Parallelizing the Spin Model Checker

Gerard J. Holzmann

Laboratory for Reliable Software,
Jet Propulsion Laboratory, California Institute of Technology,
Pasadena, CA 91109, USA
gholzmann@acm.org

Abstract. We describe an extension of the Spin model checker that allows us to take advantage of the increasing number of cpu-cores available on standard desktop systems. Our main target is to speed up the verification process for safety properties, the mode used most frequently, but we also describe a small modification of the parallel search algorithm, called the piggyback algorithm, that is remarkably effective in catching violations for an interesting class of liveness properties at little cost.

Keywords: parallelism, concurrency, multi-core, model checking, Spin, breadth-first search, safety, liveness, bounded search, software verification.

1 Introduction

We build on the infra-structure provided by the model checker Spin [5]. Although the model checker targets the analysis of multi-threaded software applications, until recently the tool itself performed its analyses single-threaded, using just a single cpu. In 2005 a modification was introduced that allowed for the execution of the depth-first search analysis on multiple cpu-cores [6]. This extension was chosen because it can support both safety and liveness properties, yet for liveness properties the depth-first algorithm could only take advantage of parallel execution on no more than two cpu-cores.

Parallelization of breadth-first search is often considered simpler, and could lead to greater gains, so it is attractive to support also this option, even if it means restricting it to the verification of safety properties alone. The parallel version of the breadth-first search described in this paper requires virtually no tuning or user adjustments and succeeds in providing an impressive performance improvement in the model checking process. We also show that a simple extension of this algorithm suffices to support also the verification of an interesting class of liveness properties without measurable overhead.

The remainder of this paper is organized as follows. In Section 2 we describe the basic breadth-first search algorithm that is used in Spin. In Section 3 we describe the parallelization of this algorithm, where we focus on the key issues of load balancing, lock avoidance, and partial order reduction. In Section 4 we discuss an extension that supports checks for liveness properties with a bounded cycle search option.
Section 5 presents documents the performance of the new algorithm when applied to a range of verification problems. Section 6, concludes the paper and summarizes the key results.

2 Breadth-First Search

Figure 1 gives the basic sequential algorithm for performing a breadth-first in a reachability graph, as used in the Spin model checker. The algorithm uses three sets of states: S, Q[0], and Q[1]. Set S is the set of visited states, which is initially empty. Every new state that is encountered during the search is entered into this set, to avoid duplicate work when the state is revisited later. Set S is typically implemented as a hashtable.

```
1 global t = 0 // toggle bit 0..1
2 global S = {} // statespace set
3 global Q[0] = {} // successor set
4 global Q[1] = {} // successor set
5 safety property f
6
7 add s0 to Q[0] and to S // initial state
8
9 Search()
10 do {
11    for each s in Q[t]
12       { delete s from Q[t]
13          for each successor s’ of s
14             { if s’ not in S
15                { add s’ to S
16                  if s’ violates f
17                    { report safety violation
18                      } else
19                        { add s’ to Q[1-t]
20                          }
21                }
22          }
23        t = 1 - t
24    } while (Q[t] is non-empty)
25 }
```

Fig. 1. Sequential breadth-first search

The breadth-first search proceeds by repeatedly generating the set of successor states (the ‘next’ generation) for a given set of states (the ‘current’ generation). These two sets are stored in successor sets Q[0] and Q[1]. As soon as all states in the ‘current’ generation of states have been processed, the roles of Q[0] and Q[1] switch, and what was the ‘next’ generation of states becomes the new ‘current’ generation, and the now empty former ‘current’ generation becomes