Analysis of Distributed Spin Applied to Industrial-Scale Models*

Murali Rangarajan, Samar Dajani-Brown, Kirk Schloegel, Darren Cofer

Honeywell Laboratories
3660, Technology Drive, Minneapolis, MN 55418, USA

{murali.rangarajan, samar.dajani-brown, kirk.schloegel, darren.cofer}@honeywell.com

Abstract. As software systems become increasingly complex, there is growing interest in the use of formal techniques to obtain higher assurance in their correctness. The most commonly used tools involve model-checking, such as SMV and Spin. But modeling complex systems with a high degree of fidelity implies exceedingly large state spaces that must be analyzed. These state spaces are typically too large for single processing nodes, in spite of great advances in memory reduction techniques. Moreover, approximation techniques such as hash compaction are less well-received where safety-critical systems are concerned. Effective distribution of the problem over many processing nodes has the potential of supporting the huge state spaces. Since our primary interest is in safety-critical software, we have spent considerable time evaluating the performance of distributed implementations of Spin in this context. In this paper, we present our analysis of PSPIN, a distributed implementation of Spin. We identify key measures of effectiveness, and evaluate PSPIN with respect to these measures. We also present an alternative approach to partitioning that performs comparably with respect to all measures, and is up to orders of magnitude faster. Finally, we consider the question of which measures have the greatest impact on peak memory, a measure that is most critical to effective distribution.

1. Introduction

Software applications used to assist the flight of aircraft have long been considered safety-critical, and as such have been required to be developed using strict processes to maintain the high standards of the end products. These processes rely on reviews, simulations and testing to achieve those high standards. A number of factors have resulted in Formal Methods tools being seriously considered as part of this mix. The first of these reasons is the increasing functionality being demanded from the software, resulting in a steady increase in complexity. The second is the pressure to bring the product to market as quickly as possible. Third is the growing realization

* This material is based upon work supported in part by NASA under cooperative agreement NCC-1-399.

that existing processes are inadequate. It is not possible to test some requirements explicitly (requirements such as all threads are guaranteed their budgets by the operating system). It is also not possible to identify and test all corner cases in complex systems – especially in systems that involve parallelism or distribution. Finally, there is growing data that early and effective use of formalisms in the development process reduces the chances of major rework late in the development process.

Among the formal analysis techniques available, we have found model checking, and in particular, Spin[7], to be the most amenable to integration with safety-critical software development processes. As with other model-checking tools, Spin's verifier can automatically analyze models against requirements. In addition, we have found Spin's modeling language, PROMELA, to be easiest to translate to from the C++ code being used in our studies. Finally, Spin has been used extensively and is highly trusted, and it can check for a wide range of properties.

We have successfully used Spin to analyze various embedded software in the past[4]. Our current work with the Deos™ real-time operating system[8] has pushed Spin to its limits. In particular, as features were added to the model, the memory requirements for full verification far exceeded the 4GB memory limit imposed by the 32-bit Spin implementation. This was in spite of using Spin's memory reduction techniques such as state-vector compression and partial order reduction, and also predicate abstractions.

One solution to deal with the memory problems is to look at distributing the problem over many processing nodes. In theory, parallelization should cut the computation time as well as the amount of memory per node. However, it is believed [12] that due to fine grained communication requirements, distributed model-checking is inherently unscalable. Therefore, parallelization actually results in significant performance slow down. Thus, distribution of Spin is most effective when the memory overhead of distribution is low, and the time overhead is not prohibitively large.

In this paper, we evaluate the performance of a distributed implementation of Spin, namely PSPIN[12]. This is particularly interesting as it is implemented as a wrapper around the traditional Spin, thereby reducing the probability of errors in the verification. In particular, we analyze the memory and time performance of this implementation. We identify measures of effectiveness for this category of distributed verifiers. We present an alternative scheme that partitions the state space according to the value of automatically selected elements of the state vector. We provide results that show that the new scheme performs comparably with respect to all these measures while over 25 times better with respect to verification time. Finally, we consider the issue of memory overhead for distribution and identify potential optimization measures that have the most effect on this.

This paper is organized as follows. In the next section, we provide an overview of the Deos real-time operating system, and discuss briefly our approach to modeling it. Next, we analyze the performance of PSPIN, and identify the key measures by which we can judge the effectiveness of distribution. Then, we present our approach to partitioning and compare it with PSPIN's partitioning schemes. We conclude the paper with related work and directions for further study.

2. The Deos RTOS and its modeling

The Deos real-time operating system was developed by Honeywell for use in the Primus Epic avionics suite for business, regional, and commuter jet aircraft. Deos hosts many safety-critical applications in these aircraft, including primary flight controls, autopilots, and displays.

Deos is a microkernel-based real-time operating system that supports flexible Integrated Modular Avionics applications by providing both space partitioning at the process level and time partitioning at the thread level. Space partitioning ensures that no process can modify the memory of another process without authorization, while time partitioning ensures that a thread's access to its CPU time budget cannot be impaired by the actions of any other thread. Deos supports many advanced features such as dynamic creation and deletion of processes and threads, reuse of unused thread budgets (also known as *slack time*), aperiodic interrupts, synchronization mechanisms such as mutexes, etc.

Deos is an interesting problem for this study for many reasons. First, analysis of source code is an unobtrusive way for formal analysis tools to gain acceptance within existing development processes. Second, Deos has a number of interesting properties that are well-suited for formal analysis. These include: (1) Time partitioning, which is a property that is practically impossible to test; and (2) Function pre-conditions, inserted in the code as comments that need to be checked at every invocation of the function. Finally, the Deos model has an atypical asymmetric model with one very large process and many smaller processes, which is a more challenging distribution problem.

One requirement for our approach was the ability to automatically generate verification models, since it was expected that engineers with minimal training in formal analysis techniques would be the users. Therefore, though our Deos model was constructed by hand, automated translation was a key consideration in creating the model. This resulted in a model with a direct mapping (almost line-to-line correspondence) to the source code, which had the added benefit of being easier to review by the developers.

The Deos model consists of three parts – the kernel, user threads and the environment. The kernel corresponds to the code translated from C++. It provides services to the user threads and interfaces with the hardware environment. It is the most complex part of the model. The user threads are very simple and just invoke various calls to the kernel and responds to messages (such as preemption) from the kernel. The environment provides the hardware services such as timers and interrupts. Though the environment is not as complex as the kernel, it is more difficult to model realistic hardware behavior in pure software. In our tests, we have found an event-based simulation-like approach to be the most suitable for modeling the environment. The Deos model initially consisted of the basic rate-monotonic scheduling algorithm with support for multiple threads and periods. Various features were incrementally added to the base model, including slack time, asynchronous interrupts, mutexes and overhead accounting.

The model incorporates a number of memory optimizations. These optimizations fall into two categories – those that reduce the size of the state vector, and those that reduce the number of distinct states seen by the verifier. State vector size is reduced

by replacing all integers with bytes, reusing temporary variables, and by judicious use of hidden variables. The size of the state space is reduced by the use of predicate abstractions where possible, and judicious use of configurations (number of threads, their budgets, whether the threads are slack-enabled, etc.).

Our initial models could be exhaustively verified by Spin in 335 MB of memory. But by the time slack and interrupt threads were added, the state space became too large to be exhaustively verified in 4GB of memory. Though we continued to use Spin's approximation techniques such as bit-state hashing as debugging aids, these techniques are a harder sell for incorporation in the development processes of safety-critical systems. This led to our interest in other tools that can handle these large state spaces, and in particular, PSPIN.

3. PSPIN overview

PSPIN is a wrapper around Spin that distributes the memory used for verification over many processing nodes (typically workstations in a network). The overall state space is partitioned into as many state sub-domains as the number of network nodes. Each node is assigned a different state sub-domain, and holds only the states that belong to that subset of the state space. During the verification run, each node computes the successors of the states it holds, and if it finds any successors belonging to other state sub-domains, it sends them to the nodes that are in charge of processing them. Since this is implemented as a wrapper around Spin, it is compatible with some of Spin's memory reduction techniques such as state compression. Its compatibility with partial order reduction is discussed later in this paper.

There are a number of challenges associated with partitioning vast and unknown state spaces:

1. It is not efficient to construct a structural model of the entire state space. Therefore, all schemes must utilize a predictive approach. The use of such an approach can impact guarantees of optimality that may characterize particular partitioning schemes. (Although it is still possible to provide guarantees of optimality for a sample of the state space.) Furthermore, it is often not possible to guarantee communications and load balance bounds for predictive schemes.
2. Since the partitioning function is called frequently (every time a child state is examined), processing nodes should be able to compute the owner node of a particular state quickly using purely local information. At the very least, the partitioning scheme must require non-local information infrequently.
3. As the state space is larger than the available memory of a single processing node, it is not feasible for each processing node to maintain an array that maps every state to its assigned node. (We refer to such an array as a *partition array*.) Instead, it is necessary to encode and decode the partitioning using some technique whose memory requirement is much less than the size of the state space and that is relatively fast. It is typically the case that not every possible partition array is representable by a particular encoding technique. Therefore, the reachable partitioning space is effectively reduced by such methods. This effect impacts

optimality, as optimal partitionings may not reside within the reachable solution space.

PSPIN implements three partitioning functions: *Global Hash (GHP)*, *Local Hash (LHP)*, and a graph partitioning-based method that we refer to as *Source Code Partitioning (SCP)*. These address the above challenges in various ways. The Global Hash partitioning function maps states to processing nodes using a computation that involves the whole state vector. For example, the computation could be as simple as adding the values of all the variables in the state vector and computing its modulo with the number of processing nodes. Figure 1 illustrates this scheme as well as the Local Hash scheme. Here, a state vector v of size m is divided into n sub-vectors, one for each process in the model. Figure 1 shows the resulting processing node computation for this state vector using the Global Hash scheme. (Note that the computation is incomplete, as some of the state vector is not shown.) Global Hash requires only local information (i.e., the state vector). It can be shown to balance the load with high probability given a few reasonable assumptions about the distribution of the values of the state vector. However, this partitioning approach displays little or no locality with respect to how the states will be partitioned. That is, the probability that any two states that are adjacent in the state space will be mapped to the same processing node approaches zero as the number of nodes approaches infinity.

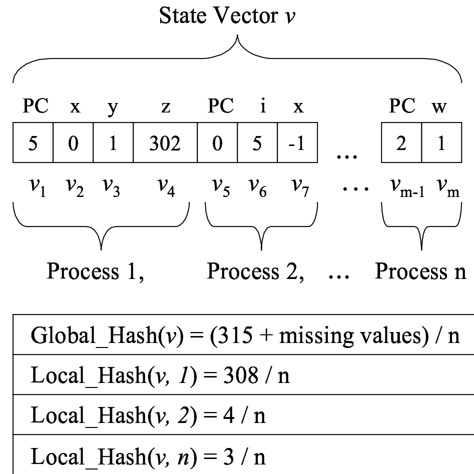

Fig. 1. Illustration of Global Hash and Local Hash partitioning schemes

Local Hash performs a similar computation to Global Hash, but utilizes only the values of a portion of the state vector belonging to a single Spin process, p. Figure 1 shows the resulting processing node computations for the state vector v when p is set to 1, 2, and n. Local Hash requires only local information. It increases locality compared to Global Hash, as any two adjacent states are guaranteed to be mapped to the same processing node if PSPIN is not currently executing the selected process p. So given a model consisting of processes of roughly equal sizes, the probability that any two

adjacent states will be mapped to the same processing node approaches one as the number processes approaches infinity. However, as the portion of the state vector that is associated with process *p* is small compared to the entire state vector, the modulus scheme is less likely to result in good load balance. Furthermore, as the number of processes increases, the Local Hash scheme will tend to obtain worse load balancing results than the Global Hash scheme.

The third partitioning scheme available in PSPIN, Source Code Partitioning, is based on graph partitioning. In this scheme, the graph representing the source code for a selected process *p* is partitioned (and not a graph modeling the state space or a sample of the state space). Figures 2 and 3 illustrate this approach. Figure 2 shows a Promela model consisting of two processes, *init* and *P1*.

```
int sum;

       proctype P1()
       {
         int i = 0;
         int evensum = 0;
         do
 1:        :: sum < 7 ->
             do
 2:            :: i< 10 ->
 3:               if
 4:                 :: (i % 2 == 0) -> evensum =evensum + i;
                      i++;
 5:                 :: i % 2 !=0   -> i++;
                  fi
 6:            :: else ->
                 break;
             od;
 7:          sum++;
 8:        :: else ->
           break;
         od;
 9:      printf("sum=%d\tsum= %d\n", sum, evensum);
10:    }

       init {
         sum = 0;
         run P1();
       }
```

Fig. 2. Sample code to illustrate Source Code Partitioning

Assume that the selected process is P1. (Note that using the init process would result in a partitioning with extremely poor load balance.) Figure 3(a) illustrates the source code graph for P1. This graph is constructed by PSPIN and models the possible control flows for the model. The circled numbers next to each vertex refer to the relevant line of source code from Figure 2. PSPIN can optionally weight the vertices

and edges of this graph. It does so by performing a prerun of Spin. Each time a forward transition is taken, the weight of the associated edge on the graph is incremented. Similarly for vertices, each time a line of source code is executed by Spin, the weight of the corresponding vertex is incremented. Figure 3(a) includes edge weights, but for the sake of clarity, vertex weights are not shown. A simple method for computing the vertex weights is to add up the weights of all outgoing or incoming arcs.

Figure 3(b) shows the abstract graph that is passed to the Metis Graph Partitioning Package [10]. Metis partitions this graph into a number of sub-domains equal to the number of processing nodes. (In this example, we have three processing nodes.) Metis returns a partition array similar to the one shown in Figure 3(b). This partition array is compiled into a lookup table for the PSPIN partition function. The input to this partition function is the value of the program counter (PC) element from the state vector for the selected process (P1). In Figure 3(b), when the value of the PC is 6, 7, 9, or 10, the state maps to node 0. When the value is 1, 2, or 8, the state maps to node 1. PC values of 3, 4, and 5 map to node 2.

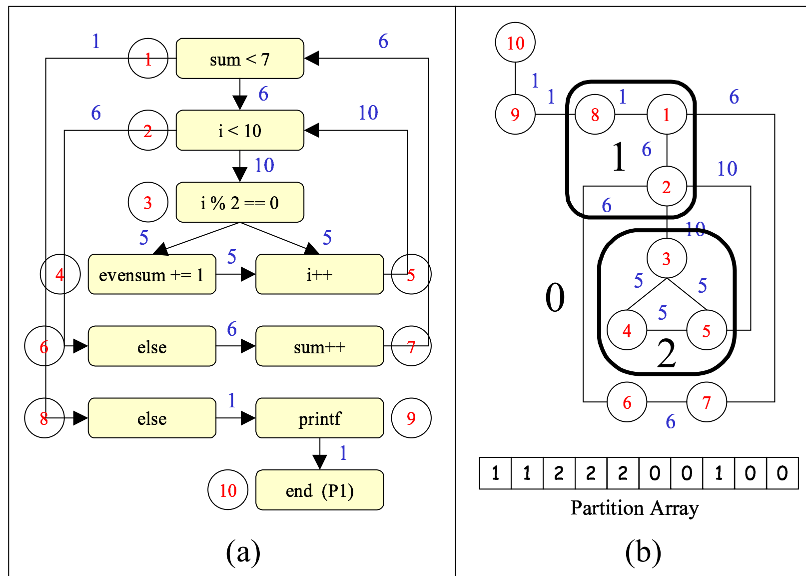

Fig. 3. (a) Source code graph for p1, and (b) the abstract graph that is passed to Metis

The key idea behind Source Code partitioning is that the structure of the state space can be predicted given the flow control structure of one of the processes. By partitioning the source code graph to minimize the total weight of the control flow edges that are cut by the partitioning (as Metis does), the resulting partitioning should demonstrate good locality. Furthermore, by applying vertex weights that correspond to the relative frequency that source code lines are executed, the state space can be load balanced. That is, source code lines that are likely to be executed frequently (for instance, the code within a nested loop), will be likely to result in proportionally more states in the state space than less frequently executed code. So taking this information

into account when partitioning the source-code graph can result in good load balance for PSPIN. Finally, the mapping of a state to its owner node can be computed using purely local information if the Metis partition array is duplicated on every node. In this case, the value of the appropriate PC in the state vector is used as the index to this local copy of the partition array.

3.1 PSPIN evaluation with small Deos model

The results from our initial evaluation of PSPIN's three partitioning schemes on a model of Deos over 2 and 4 processors is shown in Table 1. Note that GHP refers to Global Hash partitioning, LHP refers to Local Hash partitioning and SCP refers to Source Code partitioning schemes. Also note that the memory values are those reported by Spin at the end of the verification run, with the values in bold face being the aggregate sum from all the individual processing nodes. The time is the user time reported by the `time` command. Finally, all these results were obtained with partial order reduction disabled. The following points are evident from the results:

1. GHP results in even distribution of state space (memory) over all the processing nodes. But the verification time (run time) for GHP is much higher than the other two partitioning schemes, and also serial Spin. Moreover, the rate of increase in verification time with the increase in number of processing nodes is far too great for this approach to be practical. This is expected, as there is no locality in the partitioning scheme.
2. LHP on two nodes resulted in an approximately even distribution of states (memory) among the two nodes. The amount of memory used per node was less than half the memory used by serial Spin, and the run time was approximately the same as serial Spin. However, with 4 nodes, the results were more like running the test on two nodes, with one node not processing any states, and another node seeing very few states. Most of the work was done by two nodes.
3. SCP over two nodes provided results that were comparable to those using local hash partitioning. But, over four nodes, two of the nodes did not perform any work.

It is to be noted that the above results were obtained with partial order reduction (POR) turned off. This is significant, as we found in our tests that turning POR on results in inconsistent performance of distributed verification. With POR turned on, the sum total of states seen by all the nodes is typically larger than that seen by serial Spin. Traditional wisdom [12] has been that this is due to the fact that POR is less effective when distributed over many nodes. But when we compared the states seen by the distributed PSPIN with those seen by serial Spin, we found that while the distributed version saw some additional states, it also missed some states seen by serial Spin. This is a big problem, as it now becomes more difficult to validate the correctness of the distributed implementation. We are studying this issue further. For the purposes of this paper, all the results henceforth are with POR turned off.

It is clear that both local hash and source code partitioning have better time and memory performance as compared to global hash partitioning. But both techniques were not making effective uses of all available nodes. Repeated tests with increasing prerun times for source code partitioning did not result in better use of all available

nodes. Further study indicated that the problem was with the choice of Spin process used by the partitioning algorithm, which must be specified by the user.

	Partition Function	Memory	Messages Sent	States Stored	Computation Time
Serial	**none**	**24.077**		**477722**	**17 seconds**
2 Nodes	**SCP**	**20.703**	**17824**	**476577**	**13 seconds**
Node 1		10.607	13468	201431	
Node 2		10.095	4356	275146	
4 Nodes	**SCP**	**25.738**	**17824**	**476577**	**10 seconds**
Node 1		10.607	13468	201431	
Node 2		2.518	0	0	
Node 3		2.518	0	0	
Node 4		10.095	4356	275146	
2 Nodes	**LHP**	**20.703**	**17824**	**476577**	**14 seconds**
Node 1		10.607	13468	201431	
Node 2		10.095	4356	275146	
4 Nodes	**LHP**	**26.659**	**88368**	**476577**	**12 seconds**
Node 1		10.607	13468	201431	
Node 2		7.638	4356	169646	
Node 3		2.518	0	0	
Node 4		5.897	70544	105500	
2 Nodes	**GHP**	**48.864**	**3125460**	**1467660**	**251 seconds**
Node 1		24.432	1562940	733953	
Node 2		24.432	1562520	733712	
4 Nodes	**GHP**	**66.495**	**5139780**	**1665980**	**2835 seconds**
Node 1		16.649	1283630	416332	
Node 2		16.547	1283730	416197	
Node 3		16.649	1287480	417253	
Node 4		16.649	1284940	416202	

Table 1. Comparitive performance of PSPIN's partitioing schemes

Our Promela model consists of one large process (corresponding to the Deos kernel) and a number of smaller processes. However, the default value for selecting the process whose source code graph is partitioned (variable `NPROC` in files `table.t and ppan.c`) corresponded to one of the smaller processes. We set the process selection variable (`NPROC` in SCP, HASHPROC in LHP) to correspond to the large Deos process (the Deos kernel) and the results were much better (as reflected in results shown later in this paper), with all the processors being well utilized.

3.2 PSPIN evaluation with large Deos model

We next analyzed larger models, albeit models that could be verified using serial Spin within 4GB of memory, using PSPIN. We found that some of these runs did not

complete, and had to be killed, for no apparent reason. Detailed analysis of reasons for PSPIN's behavior pointed to issues of memory usage. PSPIN code was then modified to incorporate counters that kept track of memory allocation and deallocation within the Spin and PSPIN parts of the code. In addition, the new code also kept track of the corresponding maximum values (*peak memory*) reached by the memory counters for both Spin and PSPIN during the verification runs. Experiments with this updated implementation indicated that actual memory usage, at some point in the verification run, far exceeded the amounts being reported by Spin, thus explaining why PSPIN failed on some large Deos models.

The results from one such experiment are presented in Table 2. Verification using serial Spin of the model used in this experiment consumed 1100.946 MB of memory. This experiment was performed on a four processor (P0, P1, P2, and P3) shared memory LINUX computer with 16GB of total physical memory. The total amount of memory used by all four processors to distribute the memory requirements of SPIN was 1414 Mbytes. The amounts of memory consumed by each processor as reported by PSPIN are listed in the column labeled *Total Memory Reported*, rows one through four. Note that the values in this column match the values of peak memory consumed by a processor for the SPIN verification (column *Peak Memory SPIN*). So we can conclude that the reported memory usage only consists of SPIN-allocated memory (and not PSPIN-allocated memory). The column labeled *Peak Memory PSPIN* is the peak memory consumed by a processor as a result of overhead from PSPIN memory allocations. It does not include SPIN memory.

Processors	Total Memory Reported (MB)	Peak Memory Spin (MB)	Peak Memory PSPIN (MB)	Script Results from Top
P0	302.19	302.19	50.44	~ 328 MB
P1	322.37	322.37	1,883.87	~ 3 GB
P2	407.15	407.15	115.48	~ 464 MB
P3	382.99	382.99	49.28	~ 400 MB

Table 2. PSPIN on four processors, with detailed tracking of memory usage.

Table 2 shows that one processor (P1) requires approximately 30% more memory than serial SPIN. To confirm this observation, we wrote a simple UNIX shell scripts that recorded results from the UNIX system command "top" every 5 seconds while the above experiment was conducted. The results from the shell script, described in column 5, shows that processor one (P1) requires the largest amount of memory by far. The unreported PSPIN memory usage is due to communication buffers and explains why at times PSPIN does not complete on large DEOS models.

The notion of peak memory is a critical measure of effectiveness of any distributed verification scheme. If the actual memory consumed during a verification run is more than the available physical memory, it does not matter if the final consumption would be less than the physical memory, as verification would definitely fail. Our analyses show that the predominant component of this memory used by PSPIN is communication overhead.

4. State space partitioning

The Source Code Partitioning scheme tries to elicit the structure of the state space given the structure of the flow control graph as well as by weighting this graph by traversing a sample of the state space. A related approach is to elicit the structure of the state space by examining a sample of this space and constructing a graph, not of the flow control, but of the state space directly. That is, during a prerun, a graph can be constructed that represents states as vertices and the transitions between states as the edges of a graph. Then this graph can be partitioned directly by an off-the-shelf graph-partitioning package.

4.1 Weighting vertices

Weights can be applied to vertices by a number of methods. Typically a single weight is assigned to each vertex. However, a single weight does not allow you to model both the memory and computation associated with a single state. You can only model one or the other. It is possible to model both memory and computation if each vertex is given two weights. That is, each vertex of the graph is assigned a weight vector of size two. The first element of this vector represents the memory requirement of the state and the second represents the work requirement of the state. Every vertex that corresponds to an examined state is given a weight vector of (1, 0) that indicates the state requires one unit of memory to store, but no further work is associated with this state. Every vertex that corresponds to an open state is given a weight vector of (1, 1) that indicates it has both memory and work requirements. A multi-constraint graph partitioner [11] can be used to partition such a graph. (Note that the Metis package also implements a number of multi-constraint graph partitioning algorithms.)

4.2 Generalized state vector element partitioning

The Source Code Partitioning scheme essentially partitions the state space based upon the PC for a selected process p. We can automate this approach to some extent by computing a partitioning for each Spin process, and then automatically selecting the best partitioning to use. Hence, the user does not need to select a process manually. Indeed, in our experiments, we have found that it is often the case that certain processes can result in extremely bad partitionings.

We can further extend and generalize this scheme by allowing the state space to be partitioned based upon any element of the state vector -- and not just the various PCs. A simple algorithm is to compute a partitioning for every element of the state vector and to select the state vector element and partition array of the best partitioning seen. We have implemented an algorithm that does so and refer to it as State Vector Element Partitioning (SVEP).

The SVEP algorithm requires a prerun, during which the graph that models the state space is constructed. At this time, it is necessary to record not only the connectivity among the states in the state space, but also the state vector for each state

visited. The prerun terminates after a pre-selected depth is reached. After the augmented state-space graph is constructed, SVEP applies the following algorithm in a recursive bisection manner [2] to result in a *k*-way partitioning.

For each element of the state vector:

1. The range of values *r* that has been seen for the element is determined.
2. If the *r* is less than or equal to a specified threshold *t*, then the graph is collapsed into *r* vertices. If *r* is greater than *t*, then the values are grouped together into *t* evenly sized sets, and the graph is collapsed into *t* vertices.
3. The graph is collapsed by creating a single *super vertex* for each distinct value (or set of values in the case in which *r* is greater than *t*) seen. The weight of each super vertex is equal to the sum of the weights of its component vertices.
4. For each edge *e* in the initial (i.e., uncollapsed) graph that connects two vertices that are mapped to different super vertices, either (i) a *super edge* is added to the collapsed graph, or (ii) if a super edge already exists between the two super vertices, then the weight of *e* is added to the weight of the super edge.
5. All edges of the initial graph that connect vertices that are mapped to the same super vertex are ignored.
6. The resulting collapsed graph is likely to be small. If it contains less than 15 vertices, a two-way partitioning is computed optimally by brute force. (That is, every possible partitioning is examined, and the best one is returned.) If the collapsed graph is larger than 15 vertices, Metis is called to partition it in half.

The best partitioning that is found by this process is returned along with identification information (e.g., the index of the associated element and its range information). The returned information is used to generate a partition function.

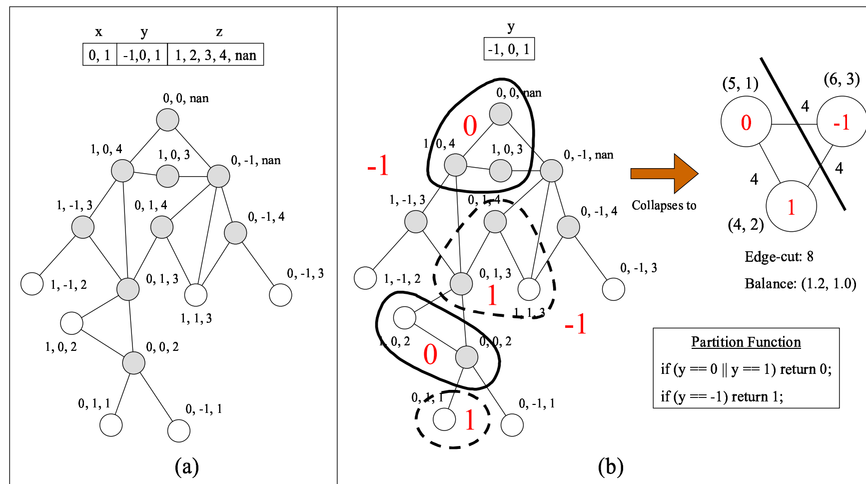

Fig. 4. (a) An example of an augmented state-space graph, and (b) the resulting collapsed graph with respect to the value of variable *y*

Figures 4 and 5 illustrate this process. Figure 4(a) shows an augmented state space graph. In this example, the state vector consists of three elements that correspond to

Promela variables x, y, and z. (Note, that for purposes of this simple example we do not include process PCs in the state vector. However, any or all of these three variables could be PCs and the results would be the same.) Vertices are shaded if their corresponding states have been examined. White vertices correspond to open states. The range of seen values for each element is shown at the top of the figure. For example, variable z has taken the values 1, 2, 3, 4 and nan (i.e., undefined) in the sampled state space. The graph shows each state encountered along with its unique state vector and the connectivity among the states. For example, state (1, 0, 4) can reach or be reached by states (0, 0, nan), (1, 0, 3), and (1, -1, 3).

Figure 4(b) shows how the augmented graph is collapsed with respect to variable y. (Note that since only two values are seen for variable x, the resulting graph has only two vertices and the partitioning can be performed trivially.) In this case, the range of three values that are seen for y result in a collapsed graph with three vertices. The optimal partitioning of this graph is shown. This partitioning has an edge-cut of nine and state balances of 1.2 and 1.0. (Balances are computed by max sub-domain weight over average weight.) The partition function that is derived from this partition array is shown. In this case, the partitioning results in a better edge-cut and the same balance compared to the partitioning that was trivially computed for variable x. Hence, this partitioning is saved as the *current best*.

Figure 5 shows the graph collapsed with respect to variable z. A range of five values results in a collapsed graph with five nodes. The optimal partitioning is shown for the collapsed graph. This partitioning has an edge-cut of eight (same as the current best) and balances of 1.07 and 1.3. Depending on which vertex weight is favored, either of these partitionings could be selected as the current best. We give more weight to the balance that corresponds to the open vertices, as this approach will favor balancing the predicted work among the processing nodes to balancing the memory load. Therefore, the current best partitioning is that of variable y.

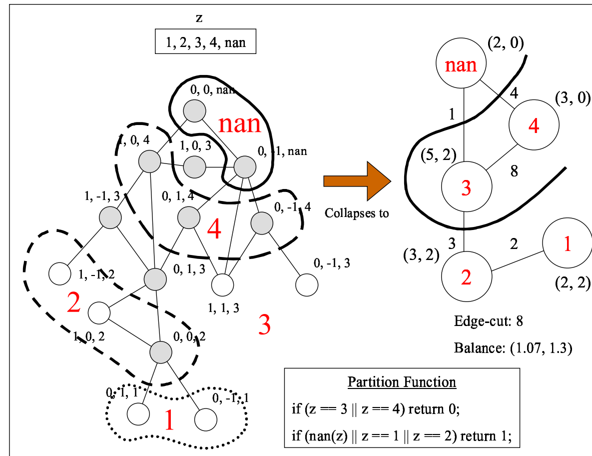

Fig. 5. The augmented state-space graph collapsed with respect to the value of variable z

This algorithm is applied in a general recursive bisection manner [2]. That is, for a k-way partitioning, after the initial bisector of the state space is determined, the

augmented state-space graph is split into two graphs using this bisector. Then the algorithm is applied recursively to both sub-graphs. The automatically generated partition function then is structured as a set of nested if-then-else statements. This scheme can handle values of k that are not powers of two by splitting the graph, not exactly in half by weight, but into one-third and two-third weight regions. The one-third side is not split further, while the two-third side is split further.

Sometimes a variable might only take a single value throughout the entire sample space. In this case, it is not possible to compute a partition using this variable. Also, it is possible that more values might be eventually encountered in the full state space than are seen during the prerun. Therefore, the partition function must be generated to be complete. In our implementation, we cover these possible values by a simple scheme. (That is, all values less than those seen are mapped to node zero. All values higher than those seen are mapped to node one.)

4.3 Evaluation of SVEP

In this section, we compare the results from using SVEP against those obtained by PSPIN's SCP and LHP schemes. All experiments were conducted on a network of LINUX workstations at Michigan Technological University (MTU). The MTU cluster is a network of workstations that are all 2.0 GHz P4's with 2GB of memory. In all of the experiments, partial order reduction was disabled and compression was enabled. In addition, the amount of memory that a SPIN process can consume was set to 2GB. The maximum depth of the stack was set to 300,000 for experiments using LHP and SVEP schemes as they typically produced search depths in excess of the 10,000 default used by Spin. The maximum depth for the stack was left at 10,000 for experiments using SCP. The stack depths were not fine-tuned as specifying depths greater than the actual depths do not invalidate the results, and having the same depth for both LHP and SVEP eliminated the effects of stack depth on memory usage from consideration. The Deos model used for these tests was a medium sized one, with a depth of 267,386, memory of 1000.84MB, and a run time of 98 seconds when run serially. Note that peak memory is discussed separately, after the discussions regarding other performance measures.

Results from SVEP

As stated earlier, our scheme generates a partition function based on exploring a sample of the state space. Table 3 lists results obtained using our algorithm on three sets of experiments from between two and 16 processors. In the first set, a very small amount of the state space (0.04%) was explored during the prerun. In the second set, 0.16% of the state space was explored, and 0.24% of the state space was explored in the third set of experiments. The table is arranged such that results of all three sets are grouped together for easy comparison. The first criterion examined is computation time in seconds. The table shows both the time for the PSPIN computation, and the time for the prerun in the parentheses. We note that the computation time when four through sixteen processors are used is always less than the computation time required for 2 processors. This is a good result in the sense that we do not see a trend where the computation time increases as the number of

processors is increased. The second criterion examined is the amount of memory in Mbytes reported by PSPIN. Recall that the results listed are strictly the amounts of memory required by the Spin part of the verification and do not reflect the overhead memory required by a single PSPIN process. Our tables list the memory consumed by the node that consumed the highest memory. These results show that the maximum amount of memory required to perform the Spin verification decreases as the number of processor is increased for all three sets of experiments. The third and fourth criteria measure communication overhead. The third criterion is the total number of messages transmitted as a result of the distributed computation using PSPIN. The fourth criterion is the total information transmitted in a distributed computation of PSPIN. Note that all criteria examined are generated as output of PSPIN. There are no discernable trends that can be observed from the communications results. In general, when sampling between 0.04% and 0.24% of the state space, the results can vary significantly. This is not unexpected, as our partitioning scheme is both heuristic and predictive.

In order to make the following comparisons straightforward, we will use a single result from our experiments above (and not three results). For the remainder of the paper, when comparing the results from our algorithm versus results from partitioning by SCP and Local Hash partitioning, we will use the worst result selected from each criterion given in Table 3. For example, we use a PSPIN computation time of 3230 seconds for two processors, with the corresponding prerun time of 80, and a max memory of 170.768 for 16 processors. We feel that this is a fair way to compare the schemes as results from our algorithm can vary significantly. Note that performance comparisons with respect to peak memory is discussed later in this section.

SVEP versus SCP

Table 4 is similar in format to Table 3 with the exception that the column labeled "Algo" indicates the type of algorithm used for generating the results. The results from partitioning by SCP show that the maximum amount of memory needed for Spin on any given processor is higher than the corresponding amount of memory needed by our algorithm. Also, the total number of messages communicated with SCP is much higher than the corresponding number for SVEP. In addition, the computation time required by SCP is higher than the computation time with SVEP. Thus, the SVEP scheme requires less memory than SCP, less communication and also less computation time. Therefore, our method is an improvement over SCP. This result is reasonable as our scheme can be considered as a generalization of the SCP scheme.

SVEP versus LHP

As stated earlier, we have found that LHP yields the best results when we use the local process that corresponds to the DEOS kernel to generate the partition function. The results for LHP that are listed in Table 5 were obtained by using the local process corresponding to the DEOS kernel to generate the partition function.

When comparing LHP and SVEP schemes, we notice that our scheme is much faster across the board, and especially as the number of processors increases, than using PSPIN with LHP. This is explained because both the number of messages and the total amount of information communicated in LHP is much higher than with