

Dynamic Resource Scaling for Explicit State Model Checking on Cloud

Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science
in
Computer Science and Engineering by Research

by

Yuvraj Singh
201102049

yuvraj.singh@research.iiit.ac.in



International Institute of Information Technology
Hyderabad - 500 032, INDIA
April 2023

Copyright © Yuvraj Singh, 2022
All Rights Reserved

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled “**Dynamic Resource Scaling for Explicit State Model Checking on Cloud**” by **Yuvraj Singh**, has been carried out under my supervision and is not submitted elsewhere for a degree.

Date

Adviser: Prof. Suresh Purini

Date

Co-Adviser: Prof. Venkatesh Choppella

To my Parents and Gurus

Acknowledgments

I want to thank my advisor, Prof. Suresh Purini from the bottom of my heart for all of his guidance, care, support, and encouragement. He has been exceedingly kind with his time, listening to my ideas, and providing insightful feedback. His dedication to producing impactful work has served as an inspiration. I would be remiss if I failed to mention his humility in dealing with life.

I am also grateful to my co-advisor, Prof. Venkatesh Choppella for his guidance. His formal and concrete ways of expressing an idea helped me think more clearly, and I learned a lot as a result.

I had the pleasure of collaborating with Surya Teja Palavalasa and Adhish Singla, I would like to thank them for their contribution to my work. I can't stress enough significant role my friends played during this period. Chandan, Tejas, Prateek, Divyanshu and Ghizlane for their support and encouragement.

I could not have undertaken this journey without the help, support and encouragements of my parents. Their belief in me kept me going through the process.

Abstract

Model checking is now a standard technology for verifying large and complex systems. While there are a range of tools and techniques to verify various properties of a system under consideration, in this work, we restrict our attention to safety checking procedures using explicit state space generation. The necessary hardware resources required in this approach depends on the model complexity and the resulting state transition graph that gets generated. This cannot be estimated apriori. For reasonably realistic models, the available main memory in even high end servers may not be sufficient. Hence, we have to use distributed safety verification approaches on a cluster of nodes. However, the problem of estimating the minimum number of nodes in the cluster for the verification procedure to complete successfully remains unsolved.

In the thesis, we propose a dynamically scalable model checker using an actor based architecture. Using the proposed approach, an end user can invoke a model checker hosted on a cloud platform in a push button fashion. Our safety verification procedures automatically expands the cluster by requesting more virtual machines from the cloud provider. Finally, the user gets to pay only for the hardware resources he rented for the duration of the verification procedure. We refer to this as Model Checking as Service. We approach this problem by proposing an asynchronous algorithm for safety checking in actor framework. The actor based approach allows for scaling the resources on a need basis and redistributes the work load transparently through state migration. We tested our approach by developing a distributed version of SpinJA model checker using Akka actor framework. We conducted our experiments on Google Cloud Engine (GCE) platform wherein we scale our resources automatically using the GCE API. On large models such as anderson.8 from BEEM benchmark suite, our approach reduced the model checking cost in dollars by 8.6x while reducing the wall clock time to complete the safety checking procedure 5.5x times.

Contents

Chapter	Page
1 Introduction	1
section1.1 Related Work	3
1.2 Thesis Organisation	4
chapter2 Background	5
2.1 Explicit State Model Checking	5
section2.2 Actor Model	6
2.3 Akka	7
chapter3 Distributed Approach	8
3.1 State Space Exploration	8
section3.2 Termination Detection	10
4 Adaptive State Space Exploration	13
section4.1 Introduction	13
4.2 Horizontal Scaling through Migration	13
section4.3 Sensing Memory Pressure	15
5 Experimental Results	16
chapter6 Conclusion	23

List of Figures

Figure	Page
1.1 Peterson's mutual exclusion algorithm in Promela.	2
figure1.2Model checking in a nutshell.3	
2.1 Example of actor behaviour.	7
1em	
3.1 Example illustrating why we may falsely conclude that the algorithm has terminated by verifying the degree condition without convergence test.	10
1em	
5.1 Execution, transient and migration times for various benchmarks using one and two nodes at a times cluster expansion policy.	18
figure5.2Comparison of total time T and effective time T_e for different static cluster sizes, adaptive single and two node cluster expansion policies.19	

List of Tables

Table	Page
5.1 State space sizes for BEEM models	17
5.2 Percentage overhead due to migration and transient execution phases for one and two node expansion policies.	20
5.3 Cost of safety verification, Total and effective execution times for anderson.8 and fischer.7.	21

Chapter 1

Introduction

Computing hardware has become increasingly complex over time, thanks in large part due to Moore's law and improvements in fabrication techniques. The software has also become more complicated to take full advantage of the hardware. Any bugs in complex software or hardware systems, such as in an automated driving system, could result in human casualties; in the case of data center hosting applications like credit card services, millions of users could be impacted [23], and in the case of a bug in hardware systems can also result in significant financial losses for the company having to recall them as well as potential privacy risks for tens of millions of users. Although good software design, development, and testing methods are essential for software engineering, they are by no means exhaustive and may still miss some bugs. It is also common knowledge that it is best to find bugs early in the software development life cycle, ideally right at the design stage, to reduce the cost of fixing them. Model checking and formal verification methods are increasingly being used for verifying the correctness of large and complex systems [17].

An algorithm or network protocol is described in model checking using a modeling language like Promela [6]. For instance, Figure 1.1 depicts the Promela model for the two-process Peterson's mutual exclusion algorithm. The inline assertion on lines 10 and 19 checks if two processes are in a critical section simultaneously. This check is referred to be a safety property. Depending on the specific order in which the two processes interleaved in our simulation of this model, we might not encounter an assertion violation. We must confirm that the assertion violation never happens in any interleaving of the process to validate the mutual exclusion safety property. To do this, we must thoroughly test every possibility. That is done by model checking tools such as Spin [11] by systematically exploring a state transition system from a given model specification. Hence this approach is called explicit state model checking. While safety properties can be specified using inline or global assertions, there is yet another class of properties that deals with the liveness of the system. For example, a liveness property for Peterson's mutual exclusion algorithm could be that any process intending to enter the critical section should be able to do so in a bounded number of steps. Liveness properties are specified using Linear Temporal Logic (LTL) in conjunction with a model under consideration. In this work, we focus only on checking safety properties, not liveness properties.

```

1 #define A_TURN 0
2 #define B_TURN 1
3 bit x, y;
4 byte mutex, turn;
5 active proctype A() {
6     x = 1;
7     turn = B_TURN;
8     y == 0 || (turn == A_TURN);
9     mutex++;
10    assert(mutex != 2);
11    mutex--;
12    x = 0;
13 }
14 active proctype B() {
15    y = 1;
16    turn = A_TURN;
17    x == 0 || (turn == B_TURN);
18    mutex++;
19    assert(mutex != 2);
20    mutex--;
21    y = 0;
22 }

```

Figure 1.1 Peterson’s mutual exclusion algorithm in Promela.

As the complexity of the model increases, so does the number of states in the underlying state transition graph, making explicit state model checking expensive in terms of computing and memory needs. In the scientific literature, parallel and distributed model checking techniques have been suggested as solutions to these issues. With distributed algorithms, we can use cores and memory resources distributed across numerous machines connected through a high-speed network like Gigabit Ethernet or Infiniband. Parallel algorithms allow us to use multiple cores in a shared memory multi-core machine. The parallelisation techniques in the past used the standard threading libraries available, and the distributed approaches used either MPI [16] or RDMA [8]. Other approaches based on vertex-centric distributed programming models have also been proposed [22] [19].

The problem with all the aforementioned approaches is that we need to have a good estimate of the memory requirements of the underlying state transition diagram a priori. This cannot be inferred automatically from the source model specification. If the estimated memory requirement is not sufficient, then the model checking process gets killed and has to be restarted with a fresh estimate. This leads to prohibitively huge computation times, most of which is redundant, and very expensive when resources have to be paid for in a cloud setting. As it is, explicit state model checking is considered to be expensive, now with this problem, it almost makes it impractical.

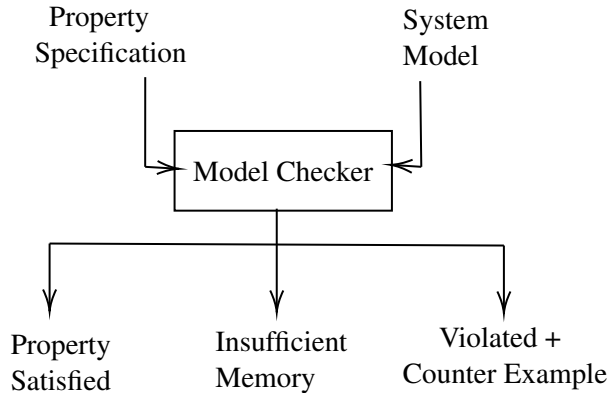


Figure 1.2 Model checking in a nutshell.

In the thesis, we propose a distributed safety checking algorithm, which senses the memory pressure, and based on that it automatically expands the cluster size (horizontal scaling) by requesting new nodes from the cloud provider. During the transient expansion phase, load balancing, state redistribution and other book-keeping work gets done. Then the regular computation continues as usual. All this happens seamlessly without the end user involvement, leading us towards Model Checking as a Service (MCaaS) paradigm.

In order to achieve dynamic scaling, we propose an asynchronous reachability analysis algorithm for safety checking in the actor framework for parallel and distributed systems [2]. An actor based model consists of a logical collection of actors which communicate with each other purely through messages. The main advantage of this approach is that actors allow mapping to one or more machines transparently leading to a parallel or a distributed setup, with no re-engineering of code. Whenever, excessive memory pressure is being sensed, the system enters a transient state in which new nodes are added to the systems and the actors are redistributed to reduce memory pressure on existing nodes. Further, during this process, computation also gets distributed. We have implemented the proposed system in the Java based SpinJa model checker[9] using Java’s distributed actor framework Akka[21]. To the best of our knowledge, no other published prior work has considered the memory estimation problem in explicit state model checking and how it can be addressed using dynamic scaling of resources.

1.1 Related Work

In this section , we confine ourselves to explicit state model checking and do not consider other model checking paradigms. An important performance metric in explicit state model checking is the number of transitions explored per second. With respect to this metric, in a sequential setting, Spin [11] is arguably the best model checker available out there. It has a robust implementation with many features. However, it does not work if the explored model size is too big to fit into memory. Murphi [10] is yet another model checker with no facility for parallel state space exploration.

DiVinE [3], Eddy Murphi [16] and PSpin [13] are examples of parallel and distributed model checkers. All of them use multi-threaded and message passing approaches (MPI) in their implementations. Xie et al. [22] and Singla et al. [19] proposed vertex centric distributed computing approaches for model checking using frameworks such as Pregel and Giraph. However, these approaches have large synchronization overhead for every super-step of computation. But a clean synchronous computational model allows for easy algorithm design and proof of correctness.

Parallel Murphi [20] uses an adaption of [15] for termination detection. It involves taking a snapshot of the system across nodes at a given instant of time to check whether the exploration is complete. Coti et al. [8] used OpenSHMEM library on an RDMA cluster to address the safety verification problem. RDMA clusters have faster access to remote memories than the Gigabit Ethernet based clusters which we used in our experimental analysis.

Aceto et al. [1] first considered the problem of offering model analysis and checking as a service on the cloud. Their approach is primary to use RESTful API to offer already existing Model Analysis and Checking (MACH) tool as a webservice. Kai Hu et al. [12] proposes an architecture for providing a multi-tenant cloud based verification as a service, but does not address scaling of model checking. All the aforementioned approaches requires us to fix the available resources upfront and cannot be changed once the program execution starts.

Camilli et al. [5, 7, 4] presented a map-reduce based approach for Computational Tree Logic (CTL) verification. First, the underlying state space is generated in a distributed file, which is then consumed by the verification procedure. In our approach, verification is done on-the-fly as the state space gets generated. The above approach even though scalable relies on expensive disk I/O operations.

1.2 Thesis Organisation

The layout of the rest of the thesis is as follows. Chapter 2 provides the necessary background on explicit state model checking and actor architecture; Chapter 3 presents our actor based distributed asynchronous safety checking algorithm; Chapter 4 presents how we adaptively scale-up resources and does load redistribution; Chapter 5 presents the relevant experimental results and their analysis; and finally we conclude with remarks on future work in Chapter 6.

Chapter 2

Background

In this chapter, we briefly review the essential ideas and terminology underlying explicit state model checking and the key ideas behind the actor model used to implement dynamically scalable distributed model checking.

2.1 Explicit State Model Checking

A state transition system $T = (S, Act, \rightarrow, I, L, AP)$ consists of a set of states S , a set of actions Act , a set $I \subseteq S$ of initial states, a set AP of atomic propositions, a transition relation $\rightarrow \subseteq S \times Act \times S$ and a labeling function $L : S \rightarrow 2^{AP}$. Let $s_0 \in I$. A state s *transits to* s' , written $s \rightarrow s'$ if there is an action a such that $(s, a, s') \in \rightarrow$. s' is *reachable from* s if there is a sequence of states s_0, \dots, s_n such that $s_i \rightarrow s_{i+1}$ for each $i \in \{0, \dots, n\}$, $s = s_0$ and $s' = s_{n+1}$.

Let ϕ denote a propositional formula over the set of atomic propositions AP . A set P of atomic propositions, where $P \subseteq AP$, *models* a proposition ϕ , written $P \models \phi$ if and only if all propositions $p \in P$ are true and all propositions $q \in AP \setminus P$ are false, then ϕ is true. The transition system T *satisfies* ϕ if for each state $s_0 \in I$, and for all state s reachable from s_0 , $L(s) \models \phi$.

Given a Promela program M , the model checker incrementally constructs a transition system T_M from M in a breadth-first search or a depth-first search manner. The transition system completely captures all possible behaviours of the Promela program across various execution runs. For example, in the Promela program in Figure 1.1, each state of the constructed transition system contains the current values of the global variables, program counter values for the processes A and B, and the value of their local variables. Thus, each state of T_M is a snapshot of the Promela program in execution.

Safety formulas are specified either globally in the Promela program, or are inlined (as in Figure 1.1). As a new state is generated, the model checker checks if the (safety) formula is satisfied by that state. If the state does not satisfy the formula, the model checker halts with a trace of the execution path leading to that state. The trace may be interpreted as a counterexample refuting the proposition.

If the number of bits required to encode a state is b , then there are 2^b distinct possible states, although all of them may not be reachable. So the total number of states in the underlying state transition system

Algorithm 1 Basic algorithm for safety checking through state space exploration.

```
1: procedure SAFETYCHECKING(Model m)
2:    $s_0 \leftarrow \text{GetInitialState}(m)$ 
3:    $S \leftarrow \{s_0\}$  ▷ Generated but unexplored states.
4:    $ES \leftarrow \{\}$  ▷ Already explored states.
5:   while  $S \neq \phi$  do
6:      $s \leftarrow \text{ChooseState}(S)$  ▷ Choose any state from  $S$ 
7:      $S \leftarrow S - \{s\}$ 
8:     if  $s \notin ES$  then
9:        $ES \leftarrow ES \cup \{s\}$ 
10:       $s_{succ} \leftarrow \text{GetSuccessors}(s)$ 
11:      for all  $p \in s_{succ}$  do
12:         $\text{CheckSafetyPredicate}(p)$ 
13:         $S \leftarrow S \cup \{p\}$ 
14:      end for
15:    end if
16:  end while
17: end procedure
```

can grow exponentially with the increase in the model complexity (e.g., the number of variables in the program) and there is no way to predict the memory required. This is the main bottleneck in the practical application of model checking. For real models of practical interest, the number of generated states are so large that they can neither be stored in the main memory of even high end machines nor the memory requirement be estimated. This is called as the state space explosion problem.

2.2 Actor Model

The actor model [2] is an abstraction that makes it easier to write concurrent, parallel and distributed applications. Actors are the primitive unit of computation. Every actor maintains an internal state. They communicate with each other by sending messages.

These messages are sent asynchronously via the network. Messages sent from an individual actor are always ordered on the receiving end, whereas they might be out of order if sent from different actors due to network latency. While processing a message, an actor can do the following things : create more actors, change internal state and send more messages

Many problems in multi-threaded code such as race conditions and dead locks arise out of sharing state across multiple threads. These bugs are hard to detect, reproduce and fix. In fact, one of the major applications of model checking, which is the theme of thesis, is to identify such bugs at the design level itself. The actor model avoids those problems by not letting any other actor modify the internal state of an actor directly, for example using method calls. Further, the mapping of actors to physical machines is done in a transparent fashion, and hence the same design and implementation, can be used for both parallel and distributed realizations of an algorithm. Apart from this, there are other added benefits like support for native supervision hierarchies which help in building resilient self-healing systems.

We use Akka toolkit that provides various actor model based tools in Scala and Java. All actors are part of an actor system. Every actor has a mailbox associated with it. Mailbox is the queue of unprocessed messages. Akka supports various concurrent data structures for the mailbox. We leverage the ability to move actors across machines in Akka and programmatically create virtual machines in modern clouds, to create an adaptive state verification system, that does not require estimation of the size of reachable state space.

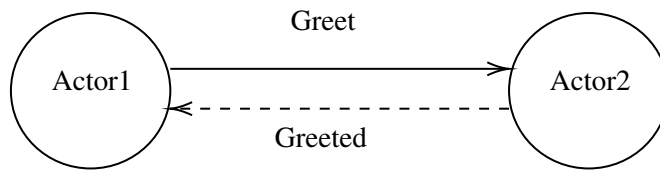


Figure 2.1 Example of actor behaviour.

In the example shown above has two message types: one to direct the actor to greet someone, and one to let the actor know when it has done so. **Greet** message contains information whom to greet and also the sender of the message. **Greeted** message hold information about whom to acknowledge that their message was received and also the initiator of the message. On receiving a **Greet** message Actor2 initiates a reply **Greeted** message and also changes it internals state representing that it has been greeted by Actor1. On receiving the **Greeted** message Actor1 updates it's internal state representing that his greeting to Actor2 has been received.

2.3 Akka

Akka is a source-available toolkit and runtime which allows to build concurrent parallel and distributed systems. Akka implements actor-based model and takes influence from Erlang. Akka is written in Scala and has language bindings both for Java and Scala. Actor model provides a level of abstraction that makes it easier to write correct concurrent, parallel and distributed systems. Akka provides multi-threaded behaviour without using low-level concurrency constructs like atomics or locks. Akka provides an elastic architecture which can either scale up and down. We would be using the scaling up capability of the architecture in our work.

Chapter 3

Distributed Approach

In this chapter, we will present the algorithms we have developed for asynchronous parallel distributed safety checking. As we know, termination detection in a distributed environment is not straight forward. We will demonstrate and prove the correctness of our termination detection algorithm.

3.1 State Space Exploration

Recall from Chapter 2, that safety checking involves generating a state transition graph from a given Promela model and verifying if each of the generated states satisfies associated local and global predicates. Algorithm 1 summarizes the sequential safety checking procedure. In this section, we propose an asynchronous algorithm for safety checking in the actor architectural framework.

Our design involves a Master actor and set of worker actors. The master actor stores the address of all the workers in a list and propagates the list along with the compiled Promela model to all the other actors before the job starts. Each worker actor stores the address of Master and other workers, and can send messages directly without any routing. We do not use any router actor, in order to avoid single point of contention for the messages. Each vertex in the state transition graph is mapped to a worker using a hash function. Thus the whole state transition graph is stored in a partitioned fashion across all the available workers. Each worker internally maintains the set of vertices already explored by it in a suitable lookup table.

The Master starts the computation by creating the initial state/vertex¹. Then it computes a hash of the vertex to find the worker to which it is mapped to, and sends it a `Compute` message with the vertex suitably packaged inside it. On receiving a `Compute` message, a worker checks from its local lookup table if the state has been already explored. If the state is not already explored, then the associated safety predicate is checked and the successor states are computed. For each successor state, a suitable `Compute` message will be sent to the worker to which it is mapped to. Each worker processes the messages independently with no synchronization requirement and this is how our parallel/distributed

¹In this work, we use the terms state and vertex interchangeably.

Algorithm 2 Actor based parallel/distributed asynchronous safety checking algorithm.

*/*Initiation of safety checking algorithm at the Master.*/*

1: **procedure** STARTJOB(*Model m*)

2: $s_0 \leftarrow \text{GetInitialState}(m)$

3: $\text{CmpMsg} \leftarrow \text{CreateCmpMsg}(s_0)$

4: */* W is the number of workers */*

5: $\text{worker} \leftarrow \text{Hash}(s_0) \bmod W$

6: $\text{Send}(\text{CmpMsg}, \text{worker})$

7: **end procedure**

8: */* Compute message handler at Worker-w. */*

9: **procedure** COMPUTE(*CmpMsg msg*)

10: $s \leftarrow \text{GetState}(msg)$

11: **if** $s \notin ES_w$ **then**

▷ ES_w is the local lookup table.

12: $\text{CheckPredicate}(s)$

13: $ES_w \leftarrow ES_w \cup \{s\}$

14: $s_{succ} \leftarrow \text{GetSuccessors}(s)$

15: $\text{StateSpaceGeneration}(S_{succ})$

16: **end if**

17: $in_w \leftarrow in_w + 1$

18: **end procedure**

19: **procedure** STATESPACEGENERATION(s_{succ})

20: **for all** $p \in s_{succ}$ **do**

21: $\text{CmpMsg} \leftarrow \text{CreateCmpMsg}(p)$

22: $\text{worker} \leftarrow \text{Hash}(p) \bmod W$

23: $out_w \leftarrow out_w + 1$

24: $\text{Send}(\text{CmpMsg}, \text{worker})$

25: **end for**

26: **end procedure**

asynchronous safety checking algorithm proceeds. Algorithm 2 summarizes the complete state space exploration algorithm. The procedure *Send* delivers the message to the destination actor asynchronously.

We use the SpinJa API that compiles the Promela Model into an Object that provides us with the information about the initial state of the model and also provides state local data for a given state s . The flow of the state space exploration is shown in Algorithm 2. Since every worker has its own model object and all of them can be encoded to compute different states at the same instant, there is no dependency between any computation and hence it is completely asynchronous.

3.2 Termination Detection

As explained in the previous section, the safety checking algorithm progresses asynchronously and the Master needs to determine when the algorithm terminated. Towards that, let us consider the underlying state transition graph associated with a given input Promela model. Every edge in the graph contributes single outdegree and single indegree to its source and destination vertices respectively. Thus the sum of indegrees of all the vertices is equal to the sum of their outdegrees.

For a worker w , let \widehat{in}_w and \widehat{out}_w denote the sum of indegrees and outdegrees of all the vertices assigned to w . If there are W workers after the asynchronous algorithm has terminated, which means that the state transition graph is completely constructed, the following condition holds good.

$$\sum_{i=1}^W \widehat{in}_i = \sum_{i=1}^W \widehat{out}_i \quad (3.1)$$

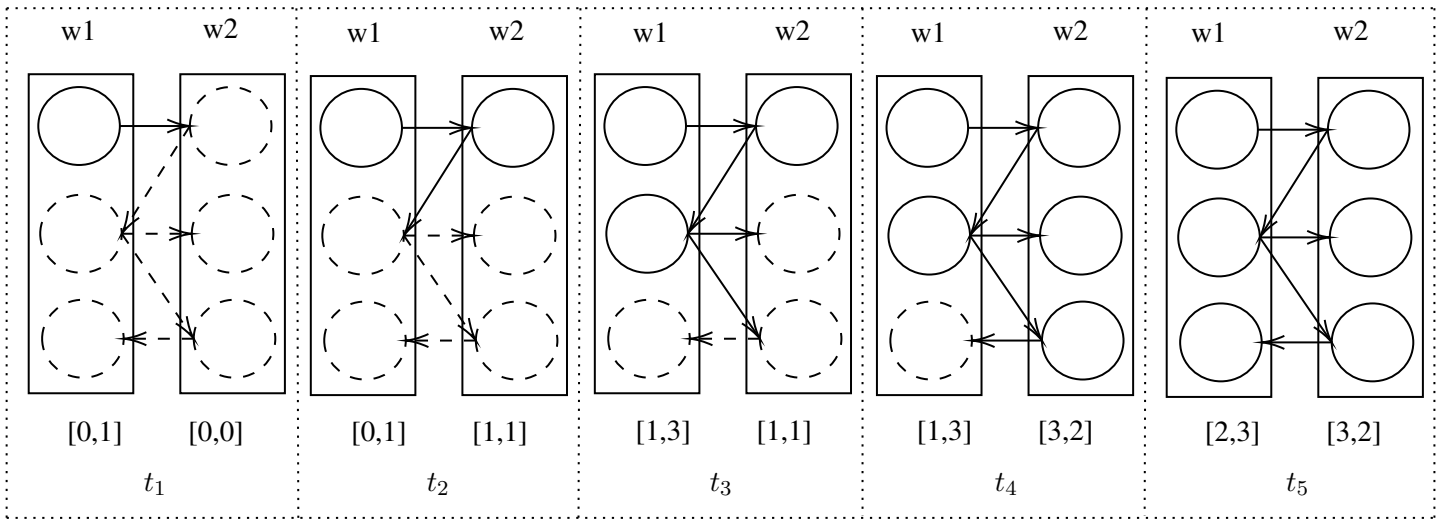


Figure 3.1 Example illustrating why we may falsely conclude that the algorithm has terminated by verifying the degree condition without convergence test.

Each worker w maintains the local accumulated indegrees and outdegrees in the variables in_w and out_w . Every time a worker is about to send a message to a successor vertex as a part of processing a vertex, refer Line 23 from Algorithm 2, it increments the out_w counter. Similarly, at the end of processing a `Compute` message, refer Line 17 from Algorithm 2, a worker increments the in_w counter.

The Master worker periodically sends a `StatusRequest` message to all the workers. In response to that, each worker w sends the tuple (in_w, out_w) to the Master. Let us say that the Master decides that the algorithm has terminated if Condition 3.1 is satisfied. We will show that it could lead to a false conclusion through an example. Figure 3.1 shows 5 snapshots across time as the state space exploration proceeds with two workers. The solid circles and edges denote already explored states and edges, whereas the dashed ones denote the yet to be explored. The tuples below denote the corresponding in and out counters for each of the workers at that time instant. The graph on the rightmost side is the final one when the algorithm has terminated. We can note that the sum of indegrees is equal to the sum of the outdegrees. When the Master sends a `StatusRequest` message, due to message delays and unsynchronized clocks, Worker 1 may send the tuple $[0, 1]$ from time instant t_1 and Worker 2 may send the tuple $[3, 2]$ from time instant $t_4 > t_1$. Now, when the Master evaluates the Condition (3.1), it will falsely conclude that the algorithm has terminated.

We circumvent this problem, by letting Master consider the in-out tuples from two consecutive iterations. Let $V_i = (in_1^i, out_1^i, \dots, in_W^i, out_W^i)$ denote the in-out tuples from i^{th} iteration.

Theorem. *If $V_i = V_{i+1}$ for some $i \geq 1$, then there exists time instant t at which $in_j^i = in_j^t$ and $out_j^i = out_j^t$ for all $1 \leq j \leq w$.*

Proof. Among the in-out tuples from iteration i , let the tuple (in_w, out_w) from worker w be the latest with time stamp t^{max} . Similarly, among the in-out tuples from iteration $i + 1$, let the tuple $(in_{w'}, out_{w'})$ from worker w' be the earliest with time stamp t^{min} . At any time instant $t \in [t_{max}, t_{min}]$, we have that $in_j^i = in_j^t$ and $out_j^i = out_j^t$ for all $1 \leq j \leq w$. \square

Termination Detection Test From the above theorem, we note that if the in-out tuples from two consecutive iterations are same, then the in-out tuples in fact correspond to a real time instant t . So the Master worker checks for the Condition (3.1) and if satisfied declares the algorithm as terminated.

Algorithm 3 Halt Detection: Master

```
1: procedure STATUSREQUEST
2:   for all  $w_i \in Workers$  do
3:      $Send(statusReq, w_i)$ 
4:   end for
5: end procedure
6: procedure STATUSREPLY( $StatusReply\ msg$ )
7:    $R \leftarrow R \cup GetVector(msg)$ 
8:   if  $|R|$  is equal to  $|W|$  then
9:      $CheckHalt()$ 
10:  end if
11: end procedure
12: procedure CHECKHALT
13:   if ( $R_{old}$  is defined and
14:     $R_{old}$  is equal to  $R$  and
15:     $\sum R(in)$  is equal to  $\sum R(out)$ ) then
16:      $Halt \leftarrow true$ 
17:   else
18:      $R_{old} \leftarrow R$ 
19:      $R \leftarrow \Phi$ 
20:      $SendRequest()$ 
21:   end if
22: end procedure
```

Algorithm 4 Halt Detection: Worker

```
1: procedure STATUSREQUEST
2:    $StatusReply \leftarrow CreateStatusReply(< in, out >)$ 
3:    $Send(StatusReply, master)$ 
4: end procedure
```

Chapter 4

Adaptive State Space Exploration

In this chapter, we present our adaptive scaling strategies and would describe why we choose horizontal scaling over vertical scaling. We also present what triggers addition of new nodes to the cluster.

4.1 Introduction

Be it centralized or distributed safety verification procedures, if any of the compute nodes runs out of main memory, then the verification procedure comes to a halt inconclusively. This wastes a lot of compute power and end user time. So enough main memory resources have to be allocated for the verification procedure to complete. However, it is not possible to know apriori, the required main memory resources, from a given Promela model. So, as the memory pressure increases, we have to dynamically scale-up the available memory. This is not possible if the compute node is a physical machine. However, if it is a virtual machine, based on the resources available on the host physical machine, we might be able to increase its capacity. Although, vertical scaling of memory resources is theoretically possible using memory ballooning technique [14], to the best of our knowledge, none of the cloud service providers support this feature. Further, the available resources on the host physical machine constrains vertical scaling.

In this work, we circumvent the aforementioned problems, by resorting to horizontal scaling of resources, wherein we add one or more virtual machines to our cluster whenever the memory pressure on a compute node crosses a threshold. Then we migrate some of the actors on the existing virtual machines to the newly added machines for load redistribution. We present the details of our approach in the rest of the sections.

4.2 Horizontal Scaling through Migration

Since our model checker is based on SpinJa, each compute node in the cluster holds multiple workers within a single Java Virtual Machine (JVM). Each compute node in the cluster sends its memory usage, as perceived within the JVM, to the Master worker. If the memory usage crosses a threshold, then the

Master worker initiates a cluster expansion and actor migration procedure. This is a time critical step because if the necessary action is not taken in time then the verification procedure will crash due to insufficient available memory. Since Akka sends messages in order between any actor pair, a critical memory usage message may end up waiting in the message queue as the system comes to a crash. We avoid this situation by using a side channel for communicating memory usage reports.

The Master compares the reported memory usage against a preset threshold and if it exceeds, sends a `Halt` message to all the workers. We chose against live migration of workers as the memory pressure increases exponentially faster when compared with the state transfer resulting in system crash ultimately. When a worker receives the `Halt` signal, it processes the `Compute` messages differently. If the vertex in the `Compute` message is already present in its explored set (ES_w), it simply ignores it. If it is not, then it caches it in a table without exploring the outgoing edges, and increments the local in_w counter in advance. Then the Master runs the termination detection algorithm from Section 3.2. Once the Master determines that the computation at all the workers has come to complete halt, it increases the size of the cluster by instantiating new virtual machines using the cloud providers' API and then initializes them by bootstrapping necessary JVMs for the model checking procedure to proceed. The number of new virtual machines added is a policy decision and the relevant studies are presented in the experimental results chapter . Then the Master migrates workers from the existing virtual machines to the new virtual machines for load redistribution. Note that migration is happening not only from the node which sensed excessive memory pressure but also from other nodes. Since the vertex assignment to different workers is done using a hash function resulting in uniform distribution, it is usually the case that memory pressure at a node indicates a similar situation at other nodes.

The Master initiates the migration of a worker by sending a `Move` message. In reality, what we do is replicate the state of a source worker into a destination worker. A source worker whose state has to be replicated gets the address of the destination worker in the `Move` message from the Master. After the state is replicated, the source worker is destroyed. The source worker sends its local state to the destination worker using messages. The set of explored vertices is the most significant component in the local state associated with an actor. We have experimented with batching the explored vertices into a single message, against sending individual messages; batching gave us a 10x faster migration when compared to sending each vertex individually. We couldn't send the entire state space in a single message due to Akka's message size limits.

Once the migration is complete, the destination worker sends a `Received` message to the Master. When all the migrations are complete, the Master sends a `Resume` message to continue state space exploration. Workers on receiving the `Resume` message, replays a `Compute` message to its mail box, for each cached vertex that it has received after the `Halt` message. Further, it suitably decrements the in_w counter too. Thus the normal execution is resumed.

4.3 Sensing Memory Pressure

In each compute node of the cluster, the model checking process is encapsulated in a JVM running on that node. There are no other significant running processes which consume CPU or memory. Since JVM automatically manages memory using garbage collectors, it is tricky to estimate the actual memory consumed at a given instance. For example, an object could be residing in memory but might be out of context waiting to be collected, thereby bloating the actual memory consumption. We used the `GarbageCollectorMXbean` that emits garbage collection notifications containing statistics about reclaimed memory and the current memory usage whenever a full garbage collector is invoked. This method of collecting memory usage immediately after a full garbage collection gives us reliable data, since all out of context objects are removed from memory.

Chapter 5

Experimental Results

In this chapter, we present experimental results obtained when our adaptive asynchronous safety algorithm is applied on a subset of BEEM benchmarks [18]. Table 5.1 shows the list of benchmarks studied. Among the benchmarks we have considered, `anderson.8` is the largest, with vertex and edge counts of approximately half-a-billion and 2 billion respectively. The states and transitions count in the Table 5.1 are obtained by running the respective Promela models in SpinJa model checker with no partial order reduction. We used Compute Engine which is an Infrastructure-as-a-Service platform offered by Google to acquire resources on-the-fly using their API.

For our experimental analysis, we used a cluster of virtual machines each with 32 vCPUs and 50 GB main memory. Our actor architecture for safety checking consists of a Master and multiple worker actors. We fixed the number of workers to 40. Each virtual machine in the cluster runs a JVM and the predetermined number of workers are equally distributed among the cluster nodes for load balancing purposes. Each JVM sends the heap utilization statistics to the Master worker periodically on a separate logical channel. If the Master infers that memory pressure is building up on a JVM, it initiates cluster expansion process. In our experiments, this happens whenever the heap utilization on a JVM goes beyond 70 percent of the available memory. If we put a higher heap utilization threshold, say 90 percent, then there is an adverse impact on the performance. This is due to the fact that as the heap gets almost full, JVM invokes garbage collector more frequently, which is an extremely expensive procedure. We used two cluster expansion policies. In the first policy, we expand the cluster by adding one node at a time, whereas in the second policy we add two nodes at a time. We can envisage many other cluster expansion policies. Overall, if we expand too fast then we may allocate more resources than necessary. It can lead to sub-optimal performance due to unnecessary state transfer while load rebalancing and increased inter-node communication costs. In addition, the resource rental cost naturally goes up. If we expand too slow, then the number of times that we have to do load rebalancing increases and takes longer time for the system to converge. For example, if we need k nodes for successfully performing a safety check procedure on a given Promela model, with a cluster expansion of policy of l nodes at a time, we require $\lceil k/l \rceil$ iterations to reach the appropriate cluster size. At the same time, we may end up incurring an additive overhead of $l - 1$ nodes on the cluster size. Thus we have to strike a trade-off

Table 5.1 State space sizes for BEEM models

S.No	Model Name	States	Transitions
1.	anderson.8	538,699,037	2,972,732,142
2.	fischer.7	386,297,015	2,096,962,041
3.	bakery.8	253,131,202	1,074,134,375
4.	loyd.3	239,500,803	678,585,604
5.	peterson.6	174,495,861	747,072,151
6.	frogs.5	182,772,130	387,950,982
7.	hanoi.4	129,140,170	387,420,494

between expanding too fast and slow. One possible application specific heuristic is to use an adaptive expansion policy based on the number of currently unexplored states.

In order to understand the overheads involved in dynamic scaling of resources we split the total execution time associated with a safety checking procedure into the following three components.

1. **Execution Time (ET):** This is the time during which all the nodes are doing normal state space exploration with no memory pressure issues at any of the nodes.
2. **Transient Execution Time (TT):** When the Master infers a memory pressure in any of the JVMs, it sends a `halt` message to all the workers. Then the workers stop exploring new states and processes the messages already in the mail box by caching the unexplored vertices. This phase continues until the termination detection algorithm concludes that all the messages are accounted for. This phase of the safety checking procedure corresponds to what we call as transient execution time (transient time for short).
3. **Migration Time (MT):** After the transient execution phase, the Master initiates the migration of workers to the newly added cluster nodes for load distribution. This is called as the migration time.

The actual wall clock execution time T of the safety checking procedure is the sum of the execution, transient and migration times, i.e., $T = ET + TT + MT$. In this the migration time is the real overhead. There is a partial overhead during the transient time too as the available compute power is used to process only messages but no state space exploration is done.

Figure 5.1 shows the split of the total wall clock execution time for various benchmarks using the one and two node cluster expansion policies. It can be noted that the total execution time using two node cluster expansion policy is always smaller than that of single node addition policy. Table 5.2 shows the overhead due to migration and transient execution times. The overhead due to migration time is computed as $\frac{MT}{T} \times 100$. The combined overhead due to migration and transient execution times is computed as $\frac{MT+0.5TT}{T} \times 100$. The 0.5 multiplicative factor for transient time is included in order to account for the fact that during this phase partial computations do happen. We can notice

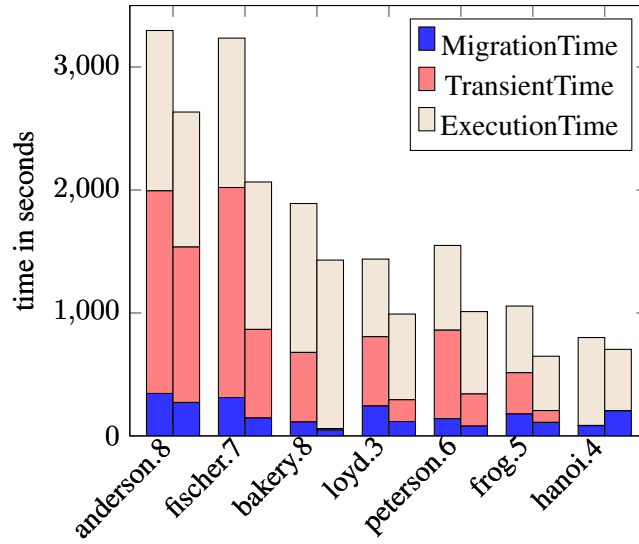
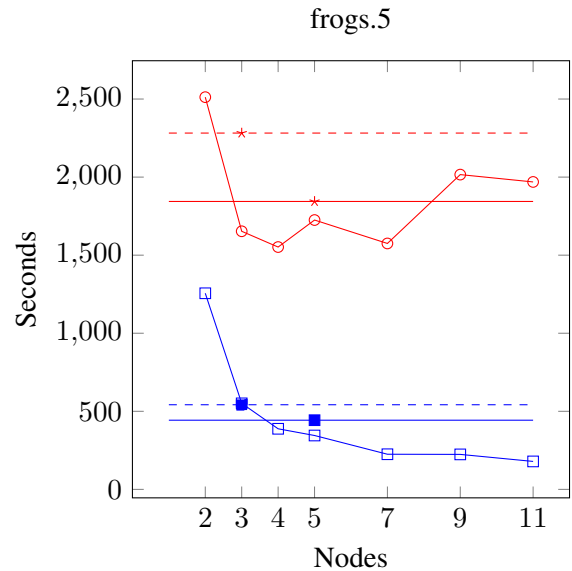
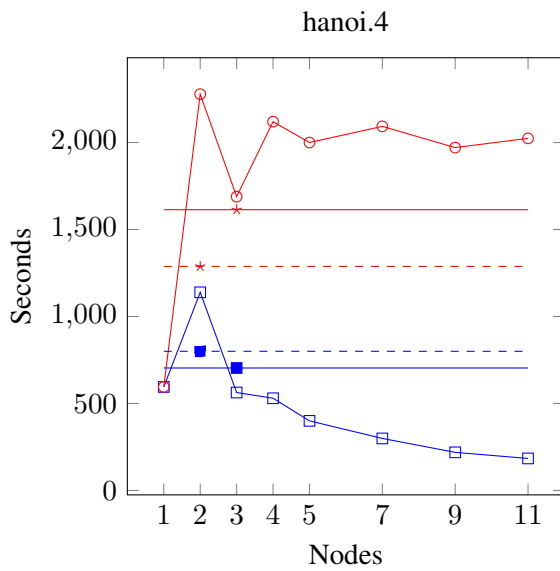


Figure 5.1 Execution, transient and migration times for various benchmarks using one and two nodes at a times cluster expansion policy.



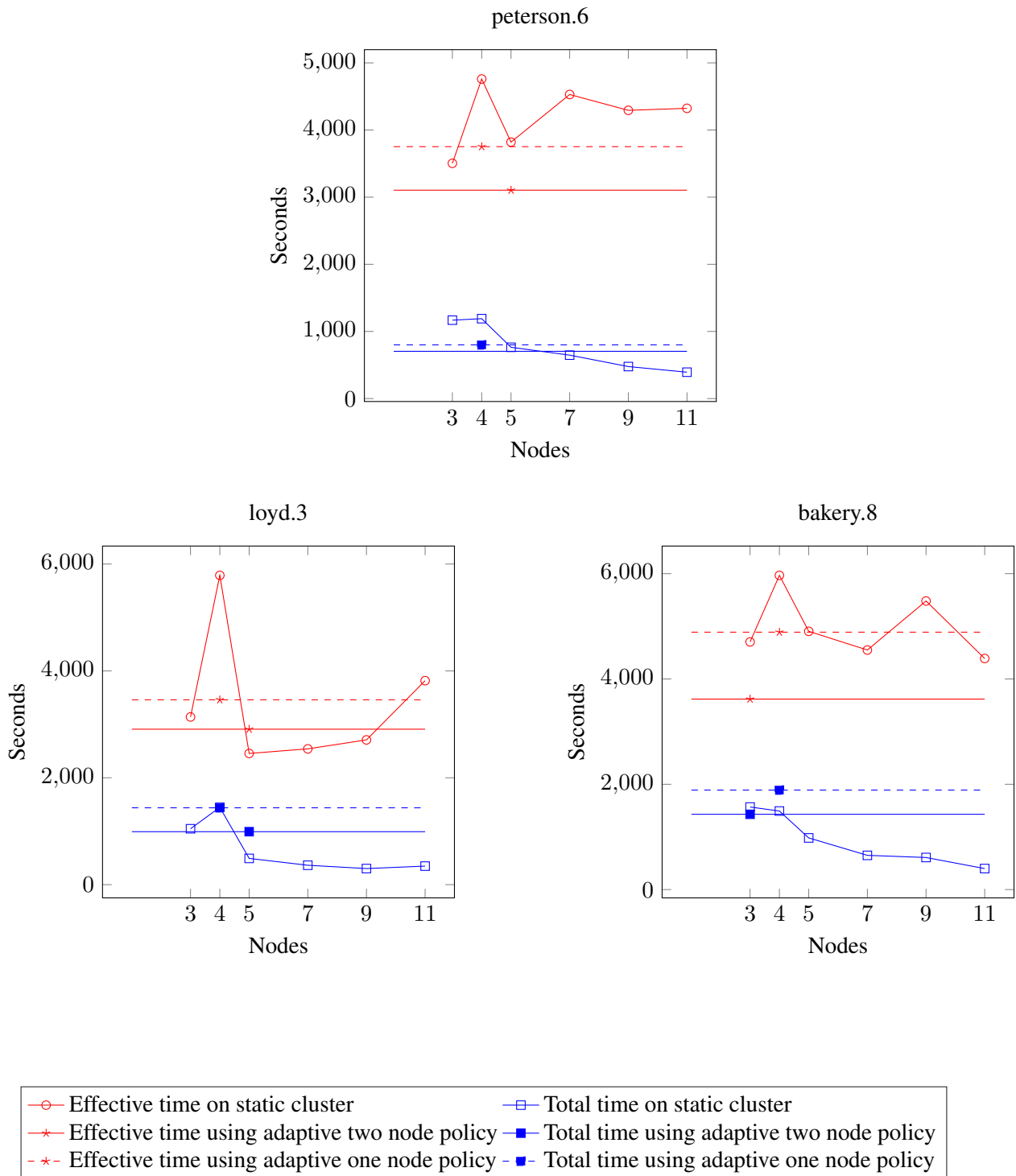


Figure 5.2 Comparison of total time T and effective time T_e for different static cluster sizes, adaptive single and two node cluster expansion policies.

Table 5.2 Percentage overhead due to migration and transient execution phases for one and two node expansion policies.

Benchmark	Migration Overhead		Migration + Transient Time Overhead	
	One Node	Two Node	One Node	Two Node
anderson.8	10.46	10.33	35.46	34.33
fischer.7	9.64	7.07	36.04	24.5
bakery.8	6.03	3.36	20.97	3.78
loyd.3	16.97	11.8	36.55	20.73
peterson.6	9.04	7.91	32.29	20.82
frog.5	16.94	16.96	32.8	24.36
hanoi.4	10.37	28.99	10.5	29.14

that the migration overhead for anderson.8 and fischer.7 which are the largest benchmarks and actually get benefited by dynamic scaling is around 10 percent. If we include the transient time overhead too, then it goes to around 35 percent. Generally speaking, two node expansion policy leads to less number of migrations and hence the associated transient times, hence less overheads compared to one node expansion policy. This trend can be observed in all the benchmarks except hanoi.4. In spite of that, the total execution time in hanoi.4 using 2 node addition policy is smaller than that of one node policy. If T is the total execution time, let $N(t)$ denote the number of nodes in the cluster at time instant t , $0 \leq t \leq T$. The effective execution time T_e which gives the actual amount of compute cycles used is defined as follows.

$$T_e = \int_{t=0}^T N(t) dt$$

For a static cluster of size k , we can easily see that $T_e = kT$. The effective time is an important metric as it gives the actual resource rental cost on the cloud. Figure 5.2 shows the total (legend \square) and effective (legend \circ) times taken by various benchmarks with increasing static cluster sizes. In the hanoi.4, the total execution time increases as we increase the cluster size from one to two. This is due to the unnecessary inter-node communication cost incurred in the two node cluster. The total time subsequently falls down from cluster size 3 onwards. It is interesting to note that the effective time remains the same while the total time drops with the cluster size. For the other benchmarks, which require at least two machines to successfully finish the safety checking procedure, we can note that the total time either decreases or remains constant with increasing static cluster sizes. Except for lloyd.3 on 4 nodes, the effective time more or less remains within a range with increasing cluster sizes.

The two horizontal axes denoted by the legends \blacksquare and \blackstar correspond to the total and effective times using adaptive one node policy. Similarly the two axes denoted by the legends \blacksquare and \blackstar correspond to the total and effective times using adaptive two node policy. The place where the '*' mark is placed on the horizontal axis denotes the number of nodes the adaptive strategy has finally

Table 5.3 Cost of safety verification, Total and effective execution times for anderson.8 and fischer.7.

	anderson.8			fischer.7		
	Total	Effective	Cost	Total	Effective	Cost
One Node	3,297	16,728	\$4.2	3,235	14,503	\$3.6
Two Node	2,634	21,865	\$5.5	2065	10,609	\$2.7
Static-13	14,572	189,433	\$47.4	4,614	59,984	\$15

converged to. For example, for loyd.3, the two node policy converged to 5 nodes whereas the one node policy converged to 4 nodes finally. As discussed earlier, usually the total time using one node policy would be greater than that of the two node policy. The effective time is also smaller in all of the cases except for hanoi.4. This generally indicates that the two node policy not only gives good performance but is also cost effective. When we compare the two node policy with the statically sized clusters, the total time is larger when compared with medium to large sized clusters, but the effective time is usually better indicating greater cost effectiveness. We can trade-off cost effectiveness with total execution time by using a more aggressive cluster expansion policy. Table 5.3 shows the total and effective execution times for anderson.8 and fischer.7. These models requires at least 13 nodes to carry out the safety checking procedure successfully. Apriori determination of this minimum node requirement is impossible as it is model specific. So we have to iteratively check through an expensive trial and error procedure. If the minimum number of nodes required is N_{min} , and if it takes T_i time to determine if i nodes are sufficient to finish the safety checking procedure, then the total total and effective execution times are given as follows.

$$T = \sum_{i=1}^{N_{min}} T_i \qquad T_e = \sum_{i=1}^{N_{min}} iT_i \qquad (5.1)$$

Many times, a JVM on a virtual machine performs lot of garbage collection whenever the memory pressure increases due to lack of available heap space before it actually throws an out-of-memory error. So the T_i terms in the summation (5.1) could be really large. This directly translates to huge impact on total time, effective time and the associated resource rental costs. This is not at all a problem in our adaptive approach where resources are added as per the need. We can note from the Table 5.3 that the total and effective times for anderson.8 using the two node policy is 5.5x and 8.8x times better than the static approach. In terms of the actual resource rental cost, our approach reduces it by 8.6x times. We would like to emphasize here that the effective time for static approach is not computed using (5.1) and assume that we got the cluster size through an oracle. The performance of the adaptive approach would grows quadratically if we use (5.1) to compute the baseline. The adaptive approach starts with a single node and finally converges to the required 13 nodes. An interesting observation is that the adaptive approach does better on the total time front also. This is because in the adaptive approach the communication messages are localized when the workers are colocated, and the messages which need to travel on a wire increases as the cluster size increases. However, in the static cluster case, from

the very beginning of the computation, the messages are equally distributed among the available 13 nodes and the number of non-local messages will be substantial as a result. For `fischer.7`, our adaptive approach performs 2.2x, 5.7x and 4.2x times better than that of the static approach with respect to total and effective times, and resource rental cost respectively.

Chapter 6

Conclusion

In order to make applications cloud native with inherent support for scalability, we usually take the stateless design approach. But there are certain applications like safety and liveness verification procedures in the model checking space which are inherently stateful in nature. In order to make them stateless, we have transfer that state to an external key-value store service. While theoretically it is possible, it will seriously impact the performance by decreasing the compute to communication ratio. In the thesis, we circumvent this problem by partitioning the state space and coupling each partition to an actor through a mapping function. When there is a resource pressure and new nodes are added, the available actors are redistributed for load distribution. This approach can be potentially applied to other inherently stateful designs. The entire process happens in a push button fashion on any cloud platform making the whole thing offerable as a service. In future, we would like to experiment with using a cluster of heterogeneous nodes and dynamic node addition policies which considers the rate of memory consumption.

In the thesis we demonstrate dynamically horizontal scalable architecture for a memory intensive stateful application using the cloud. we tackle state space explosion problem in model checking using the above architecture. In the absence of such an architecture model checker runs out of memory and the state space needs to be re-computed with increased number of resources. The proposed architecture uses actors to horizontally scale, by adding resources to the cluster and moving them across nodes during a memory pressure.

We tested the scalability of the application on a homogeneous cluster of nodes and compared the cost of running model checking on an adaptive system to that of a static cluster of nodes. We showed that the cost of running the dynamically scalable application to be far less than iteratively finding the minimum resources required to successfully verify a model.

In future we want experiment using a cluster of heterogeneous nodes and a dynamic node addition policy which considers the rate of memory consumption. we also want to explore a more exhaustive set of node addition policies and come up with model based heuristics for cost effectiveness. We would want to explore integration with platforms like [12] to provide verification as a service.

Bibliography

- [1] V. Acretoae and H. Störrle. Hypersonic: Model analysis and checking in the cloud. In D. Kolovos, D. Di Ruscio, N. Matragkas, J. De Lara, I. Ráth, and M. Tisi, editors, *Proceedings of the 2nd Workshop on Scalability in Model Driven Engineering (BigMDE 2014)*, pages 3–13, 2014.
- [2] G. A. Agha. Actors: A model of concurrent computation in distributed systems. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1985.
- [3] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček. Divine—a tool for distributed verification. In *International Conference on Computer Aided Verification*, pages 278–281. Springer, 2006.
- [4] C. Bellettini, M. Camilli, L. Capra, and M. Monga. Symbolic state space exploration of rt systems in the cloud. In *2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 295–302, Sep. 2012.
- [5] C. Bellettini, M. Camilli, L. Capra, and M. Monga. Mardigras: Simplified building of reachability graphs on large clusters. In P. A. Abdulla and I. Potapov, editors, *Reachability Problems*, pages 83–95, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- [6] M. Ben-Ari. *Principles of the Spin Model Checker*. Springer, 2008.
- [7] M. Camilli, C. Bellettini, L. Capra, and M. Monga. CTL model checking in the cloud using mapreduce. In *16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2014, Timisoara, Romania, September 22-25, 2014*, pages 333–340, 2014.
- [8] C. Coti, S. Evangelista, and L. Petrucci. One-sided communications for more efficient parallel state space exploration over rdma clusters. In *European Conference on Parallel Processing*, pages 432–446. Springer, 2018.
- [9] M. de Jonge and T. C. Ruys. The spinja model checker. In *International SPIN Workshop on Model Checking of Software*, pages 124–128. Springer, 2010.
- [10] D. L. Dill. The murphi verification system. In *Proceedings of the 8th International Conference on Computer Aided Verification, CAV '96*, pages 390–393, London, UK, UK, 1996. Springer-Verlag.
- [11] G. Holzmann. *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.
- [12] K. Hu, L. Lei, and W.-T. Tsai. Multi-tenant verification-as-a-service (vaas) in a cloud. *Simulation Modelling Practice and Theory*, 60:122 – 143, 2016.

- [13] F. Lerda and R. Sisto. Distributed-memory model checking with spin. In *International SPIN Workshop on Model Checking of Software*, pages 22–39. Springer, 1999.
- [14] H. Liu, H. Jin, X. Liao, W. Deng, B. He, and C. Xu. Hotplug or ballooning: A comparative study on dynamic memory management techniques for virtual machines. *IEEE Transactions on Parallel and Distributed Systems*, 26(5):1350–1363, May 2015.
- [15] F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2(3):161–175, Sep 1987.
- [16] I. Melatti, R. Palmer, G. Sawaya, Y. Yang, R. M. Kirby, and G. Gopalakrishnan. Parallel and distributed model checking in eddy. *International Journal on Software Tools for Technology Transfer*, 11(1):13–25, Feb 2009.
- [17] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73, Mar. 2015.
- [18] R. Pelánek. Beem: Benchmarks for explicit model checkers. In D. Bošnački and S. Edelkamp, editors, *Model Checking Software*, pages 263–267, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
- [19] A. Singla, K. Desai, S. Purini, and V. Choppella. Distributed safety verification using vertex centric programming model. In *2016 15th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 114–120. IEEE, 2016.
- [20] U. Stern and D. L. Dill. Parallelizing the murphi verifier. In *Proceedings of the 9th International Conference on Computer Aided Verification, CAV '97*, pages 256–278, London, UK, UK, 1997. Springer-Verlag.
- [21] M. Thureau. Akka framework. *University of Lübeck*, 2012.
- [22] M. Xie, Q. Yang, J. Zhai, and Q. Wang. A vertex centric parallel algorithm for linear temporal logic model checking in pregel. *Journal of Parallel and Distributed Computing*, 74(11):3161–3174, 2014.
- [23] W. Zhang, W.-k. Ma, H.-l. Shi, and F.-q. Zhu. Model checking and verification of the internet payment system with spin. *Journal of Software*, 7, 09 2012.