Quantitative Synthesis for Concurrent Programs*,**

Pavol Černý[1], Krishnendu Chatterjee[1], Thomas A. Henzinger[1], Arjun Radhakrishna[1], and Rohit Singh[2]

[1] IST Austria
[2] IIT Bombay

Abstract. We present an algorithmic method for the quantitative, performance-aware synthesis of concurrent programs. The input consists of a nondeterministic *partial program* and of a *parametric performance model*. The nondeterminism allows the programmer to omit which (if any) synchronization construct is used at a particular program location. The performance model, specified as a weighted automaton, can capture system architectures by assigning different costs to actions such as locking, context switching, and memory and cache accesses. The quantitative synthesis problem is to automatically resolve the nondeterminism of the partial program so that both correctness is guaranteed and performance is optimal. As is standard for shared memory concurrency, correctness is formalized "specification free", in particular as race freedom or deadlock freedom. For worst-case (average-case) performance, we show that the problem can be reduced to 2-player graph games (with probabilistic transitions) with quantitative objectives. While we show, using game-theoretic methods, that the synthesis problem is NEXP-complete, we present an algorithmic method and an implementation that works efficiently for concurrent programs and performance models of practical interest. We have implemented a prototype tool and used it to synthesize finite-state concurrent programs that exhibit different programming patterns, for several performance models representing different architectures.

1 Introduction

A promising approach to the development of correct concurrent programs is *partial program synthesis*. The goal of the approach is to allow the programmer to specify a part of her intent declaratively, by specifying which conditions, such as linearizability or deadlock freedom, need to be maintained. The synthesizer then constructs a program that satisfies the specification (see, for example, [17, 16,

* This work was partially supported by the ERC Advanced Grant QUAREM, the FWF NFN Grant S11402-N23 and S11407-N23 (RiSE), the EU NOE Grant ArtistDesign, and a Microsoft faculty fellowship.

** Full version available at http://arxiv.org/abs/1104.4306v1

19]). However, quantitative considerations have been largely missing from previous frameworks for partial synthesis. In particular, there has been no possibility for a programmer to ask the synthesizer for a program that is not only correct, but also *efficient* with respect to a specific performance model. We show that providing a quantitative performance model that represents the architecture of the system on which the program is to be run can considerably improve the quality and, therefore, potential usability of synthesis.

Motivating examples: *Example 1.* Consider a *producer-consumer* program, where k producer and k consumer threads access a buffer of n cells. The programmer writes a partial program implementing the procedures that access the buffer as if writing the sequential version, and specifies that at each control location a global lock or a cell-local lock can be taken. 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 currently accesses. The second program allows more concurrent behavior and is better in many settings. However, if the cost of locks is high (relative to the other operations), the global-locking approach is more efficient. In our experiments on a desktop machine, the global-locking implementation out-performed the cell-locking implementation by a factor of 3 in certain settings.

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

Fig. 1. Example 2

Example 2. Consider the program in Figure 1. It uses classic conflict resolution mechanism used for optimistic concurrency. The shared variables are `gdata`, on which some operation (given by the function `work()`) is performed repeatedly, and `gver`, the version number. Each thread has local variables `ldata` and `lver` that store local copies of the shared variables. The data is read (line 2) and operated on (line 6) without acquiring any locks. When the data is written back, the shared data is locked (line 8), and it is checked (using the version number, line 9) that no other thread has changed the data since it has been read. If the global version number has not changed, the new value is written to the shared memory (line 10), and the global version number is increased (line 11). If the global version number has changed, the whole procedure is retried. The number of operations (calls to `work`) performed optimistically without writing back to shared memory can influence the performance significantly. For approaches that perform many operations before writing back, there can be many retries and the performance can drop. On the other hand, if only a few operations are performed optimistically, the data has to be written back often, which also can 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 (line 4).

The partial program resolution problem. Our aim is to synthesize concurrent programs that are both correct and optimal with respect to a performance model. The input for partial program synthesis consists of (1) a finite-state partial program, (2) a performance model, (3) a model of the scheduler, and (4) a correctness condition. A *partial program* is a finite-state concurrent program that includes nondeterministic choices which the synthesizer has to resolve. A program is *allowed* by a partial program if it can be obtained by resolving the nondeterministic choices. The second input is a *parametric performance model*, given by a weighted automaton. The automaton assigns different costs to actions such as locking, context switching, and memory and cache access. It is a flexible model that allows the assignment of 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 automaton to specify complex cost models for memory and cache accesses. The performance model can be fixed for a particular architecture and, hence, need not be constructed separately for every partial program. The third input is the *scheduler*. Our schedulers are state-based, possibly probabilistic, models which support flexible scheduling schemes (e.g., a thread waiting for a long time may be scheduled with higher probability). In performance analysis, average-case analysis is as natural as worst-case analysis. For the average-case analysis, probabilistic schedulers are 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.

Quantitative games. We show that the partial program resolution problem can be reduced to solving *imperfect information* (stochastic) graph games with quantitative (limit-average or mean-payoff) objectives. Traditionally, imperfect information graph games have been studied to answer the question of existence of general, *history-dependent* optimal strategies, in which case the problem is undecidable for quantitative objectives [8]. We show that the partial program resolution problem gives rise to the question (not studied before) whether there exist *memoryless* optimal strategies (i.e. strategies that are independent of the history) in imperfect information games. We establish that the memoryless problem for imperfect information games (as well as imperfect information stochastic games) is NP-complete, and prove that the partial program resolution problem is NEXP-complete for both average-case and worst-case performance based synthesis. We present several techniques that overcome the theoretical difficulty of NEXP-hardness in cases of programs of practical interest: (1) First, we use a lightweight static analysis technique for efficiently eliminating parts of the strategy tree. This reduces the number of strategies to be examined significantly. We then examine each strategy separately and, for each strategy, obtain a (perfect information) Markov decision process (EDP). For MDPs, efficient strategy improvement algorithms exist, and require solving Markov chains. (2)

Second, Markov chains obtained from concurrent programs typically satisfy certain progress conditions, which we exploit using a forward propagation technique together with Gaussian elimination to solve Markov chains efficiently. (3) Our 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 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. For the producer-consumer example, we synthesized a program where two producer and two consumer threads access a buffer with four cells. The most important parameters of the performance model are the cost l of locking/unlocking and the cost c of copying data from/to shared memory. If the cost c is higher than l (by a factor 100:1), then the fine-grained locking approach is better (by 19 percent). If the cost l is equal to c, then the coarse-grained locking is found to perform better (by 25 percent). 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 increments the performance started to decrease. We measured the running time of the program on a desktop machine and observed the same phenomenon.

Related work. Synthesis from specifications is a classical problem [6, 7, 15]. More recently, sketching, a technique where a partial implementation of a program is given and a correct program is generated automatically, was introduced [17] and applied to concurrent programs [16]. However, none of the above approaches consider performance-aware algorithms for sketching; they focus on qualitative synthesis without any performance measure. We know of two works where quantitative synthesis was considered. In [2, 3] the authors study the synthesis of sequential systems from temporal-logic specifications. In [19, 5] fixed optimization criteria (such as preferring short atomic sections or fine-grained locks) are considered. Optimizing these measures may not lead to optimal performance on all architectures. None of the cited approaches use the framework of imperfect information games, nor parametric performance models.

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 definitions 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 $t_1 \ op \ t_2$, 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: $e := t_1\ rel\ t_2 \mid e \wedge e \mid \neg e$. *Guards* are formulae over L, G, and I. *Operations* are simultaneous assignments to variables in $L \cup G$, where each variable is assigned a term over L, G, and I.

Threads. A *thread* is a tuple $\langle Q, L, G, I, \delta, \rho_0, q_0 \rangle$, with: (a) a finite set of control locations Q and an initial location q_0; (b) L, G and I are as before; (c) an initial valuation of the variables ρ_0; and (d) a set δ of transition 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 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 the locations where δ is non-deterministic, i.e., locations where there exists a valuation of variables in L, G and I, for which there are multiple transitions whose guards evaluate 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, in which 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 the transition function of each thread becomes deterministic.

The guarded operations allow us to model basic concurrency constructs such as locks (for example, as variables in G and locking/unlocking is done using guarded operations) and compare-and-set. As partial program defined as a collection of fixed threads, 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 (denoted as $\mathsf{Tr}(M)$). Given a partial program M with n threads, let $\mathcal{C} = \{1, \ldots, n\}$ represent the set of threads of M.

- *State space.* Each state $s \in S$ of $\mathsf{Tr}(M)$ contains input and local variable valuations and locations for each thread in $\mathcal{C}$, and a valuation of the global variables. In addition, it contains a value $\sigma \in \mathcal{C} \cup \{*\}$, indicating which (if any) thread is currently scheduled. The initial state contains the initial locations of all threads and the initial valuations ρ_0, and the value $*$ indicating that no thread is currently scheduled.
- *Transition.* The transition function Δ defines interleaving semantics for partial programs. There are two types of transitions: thread transitions, that model one step of a scheduled thread, and environment transitions, that model input from the environment and the scheduler. For every $c \in \mathcal{C}$, there exists a thread transition labeled c from a state s to a state s' if and only if there exists a transition (q, g, a, q') of c such that (i) $\sigma = c$ in s (indicating that c is scheduled) and $\sigma = *$ in s', (ii) the location of c is q in s and q' in s', (iii) the guard g evaluates to true in s, and (iv) the valuation of

local variables of c and global variables in s is obtained from the valuation of variables in s' by performing the operation a. There is an environment transition labeled 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' differ 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, $\delta : Q_W \times \Sigma \to Q_W$ 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 performed by the system to which the performance measures are applied. A special symbol $ot \in \Sigma$ denotes that none of the tracked actions 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, 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 can be fixed for a particular architecture. This eliminates the need to construct a performance model for the synthesis of each partial program.

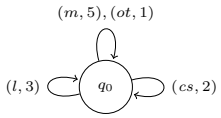

Fig. 2. Perf. aut.

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 a 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, i.e., 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 to transitions. An *execution* of a weighted probabilistic transition system is an infinite sequence of the form $(s_0 a_0 s_1 a_2 \ldots)$ where $s_i \in S$, $a_i \in A$, 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 and requires that states in B are never present in an execution. An execution $e = (s_0 a_0 s_1 a_2 \ldots)$ is *safe* (denoted by $e \in \text{Safety}_B$) if $s_i \notin B$, for all $i \geq 0$.

Limit-average and limit-average safety objectives. The *limit-average* objective assigns a real-valued quantity to every infinite execution e. We have $\text{LimAvg}_\gamma(s_0 a_0 s_1 a_1 s_2 \ldots) = \limsup_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n} \gamma((s_i, a, s_{i+1}))$ if there are no infinite cost transitions, and ∞ otherwise. The *limit-average safety* objective (defined by γ and B) is a *lexicographic* combination of a safety and a limit-average objective: $\text{LimAvg}_\gamma^B(e) = \text{LimAvg}_\gamma(e)$, if $e \in \text{Safety}_B$, and ∞ otherwise. Limit-average safety objectives can be reduced to limit-average objectives by making states in B absorbing (states with only self-loop transitions) and assigning the self-loop transitions the weight ∞.

Value of WPTS. 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 let $E^{pf}(\cdot)$ be the associated expectation measure. Given a WPTS T with weight function γ, and a policy pf, the value $Val(T, \gamma, \text{Safety}_B, pf)$ is the expected value $E^{pf}(\text{LimAvg}_\gamma^B)$ of the limit-average safety objective. The *value* of the WPTS is the supremum value over all policy functions, i.e., $Val(T, \gamma, \text{Safety}_B) = \sup_{pf} Val(T, \gamma, \text{Safety}_B, pf)$.

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) nondeterministically (nondeterministic 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 evaluate the performance of a program, we need to take into account the scheduler and the performance model. Given a program P, a scheduler Sch, and a performance model W, we construct a WPTS, denoted $\mathsf{Tr}(P, \text{Sch}, W)$, with a weight function γ as follows. A state s of $\mathsf{Tr}(P, \text{Sch}, W)$ is composed of a state of the transition system of P ($\mathsf{Tr}(P)$), a state of the scheduler Sch and a state of the performance model 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 mention 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. We can check for absence of data-races by denoting as unsafe states those in which there exist two enabled transitions (with at least one being a write) accessing a particular shared variable, from different threads. (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. The unsafe states are those where there exists two or more threads with each one waiting for a resource held by the next one.

Value of a program and of a partial program. For P, Sch, W as before and Safety_B is a safety objective, we define the value of the program using the composition of P, Sch and W as: $\mathit{ValProg}(P, \text{Sch}, W, \text{Safety}_B) = \mathit{Val}(\mathsf{Tr}(P, \text{Sch}, W), \gamma, \text{Safety}_B)$. For be a partial program M, let $\mathcal{P}$ be the set of all allowed programs. The value of M, $\mathit{ValParProg}(M, \text{Sch}, W, \text{Safety}_B) = \min_{P \in \mathcal{P}} \mathit{ValProg}(P, \text{Sch}, W, \text{Safety}_B)$.

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 $\mathit{ValProg}(P, \text{Sch}, W, \text{Safety}_B)$ is minimized. Informally, we have: (i) if the value $\mathit{ValParProg}(M, \text{Sch}, W, \text{Safety}_B)$ is ∞, then no safe program exists; (ii) if it is finite, then the answer is the optimal safe program, i.e., a correct program that is 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 $\mathit{ValParProg}(M, \text{Sch}, W, \text{Safety}_B) \leq \lambda$.

3 Quantitative Games on Graphs

Games for synthesis of controllers and sequential systems from specifications have been well studied in literature. We show how the partial program resolution problems can be reduced to quantitative imperfect information games on graphs. We also show that the arising technical questions on game graphs is different from the classical problems on quantitative graph games.

3.1 Imperfect information games for partial program resolution

An *imperfect information stochastic game graph* is a tuple $\mathcal{G} = \langle S, A, \text{En}, \Delta, (S_1, S_2), O, \eta, s_0 \rangle$, where S is a finite set of states, A is a finite set of actions, $\text{En} : S \to 2^A \setminus \emptyset$ is a function that maps every state s to the non-empty set of actions enabled at s, and s_0 is an initial state. The transition function $\Delta : S \times A \to \mathcal{D}(S)$ is a probabilistic function which maps a state s and an enabled action a to the probability distribution $\Delta(s, a)$ over the successor states. The sets (S_1, S_2) define a partition of S into Player-1 and Player-2 states, respectively; and the function $\eta : S \to O$ maps every state to an observation from the finite observation set O. We refer to these 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.

Special cases. We also consider ImpIn 2-player games (no randomness), perfect information games (no partial information), MDPs (only one enabled action for Player 1 states) and Markov chains (only one enabled action for all states) as special cases of ImpIn $2\frac{1}{2}$-player games. For full definitions, see [4].

The informal semantics for an 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 $\Delta(s, a)$.

A strategy for Player 1 (Player 2) is a "recipe" that chooses an action for her based on the history of observations (states). Memoryless Player 1 (Player 2) strategies are those which choose an action based only on the current observation (state). We denote the set of Player 1 and Player 2 strategies by Σ and Γ, respectively, and the set of Player 1 and Player 2 memoryless strategies by Σ^M and Γ^M, respectively.

Probability space and objectives Given a pair of Player 1 and Player 2 strategies (σ, τ), it is possible to define a unique probability measure $\Pr^{\sigma,\tau}(\cdot)$ over the set of paths of the game graph. For details, refer to any standard work on $2\frac{1}{2}$-player stochastic games (for example, [18]).

In a graph game, the goal of Player 1, i.e., the *objective* is given as a boolean or quantitative function from paths in the game graph to either $\{0, 1\}$, or $\mathbb{R}$. We consider only the LimAvg-Safety objectives defined in Section 2. Player 1 tries to maximize the expected value of the objective. The *value* of a Player 1 strategy σ is defined as $\mathit{ValGame}(f, \mathcal{G}, \sigma) = \sup_{\tau \in \Gamma} \mathbb{E}^{\sigma,\tau}[f]$ and the value of the game is defined as $\mathit{ValGame}(f, \mathcal{G}) = \inf_{\sigma \in \Sigma} \mathit{ValGame}(f, \mathcal{G}, \sigma)$.

For a more detailed exposition on ImpIn $2\frac{1}{2}$-player graph games and the formal definition of strategies, objectives, and values, see [4].

Decision problems. 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 if there is a Player 1 strategy (resp. memoryless strategy) σ with $\mathit{ValGame}(f, \mathcal{G}, \sigma) \leq q$. 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.). Traditional game theory study always considers the general decision problem which is undecidable for limit-average objectives [8] in imperfect information games.

Theorem 1. *[8] The decision problems for* LimAvg *and* LimAvg-Safety *objectives are undecidable for* ImpIn $2\frac{1}{2}$- *and* ImpIn 2-*player game graphs.*

However, we show here that the partial program resolution problems reduce to the memoryless decision problem for imperfect information games.

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

The proof relies on a construction of a game graph similar to the product of a program, a scheduler and a performance model. Player 2 chooses the thread to be scheduled and Player 1 resolves the nondeterminism when the scheduled thread c is in a location in $Sk(c)$. The crucial detail is that Player 1 can observe only the location of the thread and not the valuations of the variables. This partial information gives us a one-one correspondence between the memoryless strategies of Player 1 and programs allowed by the partial program.

3.2 Complexity of ImpIn Games and partial-program resolution

We establish complexity bounds for the relevant memoryless decision problems and use them to establish upper bounds for the partial program resolution problem. We also show a matching lower bound. First, we state a theorem on complexity of MDPs.

Theorem 3. *[9] The memoryless decision problem for* LimAvg-Safety *objectives can be solved in polynomial time for MDPs.*

Theorem 4. *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.

Remark 1. The NP-completeness of the memoryless decision problems rules out the existence of the classical strategy improvement algorithms as their existence implies existence of randomized sub-exponential time algorithms (using the techniques of [1]), and hence a strategy improvement algorithm would imply a randomized sub-exponential algorithm for an NP-complete problem.

Theorem 5. *The partial-program resolution decision problem is* NEXP*-complete for both nondeterministic and probabilistic schedulers.*

Proof. (a) The NEXP upper bound follows by an exponential reduction to the ImpIn games' memoryless decision problem (Theorem 2), and by Theorem 4.

(b) We reduce the NEXP-hard problem *succinct 3*-SAT (see [14]) to the partial program resolution problem to show NEXP-hardness. The idea is to construct a two thread partial program (shown in Figure 3) where Thread 1 chooses a clause from the formula and Thread 2 will determine the literals in the clause and then, enters an error state if the clause is not satisfied.

Given an instance of succinct 3-SAT over variables $v_1, \ldots v_M$, i.e., a circuit $\mathcal{Q}$ which takes pairs (i, j) and returns the j^{th} literal in the i^{th} clause. Thread 1 just changes the global variable i, looping through all clause indices.

```
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

Thread 2 will first non-deterministically choose 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. This can be done in polynomial time. (c) If the i^{th} clause is not satisfied with the valuation $\mathcal{V}$, it goes to an error state.

To show the validity of the reduction: (i) Given a satisfying valuation for $\mathcal{Q}$, choosing that valuation in the first steps of Thread 2 will obviously generate a safe program. (ii) Otherwise, for every valuation $\mathcal{V}$ chosen in the partial program, there exists a clause (say k) which is not satisfied. We let Thread 1 run till i becomes equal to k and then let Thread 2 run. The program will obviously enter the error state. Note that the result is independent of schedulers (non-deterministic or probabilistic), and performance models (as it uses only safety objectives). □

4 Practical Solutions for Partial-Program Resolution

Algorithm 1 Strategy Elimination

Input: M: partial program;
W: performance model;
Sch: scheduler;
Safety: safety condition
Output: *Candidates*: Strategies
StrategySet ← `CompleteTree`(M)
{A complete strategy tree}
Candidates ← ∅
while *StrategySet* ≠ ∅ **do**
  Choose *Tree* from *StrategySet*
  σ ←`Root`(*Tree*)
  if `PartialCheck`(σ, Safety) **then**
    StrategySet = *StrategySet* ∪ `children`(*Tree*)
  if *Tree* is singleton **then**
    Candidates = *Candidates* ∪ {σ}
return *Candidates*

We present practical solutions for the computationally hard (NEXP-complete) partial-program resolution problem.

Strategy elimination. 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 from observations to actions. A *strategy tree* is a finite branching tree labelled with partial memoryless strategies of Player 1 such that: (a) Every leaf node is labelled with a complete strategy; (b) Every node is labelled with a unique partial strategy; and (c) For any parent-child node pair, the label of the child (σ_2) is a proper extension of the label of parent (σ_1), i.e., $\sigma_1(o) = \sigma_2(o)$ when both are defined and the domain of σ_2 a proper superset of σ_1. A complete strategy tree is one where all Player 1 memoryless strategies are present as labels.

In the strategy enumeration scheme, we maintain a set of candidate strategy trees and check each one for partial correctness. If the root label of the tree fails

the partial correctness check, then remove the whole tree from the set. Otherwise, we replace it with the children of the root node. The initial set is a single complete strategy tree. In practice, the choice of this tree can be instrumental in the efficiency of partial correctness checks. Trees which first fix the choices that help the partial correctness check to identify an incorrect partial strategy are more useful. The partial program resolution scheme is shown in Algorithm 1, and the details are presented in the full version.

The `PartialCheck` function checks for the partial correctness of partial strategies, and returns "Incorrect" if it is able to prove that all strategies compatible with the input are unsafe, or it returns "Don't know". In practice, for the partial correctness checks the following steps can be used: (a) checking of lock discipline to prevent deadlocks; and (b) simulation of the partial program on small inputs; The result of the scheme is a set of candidate strategies for which we evaluate full correctness and compute the value.

Evaluation of a memoryless strategy. Fixing a memoryless Player 1 strategy in a Impln $2\frac{1}{2}$-player game for partial program resolution gives us (i) a non-deterministic transition system in the case of a non-deterministic scheduler, or (ii) an MDP in case of probabilistic schedulers. These are perfect-information games and hence, can be solved efficiently. In case (i), we use a standard min-mean cycle algorithm (for example, [12]) to find the value of the strategy . In case (ii), we focus on solving Markov chains with limit-average objectives efficiently. Markov chains arise from MDPs due to two reasons: (1) In many cases, program input can be abstracted away using data abstraction and the problem is reduced to solving a LimAvg Markov Chain. (2) The most efficient algorithm for LimAvg MDPs is the strategy improvement algorithm [9], and each step of the algorithm involves solving a Markov chain (for standard techniques, see [9]).

In practice, a large fraction of concurrent programs are designed to ensure progress condition called *lock-freedom* [10]. Lock-freedom ensures that some thread always makes progress in a finite number of steps. This leads to Markov chains 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 stationary probabilities for these Markov chains. Computing the stationary distribution for a Markov chain involves solving a set of linear equalities using Gaussian elimination. For Markov chains that satisfy the special structure, we speed up the process by eliminating variables in the tree by forward propagating the root variable. Using this technique, we were able to handle the special Markov chains of up to 100,000 states in a few seconds in the experiments.

Quantitative probabilistic abstraction. To improve the performance of the synthesis, we use standard abstraction techniques. 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 [13] 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 (a) $s \equiv s'$ iff s and s' are both safe or both unsafe; (b) $\forall s \equiv s', a \in A : \sum_{t \in C} \Delta(s,a)(t) = \sum_{t \in C} \Delta(s',a)(t)$ where C is an equivalence class of $\equiv$; and (c) $s \equiv s' \wedge t \equiv t' \implies \gamma(s,a,s') = \gamma(t,a,t')$. The states s and s' are *quantitative probabilistic bisimilar* if $s \equiv s'$.

A *quotient* of an MDP $\mathcal{G}$ under quantitative 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 $s \in C$.

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.*

Consider a standard abstraction technique, *data abstraction*, which erases the value of given variables. We show that under certain syntactic restrictions (namely, that the abstracted variables do not appear in any guard statements), the equivalence relation given by the abstraction is a quantitative probabilistic bisimilarity relation and thus is a sound abstraction with respect to any limit-average safety objective. We also consider a less coarse abstraction, *equality and order abstraction*, which preserves equality and order relations among given variables. This abstraction defines a quantitative probabilistic bisimilarity relation under the syntactic condition that the guards test only for these relations, and no arithmetic is used on the abstracted variables.

5 Experiments

We describe the results obtained by applying our prototype implementation of techniques described above on four examples. In the examples, obtaining a correct program is not difficult and we focus on the synthesis of optimal programs.

The partial programs were manually abstracted (using the data and order abstractions) and translated into PROMELA, the input language of the SPIN model checker [11]. The abstraction step was straightforward and could be automated. The transition graphs were generated using SPIN. Then, our tool constructed the game graph by taking the product with the scheduler and performance model. The resulting game was solved for the LimAvg-Safety objectives using techniques from Section 4. The examples we considered were small (each thread running a procedure with 15 to 20 lines of code). The synthesis time was under a minute for all but one case (Example 2 with larger values of n), where it 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, with two consumer and two producer threads. 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.

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 most important parameters in the performance model are the cost of locking/unlocking l and the cost c of copying data from/to shared memory. If c was higher than l (by 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.

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

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 3. We synthesize the optimal number of threads for work sharing (pseudocode in the full version). For independent operations, multiple threads utilize multiple processors more efficiently. However, for small number of operations, thread initialization cost will possibly overcome any performance gain. The experimental results are summarized in Figure 4. The x- and y- axes measure the initialization cost and performance, respectively. Each plot in the graph is for a different number of threads. The two graphs (a) and (b) are for a different amounts of work to be shared (the length of the array to be operated was varied between 16, and 32). Further graphs are in the full version. As it can be seen from the figure, for smaller amounts of work, spawning fewer threads is usually better. 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.

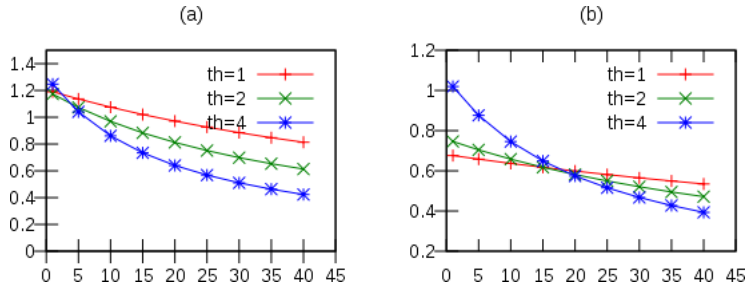

Fig. 4. Work sharing for initialization costs and thread counts: More work is shared in case (b) than case (a)

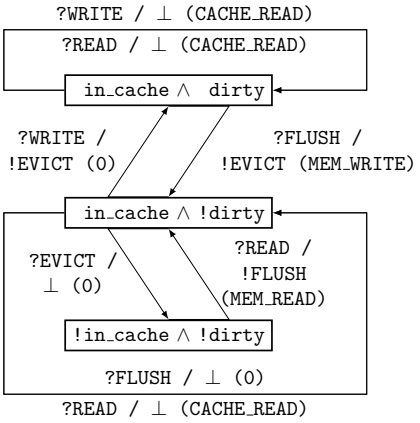

Fig. 5. Perf. aut. for Example 4

Example 4. We study the effects of processor caches on performance using a simple performance model for caches. A cache line is modeled as in Figure 5. It assigns differing costs to read and write actions if the line is cached or not. The performance model is the synchronous product of one such automata per memory line. The only actions in the performance model after the synchronous product (caches synchronize on `evict` and `flush`) are `READ` and `WRITE` actions. These actions are matched with the transitions of the partial program.

The partial program is a pessimistic variant of Figure 1 (pseudocode in full version). Increasing `n`, i.e., the number of operations performed under locks, increases the temporal locality of memory accesses and hence, increase in performance is expected. We observed the expected results in our experiments. For instance, increasing `n` from 1 to 5 increases the performance by a factor of 2.32 and increasing `n` from to 10 gives an additional boost of about 20%. The result of the synthesis is the program with $n = 10$.

6 Conclusion

Summary. Our main contributions are: (1) we developed a technique for synthesizing concurrent programs that are both correct and *optimal*; (2) we introduced a parametric performance model providing a flexible framework for specifying performance characteristics of architectures; (3) we showed how to apply imperfect-information games to the synthesis of concurrent programs and established the complexity for the game problems that arise in this context (4) we developed and implemented practical techniques to efficiently solve partial-program synthesis, and we applied the resulting prototype tool to several examples that illustrate common patterns in concurrent programming.

Future work. Our approach examines every correct strategy. There is thus the question whether there exists a practical algorithm that overcomes this limitation. Also, we did not consider the question which solution(s) to present to the programmer in case there is a number of correct strategies with the same performance. Furthermore, one could perhaps incorporate some information on the expected workload to the performance model. There are several other future research directions: one is to consider the synthesis of programs that access concurrent data structures; another is to create benchmarks from which performance automata can be obtained automatically.

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*, pages 140–156, 2009.
3. K. Chatterjee, T. A. Henzinger, B. Jobstmann, and R. Singh. Measuring and synthesizing systems in probabilistic environments. In *CAV*, pages 380–395, 2010.
4. Krishnendu Chatterjee, Laurent Doyen, Hugo Gimbert, and Thomas A. Henzinger. Randomness for free. In *MFCS*, pages 246–257, 2010.
5. S. Cherem, T. Chilimbi, and S. Gulwani. Inferring locks for atomic sections. In *PLDI*, pages 304–315, 2008.
6. A. Church. Logic, arithmetic, and automata. In *Proceedings of the International Congress of Mathematicians*, 1962.
7. E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, pages 52–71, 1981.
8. A. Degorre, L. Doyen, R. Gentilini, J.-F. Raskin, and S. Toruńczyk. Energy and mean-payoff games with imperfect information. In *CSL*, pages 260–274, 2010.
9. J. Filar and K. Vrieze. *Competitive Markov decision processes*. 1996.
10. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Elsevier Inc., 2008.
11. G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
12. R. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, (23):309–311, 1978.
13. K. Larsen and A. Skou. Bisimulation through probabilistic testing. In *POPL*, pages 344–352, 1989.
14. C. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing, Reading, MA, USA, 1994.
15. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190, 1989.
16. A. Solar-Lezama, C. Jones, and R. Bodík. Sketching concurrent data structures. In *PLDI*, pages 136–148, 2008.
17. A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, pages 281–294, 2005.
18. M. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *FOCS*, pages 327–338, 1985.
19. M. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, pages 327–338, 2010.