Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 250 (2009) 3–18
www.elsevier.com/locate/entcs

Vector Symmetry Reduction

Alastair F. Donaldson[1]

Codeplay Software
45 York Place
Edinburgh, Scotland.

Abstract

Symmetry reduction is an effective state-space reduction technique for model checking, and works by restricting search to equivalence class representatives with respect to a group of symmetries for a model. A major problem with symmetry reduction techniques is the time taken to compute the representative of a state, which can be prohibitive. In efficient implementations of symmetry reduction, a symmetry is applied to a state as a sequence of operations which swap component identities. We show that vector processing technology, common to modern computers, can be used to implement a vectorised swap operation, which can be incorporated into the representative computation algorithm to accelerate symmetry reduction. Via a worked example, we present details of this vector symmetry reduction method. We have implemented our techniques in the TopSpin symmetry reduction package for the Spin model checker, and present experimental results showing the speedups obtained via vectorisation for two case-studies running on a PowerPC vector processor.

Keywords: Model checking, symmetry, SIMD programming, parallelism, Cell BE

1 Introduction

Verification of a concurrent system by model checking [5,17] involves exhaustively searching the state-space associated with a finite state model of the system, checking whether a given temporal property holds at each state. For models of realistic systems, the size of the state-space associated with a model grows exponentially with the number of system components (where a component is e.g. a process or channel). This exponential growth means that the state-space for a system with many components may be prohibitively large, rendering straightforward exhaustive search impossible within practical limits.

Symmetry reduction techniques (see e.g. [4,10,19] or [21] for a survey) aim to alleviate the state-space explosion problem by exploiting *replication* in a concurrent system. If a system contains many identical processes connected in a regular topology (e.g. a star or clique) then a system state s belongs to a class of states which are

[1] Email: ally@codeplay.com

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2009.08.014

all identical to s, up to rearrangement of process identifiers by permutations which preserve the system topology. When checking a system property which does not refer explicitly to an individual process (e.g. deadlock freedom) it is sufficient to check a single representative state from each such symmetric equivalence class (or, alternatively, a small set of representatives from each class [3,4]). If system symmetries can be identified before search (automatically [6] or via user-supplied information [19]) then a significant reduction in memory requirements for verification can be realised [1,8,23].

The main drawback to the use of symmetry reduction is the prohibitive time which may be required to compute equivalence class representatives. A common approach to this problem involves taking the representative for a state s to be the *smallest* state in the equivalence class for s, under a suitable ordering of states [1,8]. In this case, the process of representative computation involves repeatedly applying symmetries from a group G to the state s. In efficient implementations of symmetry reduction, a symmetry is applied to a state as a sequence of operations which swap component identities. As a result, the most frequently executed operation during representative computation is that of applying a component identifier swap $(i\ j)$ to a state s. This operation can be broken down into two steps:

(i) The local state for component i at s is exchanged with the local state for component j at s

(ii) For each component identifier variable v, if $v = i$ at s then v is set to j; if $v = j$ at s then v is set to i; otherwise v is left unchanged.

For a concurrent system with many references between components, the swap operation is dominated by step (ii) above.

We present an approach to speeding up symmetry reduction by *parallelising* step (ii), the swapping of component identifiers. We develop an efficient vector swap procedure, and show that by organising a state so that component identifiers are stored contiguously it is possible to swap process identifiers in parallel using vector instructions common to modern processor architectures. We have implemented this technique in TopSpin [7], a symmetry reduction package for the Spin model checker [17], and present experimental results showing the speedups obtained by vectorising symmetry reduction for two case-studies running on the Cell Broadband Engine processor [16].

We introduce a running example and show how states for this example are represented by the explicit-state model checker Spin (§2). Using this example, we describe the process of symmetry reduction by state minimisation (§3). We then present our vector swap procedure, and show how the state minimisation process can be adapted to take advantage of vectorisation (§4). Using two case-studies, we demonstrate the effectiveness of a Spin-based implementation of our techniques (§5). The paper concludes (§7) after a discussion of related work (§6). We assume basic knowledge of the C programming language [20] throughout.

2 State Representation for Explicit State Model Checking

For illustration throughout the paper, we introduce a simple running example. We use this example to present a possible representation of states in an explicit-state model checker, as well as a representation of states suitable for formal reasoning.

2.1 Example: message passing system

Consider a simple concurrent system comprised of eight *Client* processes which route messages to one another in a peer-to-peer fashion. We refer to this example as the *message passing system*. The state of each client in the message passing system consists of an integer program counter together with two variables, *sender* and *receiver*, which holds process identifiers relevant to the message which a given client is currently routing.

The following listing shows skeleton Promela code for the message passing system. We do not specify the behaviour of a *Client* process: for our purposes it is sufficient to assume that clients are identical up to re-arrangement of process identifiers.

```
proctype Client() {

    // Program counter variable is implicit
    pid sender;
    pid receiver;

    // Behaviour
}

init {
    atomic {
        run Client(); run Client(); run Client(); run Client();
        run Client(); run Client(); run Client(); run Client();
    }
}
```

Note that in Promela each running process has an implicit program counter variable, recording the current point of execution for the process in the body of its associated proctype.

2.2 SPIN state representation for the message passing system

The following declarations illustrate the representation used by the SPIN model checker to store states for the message passing system: [2]

```
// The number of processes
#define N 8

typedef unsigned char uchar;

typedef struct Client_s {
    uchar program_counter; // implicit in the specification
```

[2] In practice, the true SPIN representation is somewhat more complex, allowing for the possibility of a dynamically changing pool of processes, and including several additional internal variables. We omit such details, which are not relevant to the presentation of our results.

```
    uchar sender;          // id variable
    uchar receiver;        // id variable
} Client;

typedef struct State_s {
    Client clients[N];
} State;
```

The state of a *Client* consists of three variables, and a state is represented by an array of $N = 8$ clients.

2.3 Reasoning about states formally

We are concerned with symmetry reduction in the general setting, where processes in a concurrent system may hold references to one another via local variables. We present some definitions (adapted from [9]) for formally reasoning about states in this model of computation.

Let $I = \{1, 2, \ldots, n\}$ be a set of component identifiers. The local state of a component is comprised of two parts, a *control* state and a *reference* state.

The control state of a component is determined by the values of all local variables of that component which are *not* references to other components, e.g. the program counter variable in the message passing system. Without loss of generality we can represent a local control state abstractly as an integer in the set $L_c = \{0, 1, 2, \ldots, k\}$ for some $k \geq 0$.

The reference state of a component is determined by the values of all local variables which *are* references to other components, for example the *sender* and *receiver* variables in the message passing system. Thus a reference state is a tuple in the set $L_r = (I \cup \{0\})^m$ for some $m \geq 0$. Here m is the number of references held by a component, and 0 is used as a default value. For the message passing system, $m = 2$ and e.g. *sender* and *receiver* can be set to 0 when a *Client* process has no message to forward.

A global state $s \in (L_c \times L_r)^n$ has the form:

$$s = ((l_1, (r_{1,1}, r_{1,2}, \ldots, r_{1,m})), (l_2, (r_{2,1}, r_{2,2}, \ldots, r_{2,m})), \ldots, (l_n, (r_{n,1}, r_{n,2}, \ldots, r_{n,m}))),$$

where $l_i \in L_c$ represents the control state of component i, and $r_{i,j} \in I \cup \{0\}$ is the value of the jth reference variable of component i ($i \in I, 1 \leq j \leq m$).

Assuming that $k \geq 24$, a potential state t for the message passing example is as follows:

$$\begin{aligned} t = (\ &(12, (1, 2)),\ (14, (2, 3)),\ (20, (1, 4)),\ (24, (2, 5)),\\ &(20, (0, 0)),\ (10, (6, 7)),\ (10, (7, 6)),\ (12, (2, 5))\ ). \end{aligned}$$

At state t, the program counter for *Client* 1 is 12, and the values of *sender* and *receiver* for *Client* 1 are 1 and 2 respectively. The *sender* and *receiver* variables for *Client* 5 are both set to the default value 0.

In the context of Promela and SPIN, a component is a process or a channel. The reference state for a process component consists of all variables of the process which have type `pid` or `chan`; the control state for a process is derived from all other

variables for the process. The reference state for a channel component consists of the fields of all messages on the channel which have type `pid` or `chan`; the channel control state is derived from the remaining message fields.

3 Symmetry Reduction for Explicit State Model Checking

We use the message passing system to provide a practical overview of symmetry reduction for explicit-state model checking, concentrating on a possible symmetry reduction implementation for the SPIN model checker. See [21] for a survey of symmetry reduction techniques.

3.1 Automorphisms and quotient models

For a system comprised of n components, let $s \in (L_c \times L_r)^n$ be as in §2.3, and $\alpha \in S_n$, where S_n is the group of all permutations of the set $\{1, 2, \ldots, n\}$. We define an action for α on s which yields another state $\alpha(s) \in (L_c \times L_r)^n$. The application of α to s can be considered as a two-stage process. For each i, first the local state of component i is replaced by the local state of component $\alpha^{-1}(i)$. Then α is applied to each reference variable of component i (with $\alpha(0) = 0$). Thus:

$$\begin{aligned}\alpha(s) = (\ & l_{\alpha^{-1}(1)}, (\alpha(r_{\alpha^{-1}(1),1}), \alpha(r_{\alpha^{-1}(1),2}), \ldots, \alpha(r_{\alpha^{-1}(1),m})),\\ & l_{\alpha^{-1}(2)}, (\alpha(r_{\alpha^{-1}(2),1}), \alpha(r_{\alpha^{-1}(2),2}), \ldots, \alpha(r_{\alpha^{-1}(2),m})), \ldots,\\ & l_{\alpha^{-1}(n)}, (\alpha(r_{\alpha^{-1}(n),1}), \alpha(r_{\alpha^{-1}(n),2}), \ldots, \alpha(r_{\alpha^{-1}(n),m}))\ ).\end{aligned}$$

where $r_{\alpha^{-1}(i),j}$ denotes the j-th reference held by component $\alpha^{-1}(i)$. For example, let t be the state for the message passing system as in §2.3, and let $\alpha = (1\ 2\ 3)$. Then

$$\begin{aligned}\alpha(t) = (\ & (20, (2, 4)),\ (12, (2, 3)),\ (14, (3, 1)),\ (24, (3, 5)),\\ & (20, (0, 0)),\ (10, (6, 7)),\ (10, (7, 6)),\ (12, (3, 5))\ ).\end{aligned}$$

Let $\mathcal{M} = (S, R)$ be the model associated with a concurrent system comprised of n components, where $S \subseteq (L_c \times L_r)^n$ is a set of states and $R \subseteq S \times S$ a transition relation on S. We say that $\alpha \in S_n$ is an *automorphism* of $\mathcal{M}$ if, for each $(s, t) \in R$, $(\alpha(s), \alpha(t)) \in R$ also. It can be shown that the set of all automorphisms of $\mathcal{M}$ forms a group, denoted $Aut(\mathcal{M})$.

In §2.1 we assumed that all *Client* processes in the message passing example are identical up to re-arrangement of process identifiers. In this scenario, we have $Aut(\mathcal{M}) = S_n$, i.e. there is full symmetry between *Client* processes.

Let G be a subgroup of $Aut(\mathcal{M})$ (written $G \leq Aut(\mathcal{M})$). Then G induces an equivalence relation on S, where the equivalence class or *orbit* of $s \in S$ is the set $[s]_G = \{\alpha(s) : \alpha \in G\}$. If *rep* is a function which maps every state to a unique representative from its equivalence class, then the *quotient* Kripke structure of $\mathcal{M}$ by G can be defined as follows: $\mathcal{M}_G = (S_G, R_G)$ where $S_G = \{rep(s) : s \in S\}$, $R_G = \{(rep(s), rep(t)) : (s, t) \in R\}$. In general $\mathcal{M}_G$ is a smaller structure than $\mathcal{M}$, but $\mathcal{M}_G$ and $\mathcal{M}$ are equivalent in the sense that they satisfy the same set of

logic properties which are *invariant* under the group G (that is, properties which are "symmetric" with respect to G). For a proof of the following theorem, together with details of the temporal logic CTL^*, see [5].

Theorem 3.1 *Let $\mathcal{M}$ be a Kripke structure, G a subgroup of $Aut(\mathcal{M})$ and ϕ a CTL^* formula. If ϕ is invariant under G then*

$$\mathcal{M}, s \models \phi \text{ iff } \mathcal{M}_G, rep(s) \models \phi.$$

In practice, Theorem 3.1 does not require the function *rep* to return a unique representative from $[s]_G$, and holds for any *rep* such that $rep(s) \in [s]_G$. An implementation of *rep* which yields unique representatives results in the maximum possible compression by exploiting symmetry. However, an implementation of *rep* which results in multiple representatives per orbit [3,4] may result in faster symmetry reduction while maintaining a reasonable state-space reduction.

3.2 Symmetry reduction by state minimisation

Given a symmetry group G, a practical approach to symmetry reduction involves choosing a subset X of G and taking $rep(s)$ to be the minimal image of s under X, with respect to a suitable total ordering on states. That is, given an ordering $\preceq$, $rep(s) = \min_{\preceq}\{\alpha(s) : s \in X\}$. The approach is formalised as Algorithm 1. To obtain full symmetry reduction we can take $X = G$. If G is large then this approach may not be feasible, and setting X to a suitable proper subset of G may result in a reasonable reduction in states with an acceptable overhead.

More sophisticated techniques for computing $rep(s)$ have been developed [1,8,9,13]. These techniques involve minimising states with respect to suitably chosen subsets of G. We present our vectorisation techniques in terms of the basic approach to symmetry reduction, but our methods can be readily applied in more sophisticated settings.

Algorithm 1 Minimising state *original* with respect to a subset X of a symmetry group.

```
procedure minimise(original, X) is
  smallest := original
  for each α ∈ X do
    temp := apply(α, original)
    if temp ≺ smallest then
      temp := smallest
    end if
  end for
  return smallest
end procedure
```

Algorithm 1 requires a function $apply(\alpha, s)$ which applies permutation α to state s. Any permutation $\alpha \in S_n$ can be represented as a product of at most $n -$

1 transpositions [15]. Given such a representation for α, the state $\alpha(s)$ can be efficiently computed by applying α to s one transposition at a time [8]. This is illustrated by Algorithm 2.

Algorithm 2 Applying permutation α to state s via incremental transpositions.

procedure *apply*(α, s) **is**
  result := s
  let $\alpha = (i_k\ j_k)(i_{k-1}\ j_{k-1})\dots(i_2\ j_2)(i_1\ j_1)$
  for d **in** $[1 \dots k]$ **do**
    result := $(i_d\ j_d)$*result*
  end for
  return *result*
end procedure

It is clear from Algorithms 1 and 2 that the most frequently executed operation during state minimisation is that of applying a transposition $(i\ j)$ to a state. In §4 we show that vectorisation techniques can be used to speed up this operation.

3.3 State minimisation for the message passing system

We now show how state minimisation can be implemented for the message passing system. Our presentation is based on the approach used by the TopSpin symmetry reduction tool [7], and extends the state representation code given in §2.2.

A permutation can be represented as a product of transpositions by the following data structure:

```
typedef struct Permutation_s {
    int numSwaps; // integer in range 0..N-1
    uchar from[N-1];
    uchar to[N-1];
} Permutation;
```

The `numSwaps` field records how many transpositions comprise a permutation α. If the k-th transposition for α is $(i\ j)$ then we have `from[`k`]`$= i$ and `to[`k`]`$= j$. Setting $N-1$ as the maximum length for `from` and `to` (where $N = 8$ is the number of *Client* processes) is sufficient since, as discussed in §3.2, any permutation $\alpha \in S_N$ can be represented as a product of at most $N-1$ transpositions.

Given this representation for permutations, the following code can be used to minimise a state:

```
void minimise(const State* original, State* smallest,
              Permutation* perms, int numPerms)
{
    State temp;

    memcpy(smallest, original, sizeof(State));

    for(int i = 0; i < numPerms; i++) {

        memcpy(&temp, original, sizeof(State));

        for(int j = 0; j < perms[i].numSwaps; j++)
           applySwapToState(&temp, perms[i].from[j], perms[i].to[j]);
```

```
        if(memcmp(&temp, smallest, sizeof(State)) < 0)
            memcpy(smallest, &temp, sizeof(State));
    }
}
```

Listing 1: State minimisation.

Listing 1 is based on Algorithm 1, with the following differences: instead of returning the minimal state `smallest`, Listing 1 assigns to this state via a pointer parameter; the call to *apply* in Algorithm 1 is replaced with a call to `applySwapToState` (code for which is given below); the $\prec$ operator is replaced with a call to the standard C function `memcmp`, which provides a suitable total ordering by comparing regions of memory byte-by-byte. Note that Listing 1 does not refer to details of the message passing system, and is therefore a general routine for state minimisation.

Implementation of the `applySwapToState` procedure does require specific details of the message passing system:

```
void applySwapToState(State* s, const int a, const int b) {

    Client tempClient;

    for(int i = 0; i < N; i++) {

        uchar id;

        id = s->clients[i].sender;
        s->clients[i].sender   = ( id == a ? b : (id == b ? a : id ) );

        id = s->clients[i].receiver;
        s->clients[i].receiver = ( id == a ? b : (id == b ? a : id ) );

    }

    tempClient = s->clients[a];
    s->clients[a] = s->clients[b];
    s->clients[b] = tempClient;

}
```

Listing 2: Applying a process id transposition to a state of the message passing system.

This procedure considers each *Client* process in turn. If the `sender` variable for a *Client* is set to `a` then the value of this variable is switched to `b`, and vice-versa. The `receiver` variable is treated similarly. The final three lines of the procedure use a temporary *Client* variable to swap the local states of *Client* processes `a` and `b`. This is in accordance with the application of permutations to states defined formally in §3.1.

4 Vectorised State Minimisation

Recall from §2.3 that the local state of a component is comprised of a reference state and a control state. The `for` loop of Listing 2 has the effect of applying the transposition $(a\ b)$ to each entry in the reference state for every component. Since there are 8 *Client* processes in the message passing system, this involves applying

16 swap operations.

In general, for a system comprised of n components where the reference state for each component has size m, $m \times n$ such swap operations are required to apply a transposition to a state. For examples where m and n are reasonably large, these swap operations dominate the state minimisation process.

We show that by reorganising the structure of a state we can use vector operations to parallelise the swapping of process identifiers.

4.1 A vector swap procedure

We develop a vector swap procedure based on data types and operations available in the AltiVec instruction set [14]. AltiVec units are common to systems based on the PowerPC architecture, in particular the Power Processing Unit of the Cell Broadband Engine Architecture [16].

We make uses of the following vector data types:

- vector unsigned char – a vector of 16 unsigned characters, abbreviated to vector uchar
- vector bool – a vector of 16 booleans,

and the following operations on these types (where, for a vector v, v_i denotes the i-th element of v):

- vec_splats(x) – takes uchar x, returns vector uchar with x in every position
- vec_cmpeq(u, v) – takes vector uchars u and v, returns vector bool w such that $w_i = \mathit{true} \Leftrightarrow u_i = v_i$
- vec_sel(b, u, v) – takes vector bool b and vector uchars u, v, returns vector uchar w such that $w_i = u_i$ if b_i is *true*, and $w_i = v_i$ otherwise.

We use an example to show how these data types and operations can be used to implement a swap operation on a single vector. Let $v = (1,3,2,4,5,4,6,7,4,5,3,3,5,1,2,3)$, $a = 3$ and $b = 5$. Suppose we wish to compute a vector, w, identical to v except that occurrences of a and b are swapped. We proceed as follows:

- Let vec_a be a vector where every entry is a: vec_a = vec_splats(a). In our example this yields $vec\_a = (3, 3, \ldots, 3)$.
- Similarly, let vec_b = vec_splats(b) $= (5, 5, \ldots, 5)$.
- Let is_a be a boolean vector indicating which entries of vector v are equal to scalar a: is_a = vec_cmpeq(v, vec_a) $= (f, t, f, f, f, f, f, f, f, f, f, t, t, f, f, f, t)$.
- Similarly, let is_b = vec_cmpeq(v, vec_b) $= (f, f, f, f, t, f, f, f, f, t, f, f, t, f, f, f)$.
- Set each entry in w to b if the corresponding entry in v is a, and to the existing entry in v otherwise: w = vec_sel(is_a, vec_b, v) $= (1, 5, 2, 4, 5, 4, 6, 7, 4, 5, 5, 5, 5, 1, 2, 5)$.
- Finally, set each entry in w to a if the corresponding entry in v is b, otherwise leave the entry of w alone: w = vec_sel(is_b, vec_a, w) $= (1, 5, 2, 4, 3, 4, 6, 7, 4, 3, 5, 5, 3, 1, 2, 5)$ as required.

This procedure for applying a swap to a single vector can be extended to operate

in-place on an array of data, as the following C code illustrates:

```
void vectorSwap(uchar* data, int size, uchar a, uchar b) {

    // Assume that size is a multiple of 16

    vector uchar vec_a = vec_splats(a);
    vector uchar vec_b = vec_splats(b);

    // Process data in vector-sized chunks
    for(int i=0; i<size; i+=16)
    {
        // Load vector of data to be swapped
        vector uchar x = *(vector uchar*)(data + i);

        vector bool is_a = vec_cmpeq(x, a);
        vector bool is_b = vec_cmpeq(x, b);

        x = vec_sel(is_a, vec_b, x);
        x = vec_sel(is_b, vec_a, x);

        // Store vector of swapped data
        *(vector uchar*)(data + i) = x;
    }
}
```

Listing 3: A vectorised swap procedure.

4.2 Using vector swap during symmetry reduction

The vector swap procedure of §4.1 cannot be directly incorporated into our state minimisation procedure since the swap procedure is designed to operate on contiguous data, and the component identifiers in a state are not all contiguous. Contiguity is a requirement for efficient loading of vector uchar data from memory into registers.

We overcome this problem as follows. Before minimising a state s, we extract all identifier variables from s and place the values of these variables in a contiguous block of memory, the *identifier block*, temporarily assigning the identifier variables in s the default value 0. With s and its identifier block in this form, we can use an efficient procedure to apply transpositions to s during minimisation. This procedure is presented below, and makes use of the `vectorSwap` outine of Listing 3. At the end of minimisation the identifier block is merged back into the resulting state, which is returned.

We now sketch the data structures and algorithms used to implement this approach for the message passing system. We refer to a state extended with an identifier block as an *augmented* state:

```
typedef struct AugmentedState_s {
    State state;
    uchar identifierBlock[2*N];
} AugmentedState;
```

Here the `identifierBlock` consists of $2 \times N = 16$ unsigned characters since there are 8 *Client* processes, each of which has two process identifier variables. For reasons of efficiency, we pad the identifier block if necessary to ensure that its size is a multiple of 16.

Given an augmented state s and component identifier values a and b, the following procedure applies the transposition $(a\ b)$ to s:

```
void applySwapToStateVectorised(AugmentedState* s, uchar a, uchar b)
{
    Client tempClient;
    uchar tempIds[2];

    vectorSwap(s->identifierBlock, N*2, a, b);

    tempClient = s->state.clients[a];
    s->state.clients[a] = s->state.clients[b];
    s->state.clients[b] = tempClient;

    tempIds[0] = s->identifierBlock[offset[a]];
    tempIds[1] = s->identifierBlock[offset[a]+1];

    s->identifierBlock[offset[a]]   = s->identifierBlock[offset[b]];
    s->identifierBlock[offset[a]+1] = s->identifierBlock[offset[b]+1];

    s->identifierBlock[offset[b]]   = tempIds[0];
    s->identifierBlock[offset[b]+1] = tempIds[1];
}
```

Listing 4: Applying transposition $(a\ b)$ to an augmented state.

The `vectorSwap` procedure (Listing 3) is first invoked to efficiently swap occurrences of a and b in the identifier block for s. The variable `tempClient` is then used analogously as in Listing 2 to swap the local states of *Client* processes a and b in the original state. This swapping of *Client* state involves a small amount of redundancy since the component identifier variables for *Client* processes are still exchanged even though these are all set to the default value 0.

The last four statements in Listing 4 have the effect of swapping the portion of the identifier block holding values relevant to *Client* a with the portion for *Client* b. This involves looking up a pre-computed array of offsets, the details of which we do not show.

An augmented state t is constructed from a regular state s by copying the contents of s into the `state` field of t, then filling the identifier block of t according to the values of reference variables in the `state` field. After a reference variable has been copied into the identifier block its value in the `state` field is set to 0. The following listing shows how augmented states are constructed for the message passing example:

```
void constructAugmentedState(const State* original, AugmentedState* s) {

    int index = 0;

    memcpy(&(s->state), original, sizeof(State));

    for(int i = 0; i < N; i++) {

        s->identifierBlock[index] = s->state.clients[i].sender;
        s->state.clients[i].sender = 0;
        index++;

        s->identifierBlock[index] = s->state.clients[i].receiver;
        s->state.clients[i].receiver = 0;
        index++;

    }
```

```
}
```

A similar function, `collapseAugmentedState`, is used to obtain a regular state from an augmented state. This function works by copying the contents of the identifier block for the augmented state back into the `state` component, which is then copied into the result state.

Combining the above routines leads to a vectorised state minimisation procedure which can be used in place of the `minimise` procedure of Listing 1 for efficient symmetry reduction:

```
void minimiseVectorised(const State* original, State* smallest,
                        Permutation* perms, int numPerms)
{
    AugmentedState originalAugmented;
    AugmentedState smallestAugmented;
    AugmentedState temp;

    constructAugmentedState(original, &originalAugmented);

    memcpy(&smallestAugmented, &originalAugmented, sizeof(AugmentedState));

    for(int i = 0; i < numPerms; i++) {

        memcpy(&temp, &originalAugmented, sizeof(AugmentedState));

        for(int j = 0; j < perms[i].numSwaps; j++)
            applySwapToStateVectorised(&temp, perms[i].from[j], perms[i].to[j]);

        if(memcmp(&temp, &smallestAugmented, sizeof(AugmentedState)) < 0)
            memcpy(&smallestAugmented, &temp, sizeof(AugmentedState));
    }

    collapseAugmentedState(&smallestAugmented, smallest);
}
```

4.3 A formal description of augmented states

Recall from §3.1 that a state s belongs to the set $(L_c \times L_r)^n$ described in §2.3. We can view an augmented state as belonging to the set $(L_c \times \{0\}^m)^n \times (I \cup \{0\})^{n \times m}$. The following rule defines an invertible mapping which converts a state to/from its augmented form:

$$\begin{aligned}&((l_1,(r_{1,1},r_{1,2},\ldots,r_{1,m})),(l_2,(r_{2,1},r_{2,2},\ldots,r_{2,m})),\ldots,(l_n,(r_{n,1},r_{n,2},\ldots,r_{n,m})))\\ \leftrightarrow\quad &(\ ((l_1,(0,0,\ldots,0)),(l_2,(0,0,\ldots,0)),\ldots,(0,0,\ldots,0)),\\ &\quad (r_{1,1},r_{1,2},\ldots,r_{1,m},r_{2,1},r_{2,2},\ldots,r_{2,m},r_{n,1},r_{n,2},\ldots,r_{n,m})\ )\end{aligned}$$

This mapping corresponds to the procedures discussed in §4.2 for constructing and collapsing augmented states.

For the message passing system, consider the state t introduced in §2.3:

$$\begin{aligned}t = (\ &(12,(1,2)),\ (14,(2,3)),\ (20,(1,4)),\ (24,(2,5)),\\ &(20,(0,0)),\ (10,(6,7)),\ (10,(7,6)),\ (12,(2,5))\ ).\end{aligned}$$

Augmenting t leads to the augmented state t':

$$t' = (\ ((12,(0,0)),\ (14,(0,0)),\ (20,(0,0)),\ (24,(0,0)),$$

$(20,(0,0)),\ (10,(0,0)),\ (10,(0,0)),\ (12,(0,0))),$
$(1,2,2,3,1,4,2,5,0,0,6,7,7,6,2,5)\ ).$

We could avoid the need for augmented states in the first place by reorganising our state structure, so that a state s belongs to the set $L_c^n \times (I \cup \{0\})^{m \times n}$. In this form, identifier variables are stored contiguously at the end of a state, which allows vectorised minimisation to be directly applied. We have presented our results in terms of augmented states to match the state structure for the SPIN model checker, illustrating the practical issues associated with implementing state-space reduction techniques in an existing model checker.

Note that our implementation of vectorised symmetry reduction is only applicable to platforms with support for the AltiVec instruction set. The vectorised approach could easily be implemented for other vector instruction sets which include equivalent instructions to those described in §4.

5 Experimental Evaluation

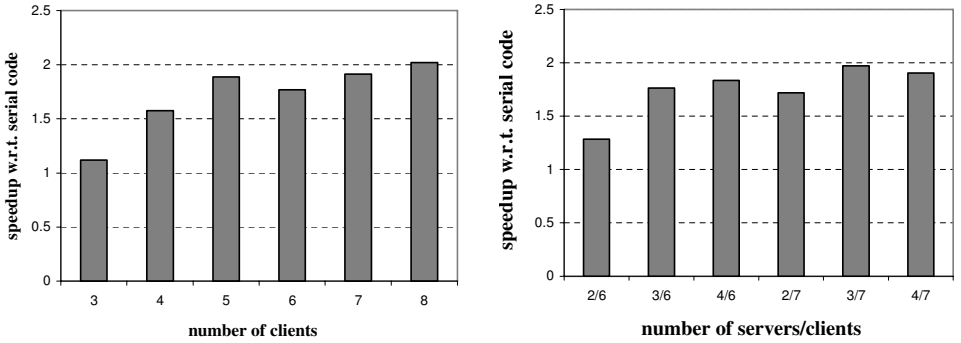
We have implemented the vector symmetry reduction technique of §4 as part of the TopSPIN symmetry reduction for the SPIN model checker. We now present experimental results showing the speedups obtained by applying vectorised symmetry reduction over standard symmetry reduction.

5.1 Experimental setup

Experiments are performed using a 3.2GHz Cell processor on a Sony PlayStation 3 console, running Fedora Linux, with IBM SDK 3.0. This is a suitable platform since the Power Processing Unit of the Cell processor is equipped with an AltiVec unit which includes the vector instructions required by our approach. The C files produced by SPIN (v4.2.6) and TopSPIN are compiled using the `ppu-gcc` compiler (v4.1.1), with the `-O2` optimisation flag. The TopSPIN *enumeration* strategy is used for symmetry reduction.

We illustrate the differences in verification time for model checking with standard symmetry reduction and with vectorised symmetry reduction using various configurations of two Promela examples: an email system, and a loadbalancer. For both examples, we verify safety properties embedded in the specifications as assertions. Results showing the state-space reduction achieved by applying symmetry reduction to the *email* and *loadbalancer* examples are presented in [9]; here we concentrate on the difference in performance between standard (serial) symmetry reduction and vectorised symmetry reduction.

The email example is adapted from [2] and used as a case study for symmetry reduction in [8,9]. A configuration of the system consists of p symmetric *client* processes, which communicate by sending messages to a *mailer* process via a *network* channel component. Components in a Promela specification of the system use reference variables to keep track of the sender and recipient of a given message. A configuration of the email example with p clients is denoted *email p*.



(a) Email configurations with 3 to 8 *client* processes.

(b) Loadbalancer configurations with 2 to 4 *server* and 6 to 7 *client* processes.

Fig. 1. Speedups obtained via vector symmetry reduction for Promela examples.

Components in a configuration of the loadbalancer example, also studied in [9], are a set of p *server* and q *client* processes with associated communication channels, and a *loadbalancer* process (with a dedicated input channel). The *load* of a server is the number of messages queued on its input channel. Client processes send requests to the loadbalancer, and if some *server* channel is not full, the loadbalancer forwards a request nondeterministically to one of the least loaded *server* queues. Each request contains a reference to the input channel of its associated client process, and the server designated by the loadbalancer uses this channel to service the request. A configuration with p clients and q is denoted p/q. There is full symmetry between the p servers and also between the q clients.

5.2 Discussion

Figures 1(a) and 1(b) show the speedups obtained by applying vectorisation to configurations of the *email* and *loadbalancer* examples respectively. For each configuration we measured the time taken for the symmetry-aware SPIN model checking algorithm to explore 1000 symmetrically distinct states, with and without vectorisation. The bar associated with each configuration indicates the speedup factor for vectorisation normalised against serial execution.

Vectorisation provides a modest speedup for the email configuration with three *client* processes. As *client* processes are added, vectorisation becomes more effective, doubling the speed of verification for the configuration with 8 clients. The reason for this increase in effectiveness is that the number of identifier variables increases with the number of clients, and so the portion of execution time dedicated to identifier swapping, and thus the potential for acceleration via vectorisation, is proportional to the number of clients. It is unclear why vectorisation does not scale well between configurations of the email example with 5 and 6 clients. Results for the loadbalancer example show similar speedups, with the vectorised model checking algorithm running between 1.3 and 1.9 times faster than the serial version.

While the speedups obtained through vectorisation for our example specifications are significant, they do not approach the theoretical maximum speedup factor for vectorisation. This factor is 16 on the PowerPC architecture, since vector operations allow 16 operations on characters to be performed using a single instruction. One reason why the speedups we have observed do not approach the theoretical limit is that memory copy and comparison operations contribute significantly to the time taken for representative computation, and these operations are *more* expensive to perform on augmented states than on standard states.

6 Related Work

To our knowledge, this is the first paper to investigate the application of parallel processing technology to symmetry reduction for model checking; research into parallel model checking has previously concentrated on distributing the state-space search algorithm over multiple workstations [22] or processor cores [18].

The approach to exploiting symmetry by applying permutations to states used throughout the paper follows the implementation of the TopSpin tool [7], which in turn is based on SymmSpin [1], another symmetry reduction package for the Spin model checker.

Not all symmetry reduction techniques work by applying permutations to states. For example, the SMC model checker [23] chooses the first state encountered for a given equivalence class as the representative for that class, and detects equivalence of states via a hash function which hashes equivalent states to the same location. Symmetry reduction via *generic representatives* [11,12] avoids computing equivalence class representatives during search by exploiting symmetry at the source code level. The vectorisation technology developed in this paper is therefore not applicable to these methods, though it is conceivable that the SMC hash function could be accelerated using vector arithmetic operations.

7 Conclusions and Future Work

We have shown that vector processing techniques can be applied to symmetry reduction for explicit state model checking. We have provided experimental results showing that vectorisation significantly speeds up verification for various configurations of two benchmark examples running on the Cell BE processor.

Our current implementation incorporates vectorisation into the *enumeration* strategy provided by the TopSpin symmetry reduction package, where the representative for a state is computed by applying each symmetry group element to the state and returning the smallest image. Future work involves applying vectorisation to a wider range of examples, and investigating the combination of vectorisation with other symmetry reduction strategies based on minimising sets [7,13] and stabilisers [1,9].

We have investigated the parallelisation of symmetry reduction at a fine level of granularity by parallelising identifier swapping, which is the innermost part of

the representative computation algorithm. Another area for future work is to investigate a coarse-grained approach to parallel symmetry reduction, distributing the representative computation algorithm across multiple processor cores.

References

[1] D. Bosnacki, D. Dams, and L. Holenderski. Symmetric Spin. *STTT*, 4(1):92–106, 2002.

[2] M. Calder and A. Miller. Generalising feature interactions in email. In D. Amyot and L. Logrippo, editors, *Proceedings of FIW 2003*, pages 187–204. IOS Press, 2003.

[3] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. Symmetry reductions in model checking. In A. J. Hu and M. Y. Vardi, editors, *Proceedings of CAV 1998*, volume 1427 of *LNCS*, pages 147–158. Springer, 1998.

[4] E. M. Clarke, S. Jha, R. Enders, and T. Filkorn. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1/2):77–104, 1996.

[5] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[6] A. F. Donaldson and A. Miller. Automatic symmetry detection for model checking using computational group theory. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *Proceedings of FM 2005*, volume 3582 of *LNCS*, pages 481–496. Springer, 2005.

[7] A. F. Donaldson and A. Miller. A computational group theoretic symmetry reduction package for the Spin model checker. In M. Johnson and V. Vene, editors, *Proceedings of AMAST 2006*, volume 4019 of *LNCS*, pages 374–380. Springer, 2006.

[8] A. F. Donaldson and A. Miller. Exact and approximate strategies for symmetry reduction in model checking. In J. Misra and T. Nipkow, editors, *Proceedings of FM 2006*, volume 4085 of *LNCS*, pages 541–556. Springer, 2006.

[9] A. F. Donaldson and A. Miller. Extending symmetry reduction techniques to a realistic model of computation. *Electr. Notes Theor. Comput. Sci.*, 2007. To appear.

[10] E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1/2):105–131, 1996.

[11] E. A. Emerson and R. J. Trefler. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In L. Pierre and T. Kropf, editors, *Proceedings of CHARME 1999*, volume 1703 of *LNCS*, pages 142–156. Springer, 1999.

[12] E. A. Emerson and T. Wahl. On combining symmetry reduction and symbolic representation for efficient model checking. In D. Geist and E. Tronci, editors, *Proceedings of CHARME 2003*, volume 2860 of *LNCS*, pages 216–230. Springer, 2003.

[13] E. A. Emerson and T. Wahl. Dynamic symmetry reduction. In N. Halbwachs and L. D. Zuck, editors, *Proceedings of TACAS 2005*, volume 3440 of *LNCS*, pages 382–396. Springer, 2005.

[14] Freescale Semiconductor. *AltiVec*, 2008. `http://www.freescale.com/altivec/`.

[15] I. Herstein. *Topics in Algebra*. John Wiley & Sons, 1975.

[16] H. P. Hofstee. Power efficient processor architecture and the Cell processor. In *Procedings of HPCA 2005*, pages 258–262. IEEE Computer Society, 2005.

[17] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[18] G. J. Holzmann and D. Bosnacki. The design of a multicore extension of the SPIN model checker. *IEEE Trans. Software Eng.*, 33(10):659–674, 2007.

[19] C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, 1996.

[20] B. W. Kernighan and D. Ritchie. *The C Programming Language, Second Edition.* Prentice Hall, 1988.

[21] A. Miller, A. F. Donaldson, and M. Calder. Symmetry in temporal logic model checking. *ACM Comput. Surv.*, 38(3), 2006. Article 8.

[22] M. Rangarajan, S. Dajani-Brown, K. Schloegel, and D. D. Cofer. Analysis of distributed Spin applied to industrial-scale models. In S. Graf and L. Mounier, editors, *Proceedings of SPIN 2004*, volume 2989 of *LNCS*, pages 267–285. Springer, 2004.

[23] A. P. Sistla, V. Gyuris, and E. A. Emerson. SMC: a symmetry-based model checker for verification of safety and liveness properties. *ACM Trans. Softw. Eng. Methodol.*, 9(2):133–166, 2000.