwise. The *limit-average safety* objectives are a *lexicographic* combination of safety and limit-average objectives: the objective is defined by a weight function γ, and a set B of bad states. For an execution e, we have that if $e \in \mathrm{Safety}_B$ (that is e never visits the set B), then $\mathrm{LimAvg}_\gamma^B(e)$ is $\mathrm{LimAvg}_\gamma(e)$, otherwise it is ∞ (if the safety objective is satisfied, then we have the limit-average value, ∞ otherwise). The limit-average safety objective can be reduced to limit-average objectives by making the states in B absorbing (states with only self-loops) and assign them weight ∞.

Value of WPTS. We now define the *value* of a WPTS given a limit-average safety objective. Given a WPTS T with weight function γ, a *policy* $pf : (S \times A)^* \times S \to A$ is a function that given a sequence of states and actions chooses an action. A policy pf defines a unique probability measure on the executions and we denote by $E^{pf}(\cdot)$ the associated expectation measure. Given a WPTS T with weight function γ, and a policy pf, the value $\mathit{Val}(T, \gamma, \mathrm{Safety}_B, pf)$ is the expected value $E^{pf}(\mathrm{LimAvg}_\gamma^B)$ of the limit-average safety objective. The value of the WPTS is then defined as the supremum over all possible policy functions, i.e., $\mathit{Val}(T, \gamma, \mathrm{Safety}_B) = \sup_{pf} \mathit{Val}(T, \gamma, \mathrm{Safety}_B, pf)$.

Schedulers. A *scheduler* has a finite set of internal memory states Q_{Sch}. At each step, it considers all the active threads and chooses one either (i) non-deterministically (non-deterministic schedulers) or (ii) according to a probability distribution (probabilistic schedulers), which depends on the current internal memory state.

Composing a program with a scheduler and a performance model. In order to compute the value of a program allowed by a partial program, we need to take into account the scheduler and the performance model. Let P be a program, Sch be a scheduler, and W a performance model. We construct a PTS, denoted $\mathsf{Tr}(P, \mathrm{Sch}, W)$, with a weight function γ as follows. A state s in the set of states S of $\mathsf{Tr}(P, \mathrm{Sch}, W)$ is composed of a state of $\mathsf{Tr}(P)$ (where $\mathsf{Tr}(P)$ is the transition system of P), a state of the scheduler Sch and a state of W. The transition function matches environment transitions of $\mathsf{Tr}(P)$ with the scheduler transitions (which allows the scheduler to schedule threads) and it matches thread transitions with the performance model transitions. The weight function γ assigns costs to edges as given by the weighted automaton W. Furthermore, as the limit average objective is defined only for infinite executions, for terminating safe executions of the program we add an edge back to the initial state. The value of the limit average objective function of the infinite execution is the same as the average over the original finite execution. Note that the performance model can specify a locking cost, while the program model does not specifically mentioned locking. We thus need to specifically designate which shared memory variables are used for locking.

Correctness. We restrict our attention to safety conditions for correctness. We illustrate how various correctness conditions for concurrent programs can be modelled as Safety objectives: (a) *Data-race freedom.* Data-races occur when two or more threads access the same shared memory location and one of the accesses is a write access. Absence of data races (data-race freedom) is a safety property. The unsafe states are those where there are two different threads, each having an enabled transition accessing a particular shared variable, and at least one of these transitions writes to these variables. (b) *Deadlock freedom.* One of the major problems of synchronizing programs using blocking primitives such as locks is that deadlocks may arise. A deadlock occurs when two (or more) threads are waiting for each other to finish an operation. Deadlock-freedom is a safety property. A deadlock can be detected by checking for cycles in a wait graph, that is, a graph whose nodes represent threads and edges indicate whether a thread waits for a resource (a lock) held by another thread.

Value of a program and of a partial program. Given a program P, together with a scheduler Sch, a performance model W, and a safety objective Safety_B, the value of the program, defined using the PTS $\mathsf{Tr}(P, \mathrm{Sch}, W)$ and the weight function γ, is as follows: $\mathit{ValProg}(P, \mathrm{Sch}, W, \mathrm{Safety}_B) = \mathit{Val}(\mathsf{Tr}(P, \mathrm{Sch}, W), \gamma, \mathrm{Safety}_B)$. (Note that the value is defined using the limit-average safety objective. The safety part is defined by the parameter Safety_B, while the quantitative part is defined by the costs defined by the performance model W.) Let M be a partial program and let $\mathcal{P}$ be the set of all programs which are allowed by the partial program M. Given a scheduler Sch, a performance model W, and a safety objective Safety_B, the value for the partial program is defined by $\mathit{ValParProg}(M, \mathrm{Sch}, W, \mathrm{Safety}_B) = \min_{P \in \mathcal{P}} \mathit{ValProg}(P, \mathrm{Sch}, W, \mathrm{Safety}_B)$.

Partial Program resolution problem. The central technical questions we address are as follows: (1) The *partial program resolution optimization problem* consists of a partial program M, a scheduler Sch, a performance model W and a safety condition Safety_B, and asks for a program P allowed by the partial program M such that the value $\mathit{ValProg}(P, \mathrm{Sch}, W, \mathrm{Safety}_B)$ is minimized. Informally, we have: (i) if the value $\mathit{ValParProg}(M, \mathrm{Sch}, W, \mathrm{Safety}_B)$ is ∞, then it means no safe program exists; (ii) if it is less than ∞, then the answer is the optimal safe program, i.e. a program that is correct and optimal with respect to the performance model. The *partial program resolution decision problem* consists of the above inputs and a rational threshold λ, and asks whether $\mathit{ValParProg}(M, \mathrm{Sch}, W, \mathrm{Safety}_B) \leq \lambda$.

3 Quantitative Games on Graphs

The use of games for controller synthesis and synthesis of sequential systems from specifications have been well studied in literature. In this section, we show how the partial program resolution problems can be solved through quantitative imperfect information games on graphs. Moreover we show that the technical questions on game graphs for partial program resolution is different from the classical problems studied for quantitative graph games. We start with the basic definitions about games on graphs.

3.1 Imperfect information games with quantitative objectives

A *imperfect information stochastic game graph* is a tuple $\mathcal{G} = \langle S, A, \mathrm{En}, \varDelta, (S_1, S_2), O, \eta, s_0 \rangle$, where S is a finite set of states, A is a finite set of actions, $\mathrm{En} : S \to 2^A \setminus \emptyset$ is the action enabledness function that assigns to every state s the non-empty set $\mathrm{En}(s)$ of actions enabled at s, and s_0 is an initial state. The transition function $\varDelta$ is a probabilistic function $\varDelta : S \times A \to \mathcal{D}(S)$ that, given a state s and an enabled action a gives the probability distribution $\varDelta(s, a)$ over the successor states. The sets (S_1, S_2) define a partition of S into Player-1 and Player-2 states, respectively; and O is a finite set of observations. The function $\eta : S \to O$ maps every state to an observation. We will refer to these games as ImpIn $2\frac{1}{2}$-player game graphs: ImpIn for imperfect information, 2 for the two players and $\frac{1}{2}$ for the probabilistic transitions.

The informal semantics for a imperfect information game is as follows: the game starts with a token being placed on the initial state. In each step, Player 2 can observe the exact state s in which the token is placed whereas, Player 1 can observe only $\eta(s)$. If the token is in S_1 (resp. S_2), Player 1 (resp. Player 2) chooses an action a enabled in s. The token is then moved to a successor of s based on the distribution $\varDelta(s, a)$. To formalize these semantics, we define the notion of strategies.

Strategies. Informally, a strategy is "recipe" to play a ImpIn $2\frac{1}{2}$-player game. A *Player 1 strategy* is a function $\sigma : (O \times A)^* \cdot O \to A$, which picks an enabled action for Player 1 given the observable history of the game. A *Player 2 strategy* is a function $\tau : (S \times A)^* \cdot S \to A$, which picks an enabled action for Player 2 given the history of the game. A strategy must always choose among enabled actions. Observe that Player 1 has imperfect information, whereas Player 2 has perfect-information. A Player 1 strategy is called *memoryless* if $\sigma(h_1 \cdot o) = \sigma(h_2 \cdot o)$ for all $h_1, h_2 \in (O \times A)^*$. Similarly, a Player 2 strategy is memoryless if $\tau(h_1 \cdot s) = \tau(h_2 \cdot s)$ for all $h_1, h_2 \in (S \times A)^*$. A memoryless strategy for Player 1 (resp. Player 2) can be simply represented as functions $O \to A$ (resp. $S \to A$). We denote the set of Player 1 and Player 2 strategies by $\varSigma$ and $\varGamma$, respectively, and the set of Player 1 and Player 2 memoryless strategies by $\varSigma^M$ and $\varGamma^M$, respectively.

Special cases. We will also consider the following special cases of ImpIn $2\frac{1}{2}$-player games:

- ImpIn 2-player games are the special case when the transition function is deterministic, i.e., $\varDelta(s, a)(s') = 1$ for some $s' \in S$.
- Perfect information games where both player have perfect information and can view the states, i.e., every state has an unique observation, $O = S$, and $\eta(s) = \{s\}$.
- $1\frac{1}{2}$-player games (or Markov decision processes) are the special case of ImpIn $2\frac{1}{2}$-player games, where every Player 1 state has exactly one action enabled, i.e., $|\mathrm{En}(s)| = 1$ for all $s \in S_1$ (recall that Player 2 has perfect information).
- Markov chains are special cases of MDPs where all states have exactly only one action enabled.

Paths and probability space. Given a game graph $\mathcal{G}$, we denote by $\varPi_{\mathcal{G}}$ the set of paths of the game graph $\mathcal{G}$. Given a Player 1 strategy σ and a Player 2 strategy τ, we denote by $\mathrm{Pr}^{\sigma,\tau}(\cdot)$ the unique probability measure over the set of paths of the game graph. For details on the probability measure, refer to any standard work on $2\frac{1}{2}$-player stochastic games (for example, [22]).

Objectives. In a graph game, we consider *boolean* or *quantitative* objectives for both players. A *boolean objective* for a game graph $\mathcal{G}$ is a function from $\phi : \Pi_{\mathcal{G}} \to \{0,1\}$ and a *quantitative objective* is a function $f : \Pi_{\mathcal{G}} \to \mathbb{R}$. We will consider boolean objectives defined by safety objectives, and quantitative objectives defined by limit-average and limit-average safety objectives.

Values. The goal of Player 1 in a game with game graph $\mathcal{G}$ and boolean or quantitative objective f, is to minimize the expected value of f whereas, the goal of Player 2 is to maximize it. In this report, we only consider the LimAvg-Safety objectives which were defined in Section 2. Formally, the *value* of a Player 1 strategy σ for an objective function f is defined as $\mathit{ValGame}(f,\mathcal{G},\sigma) = \sup_{\tau \in \Gamma} \mathbb{E}^{\sigma,\tau}[f]$. The *value* of the game is defined as $\mathit{ValGame}(f,\mathcal{G}) = \inf_{\sigma \in \Sigma} \mathit{ValGame}(f,\mathcal{G},\sigma)$.

Decision problems. We now present the decision problems for ImpIn $2\frac{1}{2}$ and 2-player game graphs, and study the complexity. Given a game graph $\mathcal{G}$, an objective f and a rational threshold $q \in \mathbb{Q}$, the general decision problem (resp. memoryless decision problem) asks whether there is a strategy (resp. memoryless strategy) σ for Player 1 such that $\mathit{ValGame}(f,\mathcal{G},\sigma) \leq q$. Similarly, the value problem (memoryless value problem) is to compute $\inf_{\sigma \in \Sigma} \mathit{ValGame}(f,\mathcal{G},\sigma)$ ($\min_{\sigma \in \Sigma^M} \mathit{ValGame}(f,\mathcal{G},\sigma)$ resp.). The traditional game theory study always considers the general decision problem which is undecidable for limit-average objectives [9] in imperfect information games.

Theorem 1. *[9] The decision problems for* LimAvg *and* LimAvg-Safety *objectives are undecidable for* ImpIn $2\frac{1}{2}$*- and* ImpIn 2*-player game graphs.*

3.2 From partial programs to games

In the previous section we have seen that the general decision problem is undecidable. However, we show here that the partial program resolution problems reduce to the memoryless decision problem for imperfect information games. In Section 3.3, we establish the complexity of the memoryless decision problem for imperfect information games with LimAvg and LimAvg-Safety objectives.

Theorem 2. *Given a partial program M, a scheduler* Sch*, performance model W, and a correctness condition ϕ: We construct an exponential size* ImpIn $2\frac{1}{2}$*-player game graph $\mathcal{G}_M^p$ with a* LimAvg-Safety *objective such that the memoryless value of $\mathcal{G}_M^p$ is equal to* $\mathit{ValParProg}(M, \mathrm{Sch}, W, \mathrm{Safety})$.

Proof. The proof relies on the construction of a imperfect information game graph, denoted $\mathcal{G}(M, \mathrm{Sch}, W)$, in which fixing a memoryless strategy σ for Player 1 yields a WPTS $\mathsf{Tr}(P_\sigma, \mathrm{Sch}, W)$ with weight function γ that corresponds to the product of a program P_σ allowed by the partial program M, composed with the scheduler Sch and the performance model W. The construction of this game graph is similar to the construction of the product of a program, scheduler and performance model, but with a partial program replacing the program. Due to the non-deterministic transition function of the partial program, there will exist extra non-deterministic choices in the WPTS (in addition to the choice of inputs). This non-determinism is resolved by Player 1 choices and the non-determinism due to input (and possibly scheduling) is resolved by Player 2 choices. We refer to this game as the *program resolution game*.

The crucial point of the construction is the observations, i.e., the information about the state that is visible to Player 1. Since Player 1 is to resolve the non-determinism from the partial program, he is allowed only to observe the scheduled thread and its current location.

He may choose a set of transitions, from that location, such that only one of the set is enabled for any valuation of the variables. The formal description is given in the full version.

To complete the proof, we show that given a memoryless Player 1 strategy σ, there exists a program P_σ allowed by M such that $\mathsf{Tr}(P_\sigma, \mathrm{Sch}, W)$ corresponds to the MDP obtained by fixing σ in $\mathcal{G}(M, \mathrm{Sch}, W)$ and vice-versa.

Given a program P_σ allowed by the partial program, we construct a memoryless σ as follows: $\sigma((t, q))$ is the action consisting of the set of transitions from location q in thread t in P_σ. As P_σ is deterministic, only one of them will be enabled for a valuation of the variables. Similarly, given a memoryless Player 1 strategy, we construct P_σ by preserving only those transitions from location q of thread t which are present in $\sigma((t, q))$. From the above construction we conclude the desired correspondence. □

3.3 Complexity of ImpIn Games and partial program resolution

We now study the complexity of the memoryless decision problems for ImpIn $2\frac{1}{2}$- and ImpIn 2-player game graphs. The problem has not been studied before in the literature. We have shown in Theorem 2 that it is the relevant problem for partial program resolution. We will first prove that the memoryless decision problems for imperfect information games are NP-complete. Using the NP-membership of these problems, we establish an upper bound for the partial program resolution problems. We establish a matching lower bound and thereby establish precisely the computational complexity of the partial program resolution problems. Firstly, for the special case of MDPs, the answer to the decision and memoryless decision problems coincide and can be solved in polynomial time [11] (using linear-programming to solve MDPs with safety and limit-average objectives).

Theorem 3. *[11] The memoryless decision problem for* LimAvg-Safety *objectives can be solved in polynomial time for MDPs.*

Theorem 4. *(Complexity). The memoryless decision problems for* Safety, LimAvg, *and* LimAvg-Safety *objectives are* NP*-complete for* ImpIn $2\frac{1}{2}$*- and* ImpIn 2*-player game graphs.*

For the lower bound we show a reduction from 3SAT problem and for the upper bound we use memoryless strategies as polynomial witness and Theorem 3 for polynomial time verification procedure (details in the full version).

Remark 1. We observe that the NP-completeness of the memoryless decision problems rules out the existence of the classical strategy improvement algorithms. The reason is as follows: the existence of strategy improvement algorithm also implies existence of randomized sub-exponential time algorithms (using the techniques of [1]), and hence a strategy improvement algorithm for the memoryless decision problem would imply a randomized sub-exponential upper bound for a NP-complete problem.

Theorem 5. *The partial program resolution decision problem is* $\mathrm{NEXPTIME}$*-complete for both non-deterministic and probabilistic schedulers.*

Proof. (a) The $\mathrm{NEXPTIME}$ upper bound is as follows: an exponential reduction to the memoryless decision problem for imperfect information games follows from Theorem 2, and then by Theorem 4 the $\mathrm{NEXPTIME}$ upper bound follows.

(b) We reduce the $\mathrm{NEXPTIME}$-hard problem *succinct 3-*SAT (see [17]) to the partial program resolution problem to show that it is $\mathrm{NEXPTIME}$-hard, which will complete the proof. The idea of the proof is to construct a two thread partial program in which one of the threads will choose a clause from the satisfiability. The second thread will determine the literals in this clause and then, check whether the clause is satisfied. If it is not, the thread enters an error state. The safety condition is to stay away from the error state.

Suppose we are given an instance of succinct 3-SAT over variables $v_1, v_2, \ldots v_M$, i.e., a circuit $\mathcal{Q}$ which takes pairs (i, j) ($i \in \{1, 2, \ldots, N-1\}$ and $j \in \{1, 2, 3\}$) as inputs and returns the literal in the j^{th} position in the i^{th} clause. The first thread of the partial program just changes the global variable i, looping through all values in $\{1, 2, \ldots, N-1\}$.

```
GLOBALS: var i;

THREAD 1:
while (true)
     i = (i + 1) mod N;

THREAD 2:
choice: {
     val[v1] = true;
     val[v1] = false;
}
...

while (true)
     l1 = compute_Q(i,1);
     l2 = compute_Q(i,2);
     l3 = compute_Q(i,3);
     if(not (val[l1] ∨
             val[l2] ∨
             val[l3]))
          assert(false);
```

Fig. 3. The reduction of succinct 3-SAT to partial program resolution

The second thread will first have a sequence of partial program locations where it non-deterministically chooses a valuation $\mathcal{V}$ for all literals. It then does the following repeatedly: (a) Read global i, (b) Compute the i^{th} clause by solving the circuit value problem for $\mathcal{Q}$ with $(i, 1)$, $(i, 2)$ and $(i, 3)$ as inputs. (c) If the i^{th} clause is not satisfied with the valuation $\mathcal{V}$, it goes to an error state.

As the circuit value can be solved in polynomial time, we can construct code (of polynomial size in $\mathcal{Q}$) which will read a global variable i and compute the output of $\mathcal{Q}$. The code for both threads is shown in Figure 3.

Now, to show the validity of the reduction: if there exists a valuation satisfying the formula generated by $\mathcal{Q}$, choosing that valuation in the first step in the second thread will obviously generate a safe program. If there exists no such valuation, for any valuation $\mathcal{V}$ chosen in the partial program, there exists a clause (say k) which is not satisfied. To generate an unsafe behavior we let the first thread run till i becomes equal to k and then let the second thread run which will obviously enter the error state. Observe that the hardness result is independent of the scheduler, and in particular holds for non-deterministic and probabilistic schedulers. Note that our hardness result uses only safety objective and no weighted automaton for limit-average objectives. □

4 Practical Solutions for Partial Program Resolution

In the previous section, we showed that the partial program resolution problem is computationally hard (NEXPTIME-complete). In this section, we present heuristics and practical solutions for the problem. For each memoryless strategy (equivalently, allowed program) in a partial program, the evaluation involves both correctness and performance checks, i.e., involves solving a LimAvg-Safety objective on a MDP. We present three practical methods for examining and evaluating memoryless strategies and they are as follows: (1) We use partial correctness checks on strategies to eliminate sets of programs that resolve some of the partial program choices in the same way. (2) We show that quantitative probabilistic bisimilarity is sound with respect to limit-average objectives, use this result to show that abstraction techniques can be used to reduce the size of MDPs. (3) We show how properties of commonly written programs and partial programs can be used to efficiently solve the MDPs that arise in the evaluation of a memoryless strategy.

4.1 Strategy elimination

Before the presentation of the elimination of strategies using partial correctness check, we present the general strategy enumeration scheme for partial program resolution. We first introduce the notions of a partial strategy and strategy tree.

Partial strategy and strategy trees. A *partial memoryless strategy* for Player 1 is a partial function that maps observations to actions. A *strategy tree* is a finite branching tree labelled with partial memoryless strategies of Player 1 such that:

– Every leaf node in the tree is labelled with a complete strategy.

– If a node and its parent are labelled with σ_1 (defined on O_1) and σ_2 (defined on O_2) respectively, then $O_2 \subsetneq O_1$.
– Two nodes, neither of which are the ancestor of the other, are labelled with partial strategies which are incompatible (i.e., there exists an observation for which they choose different actions).

A complete strategy tree for Player 1 is a strategy tree in which all Player 1 memoryless strategies are present as labels.

Algorithm 1 Strategy Elimination

```
Input: $M$: partial program;
  $W$: performance model;
  Sch: a scheduler;
  Safety: a safety condition
Output: $Candidates$: Strategies
  $StrategySet \leftarrow$ CompleteTree($M$)
  {It returns a complete strategy tree}
  $Candidates \leftarrow \emptyset$
  while $StrategySet \neq \emptyset$ do
    Pick $Tree$ from $StrategySet$
    $\sigma \leftarrow$Root($Tree$)
    {Root returns the root of the tree}
    if PartialCheck($\sigma$, Safety) then
      $StrategySet =$
          $StrategySet \cup$ children($Tree$)
    if $Tree$ is singleton then
      $Candidates = Candidates \cup \{\sigma\}$
  return $Candidates$
```

In the strategy enumeration scheme, we maintain a set of candidate strategy trees and check each one for partial correctness. If the partial strategy of the root of the tree fails the partial correctness check, then we can remove the whole tree from the set. Otherwise, we replace the tree with the set of children of the tree. Once we reach a complete strategy, we evaluate for full correctness and compute the value. The initial set consists of a complete strategy tree. In practice, the choice of this tree can be instrumental in the success of partial correctness checks in elimination of strategies. In particular, trees which fix the choices from the beginning of the partial program at the top of tree are more useful to partial correctness checks. The scheme for solving the partial program resolution problems is shown in Algorithm 1, and the full details are presented in the full version.

The PartialCheck function checks for the partial correctness of partial strategies. It either returns "Incorrect" if it is able to prove that any strategy compatible with the input is unsafe or it returns "Don't know". For example, in practice, for the partial correctness checks the following steps are useful: (a) checking of lock discipline to ensure the absence of deadlocks; (b) simulation of the partial program on small inputs; and (c) other similar static checks.

Once the set of candidate strategies is obtained, we evaluate them separately, one by one. Fixing a Player 1 memoryless strategy in a ImpIn $2\frac{1}{2}$-player game will turn it into a perfect-information MDP which can be solved efficiently due to the absence of imperfect information.

4.2 Quantitative probabilistic abstraction

Abstraction is a standard technique to handle state space explosion and it has been mainly used for boolean objectives. However for the partial program resolution problem we require abstraction that also preserves quantitative objectives such as LimAvg and LimAvg-Safety. We show that an extension of probabilistic bisimilarity [14] with a condition for weight function preserves the quantitative objectives.

Quantitative probabilistic bisimilarity. A binary equivalence relation $\equiv$ on the states of a MDP is a *quantitative probabilistic bisimilarity* relation if

– $\forall s \equiv s' : s \in B \leftrightarrow s' \in B$, i.e., s is unsafe if and only if s' is unsafe,
– $\forall s \equiv s', a \in A : \sum_{t \in C} \Delta(s,a)(t) = \sum_{t \in C} \Delta(s',a)(t)$ where C is an equivalence class of $\equiv$, and
– $\forall s, t \in C : \forall s', t' \in C' : \gamma(s,a,s') = \gamma(t,a,t')$.

Two states s and s' are called quantitative probabilistic bisimilar if $s \equiv s'$.

We now define the notion of quotients. Informally, a quotient of an MDP $\mathcal{G}$ under qualitative probabilistic bisimilarity relation $\equiv$ is an MDP $({}^{\mathcal{G}}/_{\equiv})$ where the states are the equivalence classes of $\equiv$ and: (i) $\gamma(C, a, C') = \gamma(s, a, s')$ where $s \in C$ and $s' \in C'$, and (ii) $\Delta(C, a)(C') = \sum_{t' \in C'} \Delta'(s, a)(t)$ where $s \in C$. The following theorem states that quotients preserve the $\mathrm{LimAvg\text{-}Safety}$ values of an MDP (proof in the full version).

Theorem 6. *Given an MDP $\mathcal{G}$, a quantitative probabilistic bisimilarity relation $\equiv$, and a limit-average safety objective f, the values in $\mathcal{G}$ and $({}^{\mathcal{G}}/_{\equiv})$ coincide for the limit-average safety objective.*

The previous theorem implies that any abstraction that respects quantitative probabilistic bisimilarity is a sound abstraction. We will consider the following abstractions and all of them respect the quantitative probabilistic bisimilarity.

- Standard data abstraction: This abstraction erases the values of the variables which do not appear in any guard.
- Equality abstraction: This preserves only the equality relations between the variables and erases the exact values.
- Order abstraction (in the presence of non-decreasing variables): This preserves only the order relations between variables.

4.3 Evaluation of a memoryless strategy

We now examine how the structure of common programs and partial programs can be exploited to optimize the solving of the MDPs obtained by fixing a Player 1 memoryless strategy in the ImplIn $2\frac{1}{2}$-player game for partial program resolution.

In the case of a non-deterministic scheduler where the scheduling choices are made by an antagonistic environment, fixing a Player 1 strategy in the program resolution game gives us a non-deterministic transition system. The value of the strategy can be found on this graph by using a standard min-mean cycle algorithm (for example, [15]).

In the case of analysis of probabilistic schedulers we are required to solve Markov chains with limit-average objectives. Markov chains arise for two reasons: (1) In many cases, the input can be abstracted away using data abstraction and the problem is reduced to solving a Markov Chain with a LimAvg objective. (2) The most efficient algorithm to solve MDPs with limit-average objectives is the strategy improvement algorithm [11], and each step of the algorithm involves solving a Markov chain. Hence we focus on solving Markov chains with limit-average objectives efficiently. In practice, a large percentage of concurrent programs are written to ensure some progress conditions. For example, many terminating concurrent programs ensure *lock-freedom* [12]. Lock-freedom ensures that some thread always makes progress in a finite number of steps. This leads to a Markov chain with a directed-acyclic tree like structure with only few cycles introduced to eliminate finite executions as mentioned in Section 2.

We present a *forward propagation* technique to compute the stationary probabilities for these Markov chains. The method to compute the stationary distribution for a Markov chain involves solving a set of linear equalities using Gaussian elimination. For Markov chains that satisfy the directed-acyclic tree like structure with few cycles we speed up the elimination of variables by eliminating variables in the tree by forward propagating the root variable. Using the forward propagation technique, we were able to handle the special Markov chains of up to 100,000 states in a few seconds in the experiments.

5 Experiments

We describe the results of applying our prototype implementation. The implementation uses techniques of Section 4 on four examples. In the examples, obtaining a correct program is straightforward and we focus on the synthesis of optimal programs.

LC: CC	Granularity	Performance
1:100	Coarse	1
	Medium	1.15
	Fine	1.19
1:20	Coarse	1
	Medium	1.14
	Fine	1.15
1:10	Coarse	1
	Medium	1.12
	Fine	1.12
1:2	Coarse	1
	Medium	1.03
	Fine	0.92
1:1	Coarse	1
	Medium	0.96
	Fine	0.80

Table 1. Performance of shared buffers under various locking strategies: LC and CC are the locking cost and data copying cost

The partial programs were manually abstracted (using the data and order abstractions) and translated into PROMELA, the input language of the SPIN model checker [13]. The abstraction step was straightforward and could be automated. The transition graphs were generated using SPIN. In the next step, our tool constructs the game graph as the product with the scheduler and performance model. The resulting game was solved for the LimAvg-Safety objectives using our techniques in Section 4. The examples we considered were small (each thread was running a procedure with 15 to 20 lines of code). The running time of the tool was under a minute for all but one case (Example 2 with larger value of n), where the running time was under five minutes. The experiments were run on a dual-core 2.5Ghz machine with 2GB of RAM. For all examples, the tool reports normalized performance metrics where higher values indicate better performance.

Example 1. We consider the producer-consumer example described in Section 1. The partial program models a four slot concurrent buffer which is operated on by producers and consumers. Here, we try to synthesize lock granularity. The synthesis results are presented in Table 1. The most important parameters of the performance model are the cost of locking/unlocking l and the cost c of copying data from/to shared memory. If the cost c was higher than l (by a factor 100:1), then the fine-grained locking approach is better (by 19 percent), and is the result of synthesis. If the cost l is equal to c, then the coarse-grained locking approach was found to perform better (by 25 percent), and thus the coarse-grained program is the result of the synthesis.

WC: LC	LWO	Performance for n				
		1	2	3	4	5
20:1	1	1.0	1.049	1.052	1.048	1.043
20:1	2	1.0	0.999	0.990	0.982	0.976
10:1	1	1.0	1.134	1.172	1.187	1.193
10:1	2	1.0	1.046	1.054	1.054	1.052

Table 2. Optimistic performance: WC, CC and LWO are the work cost, lock cost and the length of the work operation

Example 2. We consider the example of optimistic concurrency described in detail in Section 1. Referring back to the code in Figure 1, the number of operations performed optimistically is controlled by the variable `n`. We synthesized the optimal value for `n` for various performance models and the results are summarized in Table 2. In our experiments, we were able to find correspondence between the program behavior on a desktop machine and the behavior of our models: (a) When the partial program was tested on the desktop, we observed that the graph of performance-vs-`n` has a local maximum. In our experiments, we were able to find parameters for the performance model which have similar performance-vs-`n` curves. (b) Furthermore, changing the cost of locking operations on a desktop, by introducing small delays during locks, we were able to observe performance results similar to those produced by other performance model parameters.

Example 3. In this example, we try to synthesize the optimal number of threads for work sharing (details of the pseudocode is in the full version). In many concurrent programs, a number of operations can be performed independently. It might be better to spawn many threads to take advantage of multi-processor systems. However, for small number of operations, it is likely that the thread initialization cost will overcome any performance gain in the threads. We ran the experiments for various thread initial costs and number of operations.

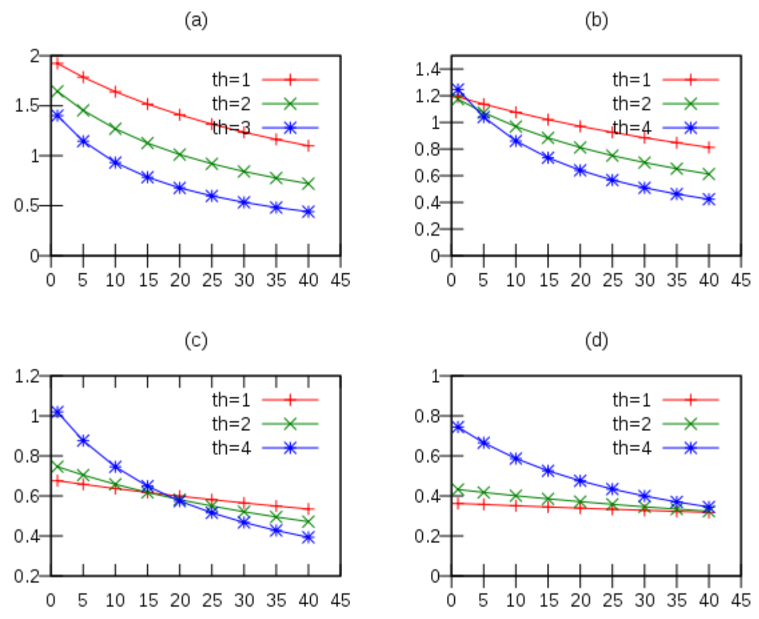

Fig. 4. Work sharing for initialization costs and thread counts: (a), (b), (c) and (d) stand for increasing amount of work to be shared

The results of the synthesis are summarized in Figure 4. In the graphs of Figure 4, the x-axis measures the initialization cost and the y-axis measures the performance. Each plot in the graph is for a different number of threads. The four graphs (a), (b), (c) and (d) are for a different amounts of work to be shared (in our case, the length of the array to be operated was varied between 8, 16, 32, and 64).

As it can be seen from the figure, for smaller amounts of work, spawning fewer threads usually outperforms spawning more threads. However, for larger amounts of work, greater number of threads outperforms smaller number of threads, even in the presence of higher initialization costs. The code was run on a desktop (with scaled parameters) and similar results were observed.

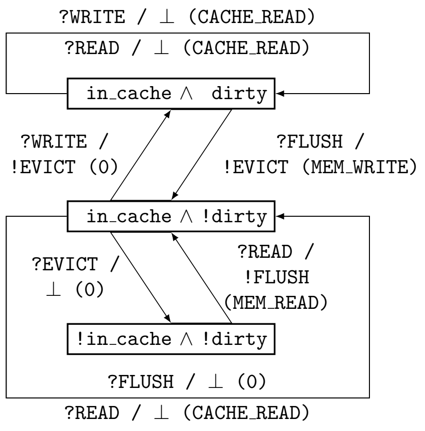

Fig. 5. Perf. Aut. for Example 4

Example 4. In this example, we study the effects of processor caches on programs using a performance model for the caches. A cache for a memory line is modeled as the weighted automaton shown in Figure 5. The performance model assigns costs to read and write actions, and these costs are different based on whether the memory line is currently in cache or not. The complete performance model is the synchronous product of such automata, one per memory line . Note that the only actions left in the performance model after taking the synchronous product (where the caches synchronize on `evict` and `flush` edges) are the `READ` and `WRITE` actions. These actions will be matched with the transitions of the partial program.

The partial program we consider is a pessimistic variation of the one used in Figure 1. The pseudo-code is presented in the full version. In this example, increasing the value of `n`, i.e., the number of operations performed under locks, increases the temporal locality of memory accesses. Hence, it is to be expected that increasing `n` will increase performance. We were able to observe the expected results from our experiments. For instance, the performance for `n = 5` is 2.32 times the performance for `n = 1`. Furthermore, increasing `n` from 5 to 10 gives an additional performance boost of about 20%. The result of the synthesis is the program where n is equal to 10.

Future Work. There are several promising future research directions: (i) the first is to consider synthesis of programs that access concurrent data structures; (ii) the second direction is to create benchmarks using which a performance model can be automatically obtained for different architecture.

References

1. H. Björklund, S. Sandberg, and S. Vorobyov. A discrete subexponential algorithm for parity games. In *STACS*, pages 663–674, 2003.
2. R. Bloem, K. Chatterjee, T.A. Henzinger, and B. Jobstmann. Better quality in synthesis through quantitative objectives. In *CAV*, 2009.
3. S. Burckhardt, R. Alur, and M. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *PLDI*, 2007.
4. S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-up: a complete and automatic linearizability checker. In *PLDI*, 2010.
5. K. Chatterjee, T. A. Henzinger, B. Jobstmann, and R. Singh. Measuring and synthesizing systems in probabilistic environments. In *CAV*, 2010.
6. S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In *PLDI*, pages 304–315, 2008.
7. A. Church. Logic, arithmetic, and automata. In *Proceedings of the International Congress of Mathematicians*, 1962.
8. E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, 1981.
9. A. Degorre, L. Doyen, R. Gentilini, J.-F. Raskin, and S. Toruńczyk. Energy and mean-payoff games with imperfect information. In *CSL*, 2010.
10. M. Emmi, J. Fischer, R. Jhala, and R. Majumdar. Lock allocation. In *POPL*, pages 291–296, 2007.
11. J. Filar and K. Vrieze. *Competitive Markov decision processes*. 1996.
12. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Elsevier Inc., 2008.
13. G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
14. B. Jonsson and K. Larsen. Specification and refinement of probabilistic processes. In *LICS*, pages 266–277, 1991.
15. R. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, (23), 1978.
16. M. Michael and M. Scott. Correction of a memory management method for lock-free data structures. Technical report, U. of Rochester, 1995.
17. C. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing, Reading, MA, USA, 1994.
18. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL*, 1989.
19. A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodík, V. Saraswat, and S. Seshia. Sketching stencils. In *PLDI*, pages 167–178, 2007.
20. A. Solar-Lezama, C. Jones, and R. Bodík. Sketching concurrent data structures. In *PLDI*, 2008.
21. A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.
22. M. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *FOCS*, pages 327–338, 1985.
23. M. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, 2010.