Eddy_SPIN and the Eddy Framework

R. Palmer, G. Sawaya, Y. Yang, R. M. Kirby, G. Gopalakrishnan

School of Computing, University of Utah
{rpalmer,sawaya,yuyang,kirby,ganesh}@cs.utah.edu

Abstract. The formal verification of software using tools that explore the entire state space is struggling with the difficulty of space state explosion; even relatively simple software may be composed of billions or more states. We have taken the novel approach with this project to take advantage of BOTH distributed computing (utilizing multi-machine clusters) and parallel processing (as in SMP shared cores / multicores). Since multi-core clusters are readily available, we designed Eddy_SPIN to divide the state space among nodes in a cluster and within each node we divide the tasks of state exploration and communication to separate SMP shared memory threads (POSIX Pthreads) in order to make the most efficient use of available hardware. We believe that this is the first model checker that combines parallel and distributed processing, being unable to find a similar paradigm in the literature. Another innovation we added from our own experience in distributed model checking was a performance tunable communication queue. This allows Eddy_SPIN to fully utilize the packet sizes that a particular cluster’s networking provides, making communications as efficient as possible. Eddy is constructed of several modules, including communication, model checking, and communication queue. In this report we detail the history of Eddy (including the design rationales), the algorithms that are behind its communication and model checking, pseudo-code of the modules, challenges encountered and the current state of the project.

1 Introduction

This paper looks at what is behind the framework that is Eddy, the distributed model checking framework designed to be extensible in order to ‘plug in’ different model checkers. It uses MPI[1] for inter-node communication and parallelizes the work on each node with PThreads[2] (the first model checker designed this way that we are aware of, though there have been other efforts towards distributed model checking[3][4]). SPIN[5] was the first model checker that we used with Eddy, but we also successfully developed a version using the Mure verifier tool[6].

The largest motivation in creating Eddy was to give formal software verification tools greater power in confronting one of its biggest problems, the state space explosion. The most commonly available model checkers are serial tools that are constrained to use machines that are generally limited to 2GB of memory, a space that is quickly consumed by the states
of a relatively simple model. For instance, if we were to try to rigorously explore a model of 100 million states, we would be constrained to a state vector size of 20 bytes on a 2GB machine (in order to store all the states). This is much too small for most SPIN models unless we were to resort to a ‘lossy’ storage scheme such as bitstate hashing. Therefore, our first reason for extending model checkers to a distributed architecture was to give us the space to explore a larger state space. Another obvious reason to distribute the model checking task is for speed, as large models, even with sufficient memory, can take a very long time to verify.

One of our original goals in expanding the capability of SPIN was to verify the DEOS[7] model. This is a model of an advanced aircraft thread scheduler produced by the Honeywell corporation, a model consisting of over trillions states. In order to hopefully get a baseline of this model, we ran it for approximately 6 weeks on a serial version of SPIN using an Intel machine with 16 processors and 16GB of RAM. We ran out of space before SPIN could complete the verification process.

Eddy is designed so that a serial model checker may be hooked in to provide distributed capabilities. As long as there is access to a function within the MC that we can supply a state, S, and get back a list of successors, U (as in $U = \text{getSuccessors}(S)$), we can easily run this with Eddy.

This report continues from this point to basic design considerations with Eddy in section 2 (For more detailed design issues, please refer to ‘Refactoring Spin for Safety’[8] and ‘Parallel and Distributed Model Checking in Eddy’[6]). Then in section 3 we detail experimental results. Section 4 explains how to run EddySPIN; section 5 concludes, adding thoughts on further work.

The entire EddySPIN package is available for download (unsupported) at: http://www.cs.utah.edu/formalverification/EddySpin/eddySpin.tar.gz.

2 Design Considerations

There are several reasons for the architecture we chose to use with EddySpin. Our group has previous experience with distributed model checking, and those, most notably PV[9] (a SPIN like tool), taught us several things we did not want to include with this implementation.

PV was also a distributed model checker that utilized MPI. However, there were some shortcomings that we wanted to address with a built from scratch tool. The biggest problem with PV was that it spent the better part of its time doing communications. The reason for this is that PV distributed every successor state one at a time, using blocking communications (such as mpi_send) with a single thread. With this mass of single states being sent across the network and the sending nodes waiting for them, the model checking work suffered.

To address this issue, we added two important features to Eddy’s design: first is the use of multiple threads per node in order to divide the tasks
of communications and model checking; the second is creating more efficient communications through the use of a thread-safe and performance tunable communication queue.

The communication queue allows us to tailor communications to whatever hardware we may be running on. We can adjust the size of the buffers to be most efficient by filling whatever size of packets the topology of our cluster utilizes. Our communication queue implementation has a very simple interface, utilizing basically three methods for push, pop and return buffer. When running Eddy_SPIN on n nodes, there will be n-1 queues per node, or each node has a communication queue for each other node in the cluster.

By utilizing separate threads for communication and model checking work, the work of model checking does not have to be interrupted to deal with communication tasks. While the worker thread, or model checker, is exploring states, the communication queue is busy with the job of initializing sends, receiving states to be enqueued to the worker, checking for complete sends to recover buffers, and handling the control of the application looking for the end of the verification and for error conditions.

Our multiple thread per node implementation is simple, requiring very little synchronization between threads as their purposes are orthogonal. For some time after the implementation of Eddy_SPIN we were vexed with a race condition due to insufficiently fine granularity in our mutex protection. The condition occurred on the part of the communication thread when it was checking the existence of an imported state in the model checker’s hash table.

Besides these specialized to be addressed, we also set out to make Eddy_SPIN as modular as possible to aid in its understanding and to make it easy to use with other model checkers. In order to have Eddy work with a particular model checker, we need mostly a function, given a state, that will return a list of successors and a function that will determine who the owner of a state is (which node in the cluster) for distribution. Once these methods are exposed it is a straightforward process to attach the model checker to the Eddy interface and compile.

### 2.1 Disk Cache Details

One difficulty that arose when we were debugging Eddy_SPIN was a race condition where we had two threads adding states to the local hash table. The function `addToLocalQueue()` (in `spin_extension.c`) is called by the communication thread as states are received from other nodes in the cluster. Concurrently, the function `distribute()` is also adding to the hash table from the worker thread side. Our first approach to this problem was not satisfactory.

Before we realized that this race condition existed, the code in both `distribute()` and `addToLocalQueue()` would, for each state: 1) check for the state’s existence in the hash table; 2) add it to the hash table if not there; and 3) add it to the local consumption stack for exploration (also if it wasn’t in the table).

Our first approach was to rearrange this algorithm slightly so that in `distribute()` and `addToLocalQueue()`, the states would simply be added
to the consumption stack. This means that the main worker thread loop, located in `maximalAmpleSet()` (also in `spin_extension.c`) has to check each state it popped from the stack for its existence in the hash table before finding successors to avoid redundancy. The problem we encountered with this algorithm was that the consumption stack became unwieldy and utilized too much of the memory resources on each node which in turn nulled one advantage we were seeking in parallelization which is having more available memory for state exploration.

To address this problem, we took a two branched approach: 1) was to create a disk caching scheme combined with an auxiliary hash table; and 2) was to increase the granularity of our mutex protection in concurrent sections of `spin_extension.c` and return it to the previous algorithm.

This section details the implementation of the disk caching scheme along with the auxiliary hash table we used for this.

The algorithm for this second approach is explained as follows:

```
***************
(Model Checker Thread):

Main Loop:
while NOT stackIsEmpty()
  pop st from consumption stack
  if st is enqueued by mc thread
    if is it in hash table
      delete state, continue while
    fi
  fi
  generate successors
  distribute successors
wend
end Main Loop

stackIsEmpty:
if consumption queue is empty
  read states from diskfile
fi
while consumption queue is empty
  wait on new work signal
  read states from diskfile
  if TERMINATED return true
wend
return (consumption queue is empty)
end stackIsEmpty

distribute(states):
for each state st in states
  if I am the owner
    if st is in hash table
      delete state
```
else
    add st to hash table
add st to consumption stack
fi
else
    enqueue st for sending by communication manager
fi
next state
end distribute

***********************
(Communication Thread):

addToLocalQueue(states):
for each state st in states
    if st NOT in aux hash table
        add st to aux hash table
        put st on diskfile
    else
        delete st
    fi
next state
end addToLocalQueue
***********************

In other words, with this version we maintain a separate hash table for use by the communication thread for checking incoming states only, to eliminate redundant states generated from external model checkers (those located elsewhere on the network). Also, all the states that are to be explored that the communication manager receives are added to the disk cache instead of the consumption stack. The function within spin\_extension.c, stackIsEmpty(), will read in the last page file in the disk cache and push all the states there onto the consumption stack for the model checker to use.

This algorithm was successful in addressing the issue of the race condition and the resulting problem of a runaway consumption stack when concurrent hash table verification of states was eliminated. One issue with the disk caching algorithm is that it is dependent on an adequate volume of free temp disk space on each node in the cluster that Eddy is operating in.

3 Experiments

There are several interesting things that the data from our experiments indicate. To begin with, we are not getting the neat linear decrease in processing time that we were hoping for when verifying a Promela model across an increasing number of nodes. Also, we have gathered some data to demonstrate the efficiency of our refactoring of SPIN, and it shows
that some models are up to 20 times less efficient than plain serial SPIN.

Table 1. Comparison between original spin and spin extensions

<table>
<thead>
<tr>
<th>model</th>
<th># of states</th>
<th>run time</th>
<th>memory usage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>original SPIN</td>
<td>with extension</td>
<td>ratio</td>
</tr>
<tr>
<td>mobile1</td>
<td>30,652</td>
<td>0.32</td>
<td>7.02</td>
</tr>
<tr>
<td>mobile2</td>
<td>10,865</td>
<td>0.09</td>
<td>1.48</td>
</tr>
<tr>
<td>peterson4</td>
<td>7,968,740</td>
<td>91.89</td>
<td>703.91</td>
</tr>
<tr>
<td>sort</td>
<td>107,713</td>
<td>2.21</td>
<td>23.25</td>
</tr>
<tr>
<td>leader2</td>
<td>1,975,050</td>
<td>29.11</td>
<td>383.05</td>
</tr>
</tbody>
</table>

Also included in this test data is the results of parallelizing Murϕ on varying numbers of nodes; Eddy_Murϕ demonstrates much more efficient scaling. We have successfully verified some fairly large models (including a leader election algorithm that generates 8E7 states) in a much shorter time than would be possible on a serial implementation of SPIN.

Fig. 1. Experimental results for performances comparison of Eddy_Murphi with standard Murphi, carried out on the same cluster

Grid1 and Grid2 show the results of running two different versions of Eddy_SPIN, one with increased granularity of critical sections (see DiskCaching above) and the other, which was initially created to address the issue of the race condition described in the Disk Caching implementation details, which adds states to the consumption stack without first checking the hash table. It mitigates the problem previously explained by using an auxiliary hash table to avoid the intake of redundant states from other nodes.
The tests in grid1 and grid2 were run on blocks of 4 tests, and the averages and StDev is generated from these groups (all on the same cluster, same number of nodes, and same model). There are also some singleton tests, which are tuning parameter tests ([modelName][queue lines][states per line]).

The machines that these tests were ran on are Inferno, a 128 node cluster with each machine providing two 2.33 GHz Intel Pentium processors and 2 GB of RAM; the other cluster is Nebula with 31 dual 1.65 GHz Pentium processors each node carrying 1 GB RAM.

<table>
<thead>
<tr>
<th>OutFile</th>
<th>states</th>
<th>queueLines</th>
<th>Max_states/Line</th>
<th>avgSt/msg</th>
<th>asm_StDev</th>
</tr>
</thead>
<tbody>
<tr>
<td>pet4neb16AHT_A</td>
<td>7968740</td>
<td>4</td>
<td>256</td>
<td>129.148</td>
<td>8.374</td>
</tr>
<tr>
<td>lead2neb16AHT_A</td>
<td>1975050</td>
<td>4</td>
<td>256</td>
<td>112.898</td>
<td>2.303</td>
</tr>
<tr>
<td>lead2neb8AHT_A</td>
<td>1975050</td>
<td>4</td>
<td>256</td>
<td>137.384</td>
<td>1.895</td>
</tr>
<tr>
<td>lead2neb4AHT_A</td>
<td>1975050</td>
<td>4</td>
<td>256</td>
<td>165.944</td>
<td>0.821</td>
</tr>
<tr>
<td>lead2neb2AHT_A</td>
<td>1975050</td>
<td>4</td>
<td>256</td>
<td>245.94</td>
<td>3.658</td>
</tr>
<tr>
<td>pet4inf32AHT_A</td>
<td>7968740</td>
<td>4</td>
<td>256</td>
<td>70.338</td>
<td>6.915</td>
</tr>
<tr>
<td>lead2neb16MT_A</td>
<td>1975050</td>
<td>2</td>
<td>256</td>
<td>141.041</td>
<td>0.796</td>
</tr>
<tr>
<td>lead2neb8MT_A</td>
<td>1975050</td>
<td>2</td>
<td>256</td>
<td>161.128</td>
<td>2.746</td>
</tr>
<tr>
<td>lead2neb4MT_A</td>
<td>1975050</td>
<td>2</td>
<td>256</td>
<td>199.103</td>
<td>0.952</td>
</tr>
<tr>
<td>lead2neb2MT_A</td>
<td>1975050</td>
<td>2</td>
<td>256</td>
<td>241.941</td>
<td>0.465</td>
</tr>
<tr>
<td>lead6inf64MT_A</td>
<td>80684137</td>
<td>2</td>
<td>256</td>
<td>117.618</td>
<td>1.478</td>
</tr>
<tr>
<td>leader2_1_128</td>
<td>1975050</td>
<td>1</td>
<td>128</td>
<td>132.065</td>
<td></td>
</tr>
<tr>
<td>leader2_1_256</td>
<td>1975050</td>
<td>1</td>
<td>256</td>
<td>252.744</td>
<td></td>
</tr>
<tr>
<td>leader2_1_512</td>
<td>1975050</td>
<td>1</td>
<td>512</td>
<td>493.027</td>
<td></td>
</tr>
<tr>
<td>leader2_2_128</td>
<td>1975050</td>
<td>2</td>
<td>128</td>
<td>85.875</td>
<td></td>
</tr>
<tr>
<td>leader2_2_256</td>
<td>1975050</td>
<td>2</td>
<td>256</td>
<td>165.832</td>
<td></td>
</tr>
<tr>
<td>leader2_2_512</td>
<td>1975050</td>
<td>2</td>
<td>512</td>
<td>319.95</td>
<td></td>
</tr>
<tr>
<td>leader2_4_128</td>
<td>1975050</td>
<td>4</td>
<td>128</td>
<td>78.403</td>
<td></td>
</tr>
<tr>
<td>leader2_4_256</td>
<td>1975050</td>
<td>4</td>
<td>256</td>
<td>131.335</td>
<td></td>
</tr>
<tr>
<td>leader2_4_512</td>
<td>1975050</td>
<td>4</td>
<td>512</td>
<td>267.243</td>
<td></td>
</tr>
<tr>
<td>leader2_8_128</td>
<td>1975050</td>
<td>8</td>
<td>128</td>
<td>61.269</td>
<td></td>
</tr>
<tr>
<td>leader2_8_256</td>
<td>1975050</td>
<td>8</td>
<td>256</td>
<td>121.47</td>
<td></td>
</tr>
<tr>
<td>leader2_8_512</td>
<td>1975050</td>
<td>8</td>
<td>512</td>
<td>213.406</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Grid1

So the key to understanding the test results in grid1 and grid2 is this: [modelName][cluster][# of nodes][Eddy type: MT (for multi-thread) or AHT (for aux hash table)].

Grid1 breaks down as follows: states is simply the number of states that the model generates, queueLines is the number of queue lines in the communication queues, Max_states/Line shows the maximum number
of states in a queue line (and in an MPI message), \( \text{avgSt/msg} \) is the average states across all the MPI state messages, and \( \text{asm}_\text{StDev} \) is the standard deviation of \( \text{avgSt/msg} \).

For grid2, states is the same as above, queue size is the number of bytes in one queue line (also the max size of a state MPI message), \( \text{AvgUtil} \) is the average utilization of MPI state messages (which is calculated by dividing the average number of states/message by the max number of states per queue line), \( \text{avgTime} \) is the start to finish time for the entire cluster, \( \text{at}_\text{StDev} \) is \( \text{avgTime} \)'s standard deviation, \( \text{ram/node} \) is the average ram used per node, and \( \text{arn}_\text{StDev} \) is the standard deviation for the ram/node.

<table>
<thead>
<tr>
<th>OutFile</th>
<th>states</th>
<th>queueSize</th>
<th>AvgUtil</th>
<th>avgTime</th>
<th>( \text{at}_\text{StDev} )</th>
<th>ram/node</th>
<th>( \text{arn}_\text{StDev} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>pet4neb16AHT_A</td>
<td>7968740</td>
<td>1592864</td>
<td>50.449</td>
<td>620.696</td>
<td>26.412</td>
<td>2147.48</td>
<td>0</td>
</tr>
<tr>
<td>lead2neb16AHT_A</td>
<td>1975050</td>
<td>1596448</td>
<td>44.101</td>
<td>154.617</td>
<td>18.45</td>
<td>748.715</td>
<td>21.165</td>
</tr>
<tr>
<td>lead2neb8AHT_A</td>
<td>1975050</td>
<td>1596448</td>
<td>53.665</td>
<td>249.155</td>
<td>23.562</td>
<td>1324.641</td>
<td>25.777</td>
</tr>
<tr>
<td>lead2neb4AHT_A</td>
<td>1975050</td>
<td>1596448</td>
<td>64.822</td>
<td>369.596</td>
<td>1.513</td>
<td>2147.48</td>
<td>0</td>
</tr>
<tr>
<td>lead2neb2AHT_A</td>
<td>1975050</td>
<td>1596448</td>
<td>96.07</td>
<td>613.219</td>
<td>5.244</td>
<td>2147.48</td>
<td>0</td>
</tr>
<tr>
<td>pet4inf32AHT_A</td>
<td>7968740</td>
<td>1592864</td>
<td>27.476</td>
<td>578.737</td>
<td>102.262</td>
<td>1395.454</td>
<td>17.772</td>
</tr>
<tr>
<td>lead2neb16MT_A</td>
<td>1975050</td>
<td>1596448</td>
<td>55.094</td>
<td>142.132</td>
<td>2.096</td>
<td>631.158</td>
<td>4.926</td>
</tr>
<tr>
<td>lead2neb8MT_A</td>
<td>1975050</td>
<td>1596448</td>
<td>62.941</td>
<td>243.423</td>
<td>1.063</td>
<td>1169.799</td>
<td>5.536</td>
</tr>
<tr>
<td>lead2neb4MT_A</td>
<td>1975050</td>
<td>1596448</td>
<td>77.774</td>
<td>403.894</td>
<td>2.35</td>
<td>2147.48</td>
<td>0</td>
</tr>
<tr>
<td>lead2neb2MT_A</td>
<td>1975050</td>
<td>1596448</td>
<td>94.508</td>
<td>599.801</td>
<td>2.137</td>
<td>2147.48</td>
<td>0</td>
</tr>
<tr>
<td>pet4inf32MT_A</td>
<td>7968740</td>
<td>1592864</td>
<td>45.945</td>
<td>392.81</td>
<td>22.411</td>
<td>1336.964</td>
<td>10.787</td>
</tr>
<tr>
<td>lead6inf64MT_A</td>
<td>80684137</td>
<td>1597472</td>
<td>59.614</td>
<td>3971.21</td>
<td>2147.48</td>
<td>1139.849</td>
<td>1139.921</td>
</tr>
<tr>
<td>lead2.1_128</td>
<td>1975050</td>
<td>798736</td>
<td>103.176</td>
<td>230.414</td>
<td>233.74</td>
<td>1143.673</td>
<td>1143.673</td>
</tr>
<tr>
<td>lead2.1_256</td>
<td>1975050</td>
<td>1592864</td>
<td>98.728</td>
<td>229.371</td>
<td>1139.849</td>
<td>1139.849</td>
<td>1139.849</td>
</tr>
<tr>
<td>lead2.1_512</td>
<td>1975050</td>
<td>2396208</td>
<td>96.294</td>
<td>233.74</td>
<td>1143.673</td>
<td>1143.673</td>
<td>1143.673</td>
</tr>
<tr>
<td>lead2.2_128</td>
<td>1975050</td>
<td>798736</td>
<td>67.089</td>
<td>235.191</td>
<td>1139.849</td>
<td>1139.849</td>
<td>1139.849</td>
</tr>
<tr>
<td>lead2.2_256</td>
<td>1975050</td>
<td>1592864</td>
<td>64.778</td>
<td>239.592</td>
<td>1159.921</td>
<td>1159.921</td>
<td>1159.921</td>
</tr>
<tr>
<td>lead2.2_512</td>
<td>1975050</td>
<td>2396208</td>
<td>62.49</td>
<td>243.832</td>
<td>1166.911</td>
<td>1166.911</td>
<td>1166.911</td>
</tr>
<tr>
<td>lead2.4_128</td>
<td>1975050</td>
<td>798736</td>
<td>61.252</td>
<td>321.702</td>
<td>1213.388</td>
<td>1213.388</td>
<td>1213.388</td>
</tr>
<tr>
<td>lead2.4_256</td>
<td>1975050</td>
<td>1592864</td>
<td>51.303</td>
<td>242.851</td>
<td>1331.271</td>
<td>1331.271</td>
<td>1331.271</td>
</tr>
<tr>
<td>lead2.4_512</td>
<td>1975050</td>
<td>2396208</td>
<td>52.196</td>
<td>268.946</td>
<td>1317.178</td>
<td>1317.178</td>
<td>1317.178</td>
</tr>
<tr>
<td>lead2.8_128</td>
<td>1975050</td>
<td>798736</td>
<td>47.866</td>
<td>315.224</td>
<td>1507.038</td>
<td>1507.038</td>
<td>1507.038</td>
</tr>
<tr>
<td>lead2.8_256</td>
<td>1975050</td>
<td>1592864</td>
<td>47.449</td>
<td>312.951</td>
<td>1506.863</td>
<td>1506.863</td>
<td>1506.863</td>
</tr>
<tr>
<td>lead2.8_512</td>
<td>1975050</td>
<td>2396208</td>
<td>41.681</td>
<td>283.217</td>
<td>1426.916</td>
<td>1426.916</td>
<td>1426.916</td>
</tr>
</tbody>
</table>

**Table 3. Grid2**

It appears for the disparity between EddyMurϕ ability to scale compared to EddySPIN’s is due to the low computational overhead in generating states in SPIN. As we increase the number of nodes, the demands
of communication quickly outweigh the processing power advantage that would be gained by parallelization. Also, we find that there is a problem sometimes with the following scenario: an idle node will receive a state that expands (it generates successors) to a few states that do not belong to this node. Then we can have a situation where many small messages are traveling through the network and the majority of the work is being accomplished is communications.

4 Usage

There are two current branches of Eddy\textsubscript{SPIN}; the disk caching version and a multi-threaded hash table state searching version (meaning that the multi-threaded one checks the main hash table as successor states are generated locally and received from external nodes on separate threads). Both versions are set up in separate folders below the root that you install Eddy\textsubscript{SPIN} to (/\texttt{mutex} and /\texttt{diskCache}), and you can think of each as its own root, as they are currently independent of one another. There are two parameters that you can adjust to best suit the cluster you are going to run Eddy\textsubscript{SPIN} on: \texttt{BUFFSIZE} and \texttt{BUFFCOUNT} control the number of states per buffer line (and thereby the max number of states in an MPI message carrying states) and the number of buffer lines, respectively. We have found that the most effective parameters take advantage of the maximum packet size of the topology used for the cluster. The size of a state vector will vary by the model; this can be determined by examining the struct \texttt{st} and struct \texttt{State} located in the \texttt{pan.h} that SPIN generates in the directory /\texttt{spin/} from the root that it was unzipped (the state vector size is the sum of both of these structures sizes) for the Promela model (or you may set a breakpoint at the line that calls \texttt{create_comm_manager()} in the function \texttt{parallelInit} in \texttt{spin_extension.c} and type `'print sizeof(struct st) + sizeof(struct State)' from a debugger such as GDB). The maximum number of states in a line should then be max packet size / state vector size. As far as the number of queue lines that are most effective, it probably depends on how efficiently the particular model you are verifying works with the Eddy\textsubscript{SPIN} framework. Often a value of only 1 or 2 works well (but you might want to experiment with a few varied settings of \texttt{BUFFCOUNT}). These values are in \#defines in either root (/\texttt{mutex} and /\texttt{diskCache}) within the file /\texttt{spin/spin_extension.c} as \#define \texttt{BUFFSIZE} and \#define \texttt{BUFFCOUNT}.

Another group of parameters that can be adjusted for the disk cache version is located in the file 'eddy\textsubscript{spin}.conf' within the ./diskCache root. Here is a sample of that file:

```
# the size of one disk page
PAGE_SIZE=1048576

# the maximal number of disk pages in one file
```
NUM_PAGES_IN_ONE_FILE = 800

# the total number of cache pages
NUM_DISKCACHE_PAGES = 19000

# maximal allowed memory on each node
MAX_MEM_USAGE = 2048.0

# maximal allowed time
MAX_EXEC_TIME = 56000.7

These values may be adjusted to the users environment.
Once you have set all the parameters to your choosing, it is a simple matter to run ‘make’ with the makefile in each root directory, depending on the version you want to run.
Eddy_SPIN has a complete makefile that allows you to build its SPMD MPI model by typing ‘make’ in the proper root directory followed by the name of the Promela model that you would like to build (located in the ./models folder). Like: ‘make petersonN.pml’. Executing this command will build the source files under ./src and link them with the appropriate SPIN files in the ./spin directory. Also, the model that you specify on this line will be built with SPIN -a [modelname], and the verifier that SPIN generates will also be compiled with Eddy.
Once you have the executable, Eddy, you may run it in any way that your MPI environment dictates. We have used both LAM and MPICH.
To run Eddy without a PBS job control system in LAM, first perform a ‘lamboot’ on the command line (refer to the man pages by typing ‘man lam’). Then you may run Eddy by typing ‘mpirun -np [# of nodes] eddy [output file].
If you’d like to utilize PBS, then open the supplied lamjob.sh using a suitable editor such as Emacs. The script is loaded with comments to explain the settings; in particular you should set the variables LAMJOB (with an absolute path name to the Eddy executable), nodes, walltime, and the line #PBS -q [queue name]. To determine your queue name, you may run ‘qstat -q’, and PBS will return the statistics on available queues, including names and resources available. For more details on PBS, see the man pages via ‘man PBS’.

5 Further Work

At this point, it seems that Eddy_SPIN is suffering from two problems: the first is that the communication could be more efficient because of the way that states are partitioned; the second is the inefficiency in the way that we have extended SPIN.
We have several ideas to address the first problem. The problem that we have now is, with our random state partitioning (the algorithm that determines a state’s owner is simply getHash(state) mod number_of_nodes)
is that it doesn’t take into consideration any kind of successor prediction or work load balancing. One idea to address the problem of state successor chain discontinuity is to find a way to give states an affinity to a particular node. It may work to partition states by their PID (process identifier), as a high percentage of successor states generated by one PID will have the same PID. Another approach may be to monitor work load across the network and have some kind of load balancing algorithm. Perhaps a regular broadcast made by each node as to its workload could keep load statistics for the entire network at each node, and could be used in the distribute function to determine where states could be most effectively expanded. There is a better way to keep the worker threads busy across the MPI network.

The second problem is a little bit harder. How do we more efficiently tie our expansions into SPIN? Robert Palmer has had this conversation with SPIN creator Gerard Holzman, and he suggests the only way to effectively extend SPIN would be to rewrite it.

Orthogonally, there is work to do to understand the precise relationships between network topology, machine speed, available memory, state vector size and the tuning parameters available with Eddy-SPIN. It would be interesting to identify specifically the factors that make for most effective communications thereby allowing models to be verified in the fastest time.

References