

Hardware Acceleration for Model Checking

Jens Brandt, Klaus Schneider and Adrian Willenbücher

Embedded Systems Group

Department of Computer Science

University of Kaiserslautern

<http://es.informatik.uni-kl.de>

Abstract

In this paper, we present a coprocessor to accelerate explicit state based model checking by computing the set of reachable states with the help of massively parallel hardware. The algorithm is based on known implementations of algorithms for the solution of the algebraic path problem on systolic arrays. We describe the mapping to a field programmable gate array, including the implementation of input and output operations. Our preliminary experimental results show a significant speedup and the applicability of our approach to real-world problems.

1 Introduction

Modern hardware and software systems are becoming more and more complex so that the formal verification of their behavior becomes mandatory to meet the tight design constraints like overall projected design costs and time-to-market. For the design of hardware systems, several kinds of model checking techniques have already successfully been employed to automatically handle this task. While the symbolic set representations, which often employs canonical normal forms for propositional logic like BDDs, have been a breakthrough in the last decade, they often turned out to not scale well with the problem sizes. Moreover, the success of their application to a given verification problem cannot be estimated in advance, since neither the size of the system in terms of lines of code nor other known metrics for the system size have proved to be useful for such estimates. Moreover, the use of BDDs is often sensible to the used variable ordering, which is sometimes difficult to determine.

For this reason, SAT-based model checking, in particular in the forms of bounded model checking and equivalence checking have recently become very popular. They still benefit from the use of symbolic methods, but tend to be more scalable as they do no longer rely on canonical normal forms like BDDs. Many algorithms used in SAT solvers could also benefit from parallel processing capabilities, even though this has not yet been a topic of the mainstream research. Usually, multiprocessor implementations [12, 16] benefit from the increased memory components and not from the computation components.

An alternative is still the use of explicit state set representations. Clearly, for most real world systems, the state spaces are far too big for a simple explicit representation. However, many techniques like partial order reduction have been developed to reduce

the state spaces so that the still prevailing model checker, namely the SPIN system, is still based on explicit state set representations.

In contrast to symbolically represented state sets, whether implemented by canonical normal forms or not, explicit state space representations can directly benefit from multiprocessor systems. Moreover, explicit state based model checking scales perfectly with the number of available processors. Hence, an appropriate hardware support can help to significantly speed up the runtimes of these algorithms.

Similar to the floating-point units of the 80s, we aim at delegating some parts of the algorithms to a separate coprocessor that offers a special hardware support for this purpose. In particular, we propose a device that computes the set of reachable states of a given model from the initial states and the transition relation. The choice of this ‘instruction set’ is motivated by the fact that the reachable states problem is the basic component of many model checking algorithms. Moreover, it is straightforward to compute other kinds of fixpoints with this coprocessor.

The rest of the paper is organized as follows: In the next section, we review previous work that forms the foundation of our implementation. Section 3 describes the design of our model checking coprocessor, which is subsequently evaluated. Finally, we conclude with a short summary in Section 4.

2 Systolic arrays and transitive closure

Systolic arrays, which have their theoretical foundation in cellular automata, were introduced by Kung and Leiserson in 1978 [6, 5, 17]. They are specialized massively parallel processors, which are well-suited for the implementation of regular algorithms. Basically, they are static networks consisting of numerous very simple data processing units (DPUs), which are usually organized in one- or two-dimensional arrays. Typically, all components run in lockstep performing the same operation. Systolic arrays are completely driven by data: Data streams flow through the array, i.e. each DPU receives data from its next neighbors, processes it and sends it to its next neighbors in the opposite direction. Inputs and outputs are only accessible at the borders of the network. Systolic arrays can be conveniently implemented by field programmable gate arrays. Their structure of small functional block arranged in a regular two-dimensional network matches the one of systolic arrays.

The implementation of algorithms on systolic arrays is a space-time mapping of the computations. Hence, only very regular algorithms described by systems of uniform recursive equations can be efficiently mapped, e. g. matrix multiplication or sorting of arrays [1, 18]. Fortunately, the computation of the reachable states in model checking procedures can be formulated in such a regular manner. Provided that the system to be verified is represented by a graph G with edges \mathcal{E} , the transitive closure of \mathcal{E} contains all reachable states (from each state). Many implementations have been proposed for the computation of the transitive closure on systolic arrays. In particular, the transitive closure implementation is a special case of the so-called algebraic path problem that also covers some other problems like the Gauß-Jordan algorithm to compute inverse matrices or algorithms to compute shortest paths between nodes in a graph. Before we discuss its applications and its implementation, we first define the problem formally:

Definition 1 (Transitive Closure of Binary Relations) *For a given binary relation $\mathcal{R} \subseteq \mathcal{M} \times \mathcal{M}$, we define the relations $\mathcal{R}^0 := \mathcal{R}$ and $\mathcal{R}^{i+1} := \mathcal{R}^i \cup (\mathcal{R}^i \circ \mathcal{R})$ where $\mathcal{R}_a \circ \mathcal{R}_b :=$*

```

module Warshall(bool  $a[N][N]$ ,  $t[N][N][N]$ )
for (int  $k = 0 \dots N - 1$ )
  for (int  $i = 0 \dots N - 1$ )
    for (int  $j = 0 \dots N - 1$ )
      if ( $k == 0$ )
         $t[i][j][k] = a[i][j] \vee a[i][0] \wedge a[0][j]$ 
      else
         $t[i][j][k] = t[i][j][k - 1] \wedge t[k][j][k - 1]$ 

```

Figure 1: Warshall Algorithm [19]

$\{(x, y) | \exists z. (x, z) \in \mathcal{R}_a \wedge (z, y) \in \mathcal{R}_b\}$. The transitive closure is then defined as $\mathcal{R}^* := \lim_{i \rightarrow \infty} \mathcal{R}^i$.

A well-known and very efficient algorithm to compute the transitive closure of a binary relation is the Warshall algorithm [19], which is shown in Figure 1. A sequential implementation of this algorithm requires $O(N^3)$ time for a given $N \times N$ input matrix. By mapping the operations onto a two-dimensional systolic array with $O(N^2)$ DPUs, the runtime can be reduced to $O(N)$ steps. Several different implementations on systolic arrays have been proposed [3, 8, 9, 11, 14] for this problem, and the approach by Kung et al. [7] has been shown to be optimal: Their array of N^2 DPUs processes a $N \times N$ matrix in only $5N - 4$ time steps, with a block period of N steps. To this end, the three-dimensional iteration space is mapped as follows to space and time: The variables j and k are mapped to processors coordinates on the array while variable i represents the time steps. After a subsequent retiming step [15, 2], a systolic array with the structure shown in Figure 2 is obtained. Since the actual data flow is not important for the subsequent sections, we do not go into details, but refer to [7].

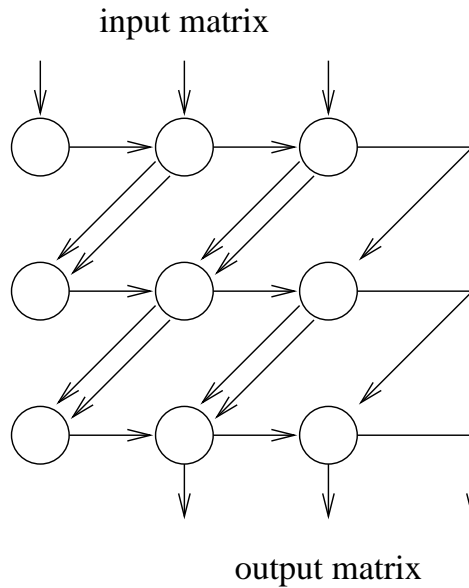


Figure 2: Systolic Array for Transitive Closure [7]

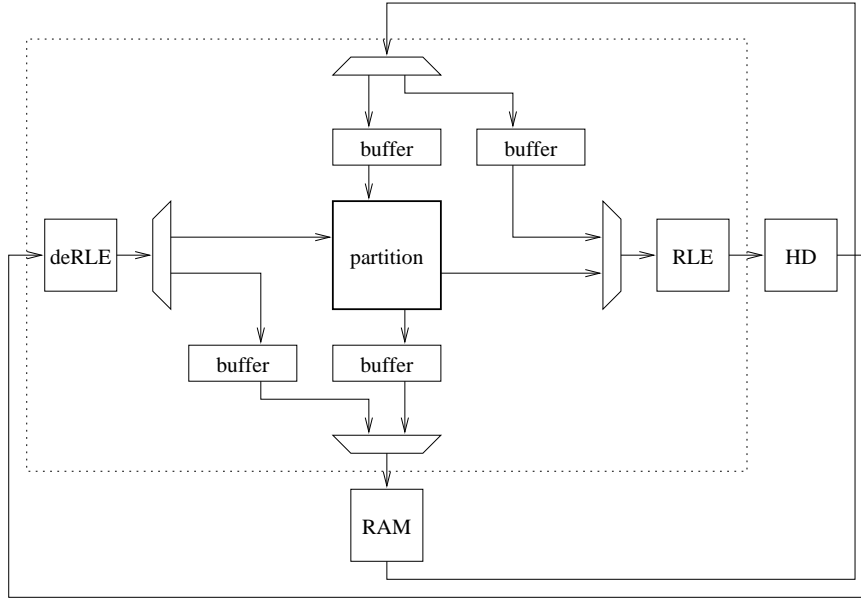


Figure 3: Structure of the Coprocessor

3 A Model-Checking Coprocessor

Figure 3 gives an overview of the structure of our model checking coprocessor. In the following subsection, we will describe how its core, the computation of the transitive closure (labelled with *partition* in the figure) is done on a systolic array. We will then proceed with the discussion of its input and its output peripherals.

3.1 Systolic Array Core

If the systolic array would be implemented as shown above, its size would be the same as the size of the given problem, which is far too large for real-world problems. Hence, we need a way to partition the array and to execute the algorithm on a fixed-sized array. Basically, there are two approaches for such partitions [10]. First, one could partition the set of states and thus the transition relation of the given graph. This is the most flexible approach, which is applicable to almost all parallel architectures. The alternative solution, which is the base of our solution, cuts the virtual systolic array of the problem size into fixed-sized pieces instead, which will be denoted in the following by *partitions*. This approach is much simpler than the first one, leading to a very efficient usage of chip area.

In order to partition the virtual systolic array, it must be sheared first: The second row is shifted by 1 position to the right, the third by 2 and so on (see Figure 4). After this transformation, all data only flows down or to the right in the array. The virtual array can now be cut into arbitrary rectangular partitions, which are processed by the coprocessor sequentially, thereby emulating the virtual array.

Processing of the partitions proceeds from top to bottom, then from left to right. This approach has two advantages: First, input data does not have to be completely available before processing can start, and second, some part of the output data is available after about half of the execution time and can already be used. As a result, only one vertical slice has to be kept in memory, while all horizontal slices need to be stored,

thus $N * N + N * n$ bits of storage are required. This is asymptotically optimal, since it has been shown in [6] that at least $O(N * N)$ bits of storage are needed.

Before the processing of the partitions begins, the input matrix is stored in compressed form on the hard disk. Since vertical slices are read from and written to RAM, the first slice needs to be loaded from disk, uncompressed and stored into RAM. After that, execution commences with the first partition. Note that data is always compressed before being written to disk, respectively decompressed after being read from disk.

For every partition in the array, a vertical (matrix data) and a horizontal slice (internal propagation) is loaded, processed, and stored. However, the operation is not always the same, since the edges of the sheared matrix must be handled differently (see Figure 5). The actual processing functions is selected by an additional control signal. For the different phases, the coprocessor does the following:

- *A1*: First, the current vertical slice is read from RAM, then it is processed. The horizontal slice is written to disk and simultaneously, the vertical slice is stored in RAM for the next step.
- *A2*: First, the current vertical slice is read from RAM and then processed in partitions. Finally, the horizontal slice is written to disk.
- *B*: This is the general operation, which is executed for all internal partitions. A vertical slice is read from RAM and a horizontal one from disk. They are processed, and the new vertical slice is written back to RAM.
- *C1*: A horizontal slice is read from disk. It is processed and written back to RAM.
- *C2*: A vertical slice is read from RAM and written to disk. In parallel, a horizontal slice is read from disk, processed, and written to RAM.

The procedure is concluded with a final reading of a vertical slice from RAM and writing to the disk, which is denoted by the arc in the lower-right corner of Figure 4. After this last step, the complete output matrix is stored in compressed form on disk.

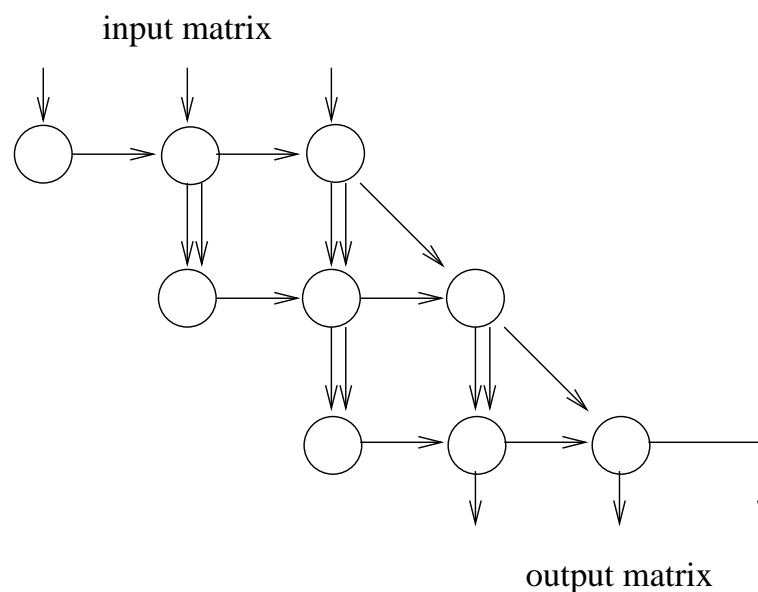


Figure 4: Partitionable Systolic Array for Transitive Closure [10]

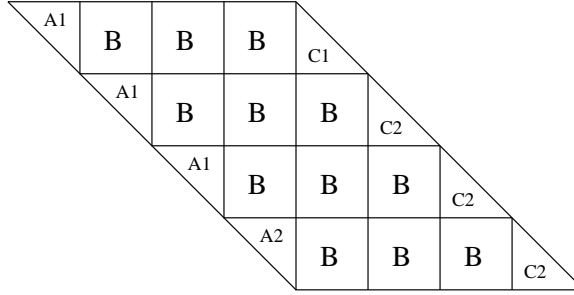


Figure 5: Coprocessor Schedule

Note that the mapping to a fixed-sized array leads to a running time of $O(N^3)$; however, the massive parallelization leads to a constant speedup. In our implementation, we select a square of size n , where the input matrix size N has to be a multiple of the edge length¹ n .

3.2 Input and Output Connections

The most critical point of the realization of the systolic array is the design of its input and output connections. Our application needs a very high bandwidth of input and output streams since the matrices storing the transition relation and the set of states are generally very large.

For instance, a partition with 32×32 DPUs, running only at 200 MHz, has a total bandwidth of $32 \cdot 3 \cdot 2e8 \text{bit/s} = 1.2 \text{GB/s}$ at its edges. In each cycle, it consumes 64 bits and produces 32 bits, except for the left and right borders of the array, where only 32 bits are consumed. Although current RAM running at a bus frequency of 200 MHz supports these data rates, 256 MB of available memory would limit input matrices to an edge length of about $N = 46300$, assuming a partition size of $n = 32$: $N * N + N * n \leq 256 * 8 * 2^{20}$.

In order to process larger matrices, intermediate results have to be written to a hard disk, which provides the appropriate amount of storage. However, current high-end disks can sustain a write-/read-rate of about 100 MB/s at most. Although a combination of preloading and reusing data can compensate for these low data rates, this approach limits the maximum size of input matrices that can be efficiently processed.

Therefore, our solution is a mix of both approaches. We store the current vertical slice in RAM, since it is continuously read and written, and store all horizontal slices on disk. In theory, this also constrains the size of input matrices, but now the limit is not relevant. For the above example, the maximum edge length of input matrices is $N \approx 67,100,000$ ($67,100,000 * 32 \text{ bits} < 256 \text{MB}$) with an execution time of about 46,000 years ($= N^3 / (n \cdot n) / 200 \text{ MHz}$).

To overcome the bandwidth limitation of the hard disks, the horizontal slices are compressed before writing and decompressed after reading. Since the matrix representing the transition relation tends to be sparse, this generally leads to very compact representations. Although compression algorithms like arithmetic coding [4] would give excellent compression ratios for this kind of data, they have two significant drawbacks: First, 32 instances (and again the same number for decompression) would cover

¹This is no restriction, since the input matrix can be padded with zeros to achieve this, the resulting overhead is negligible.

an inordinate area on chip, and second, they cannot always consume one input symbol per cycle, which would lead to a frequent stalling of the computation array. Instead, we compress data with the help of simple run-length encoding (RLE): In a bit stream, every sequence of zeros is replaced by a single zero followed by a fixed number of bits representing the length of the sequence; ones in the stream are not replaced. Moreover, its implementation is a very simple and therefore small finite state machine.

A very interesting relation between the processor parameters and its performance is revealed by the following considerations: Let f be the operational frequency of the partition. Then, the required bandwidth b of the connection between partition and storage is $b = (n \cdot 3)\text{bit} \cdot f$, while the the execution time t is $t = \frac{N}{f} \cdot \frac{N^2}{n}$.

Hence, it can be seen that, if n is multiplied by a factor k and f is divided by this factor k , the bandwidth of the partition remains constant, while the execution time is divided by k . This means that, given a larger chip, the performance can be increased while the peripheral devices (RAM and hard disk) do not need to be replaced; not even the circuitry that handles the (de)compression needs to be improved. Thus, the design can additionally profit from the permanent relative growth of chip size compared to memory bandwidth.

3.3 System Performance

We implemented the whole design on a Xilinx Virtex-5 XC5LX50T (speed grade -1), which has access to 256 MB of DDR2 RAM. While the 32×32 DPUs of the systolic array can run at more than 400 MHz, the surrounding logic (compression, RAM- and SATA-controller) currently cannot keep up with that frequency without further optimizations (e.g. pipelining). For this reason, the whole design is clocked at 200 MHz, thus executing $200 \cdot 10^6 \cdot 32 \cdot 32 = 204.8 \cdot 10^9$ Warshall operations per second. Hence, without any external reductions a full input matrix with edge length $n = 10^4$ takes only 5 seconds to be processed, one with $n = 10^5$ 80 minutes and one with $n = 10^6$ about 57 days.

Each DPU contains only a relatively simple combinatorial function (including handling of special cases), but needs six flip-flops. Some of these can be saved by the synthesis tool since the control signals propagate down as well as to the right. However, the limited length of one-hop signals in FPGAs places a limit to this. This leads to logic (lookup-tables) being unused and thus to a waste of chip area. Instead, the calculations of four PEs (or, in fact, any rectangular formation of PEs) can be replaced by a single PE. This way, the flip-flops used to model the propagation between these PEs are no longer needed, thereby allowing a larger partition size $n \times n$ on the same area. Note that this leads to a decrease in speed within the partition, which should however be outweighed by the larger partition size n and the resulting shorter overall execution time.

Further, the partitions do not need to be quadratic, as long as they are rectangular. The total execution time is determined by the number of DPUs in a partition: $t = \frac{N}{f} \cdot \frac{N}{n_w} \cdot \frac{N}{n_h}$, where n_w is the width of a partition, and n_h is its height. The bandwidth to the RAM and to the hard disk are proportional to n_w and n_h , respectively. If, for example, a second RAM is attached to the FPGA, n_w could be doubled and n_h halved, thereby reducing the bandwidth to and from the hard disk.

4 Summary

In this paper, we presented the design of a model-checking coprocessor based on a systolic array implementation of the transitive closure problem. We discussed how the transitive closure problem can be mapped to a systolic array of limited size to handle transition systems of arbitrary size. We discussed various obstacles that have to be overcome for an efficient implementation of the given algorithms and gave solutions to them. Although not all optimizations have been implemented yet, the overall system performance is adequate for real-world problems.

References

- [1] L. Adleman, K.S. Booth, F.P. Preparata, and W.L. Ruzzo. Improved time and space bounds for Boolean matrix multiplication. *Acta Informatica*, 11(1):61–70, 2004.
- [2] M. Bartha. Strong retiming equivalence of synchronous schemes. In J. Farré, I. Litovsky, and S. Schmitz, editors, *Conference on Implementation and Application of Automata (CIAA)*, volume 3845 of *LNCS*, pages 66–77, Sophia Antipolis, France, 2006. Springer.
- [3] E. Fink. A survey of sequential and systolic algorithms for the algebraic path problem. Technical Report CS-92-37, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1992.
- [4] P.G. Howard and J.S. Vitter. Practical implementations of arithmetic coding.
- [5] H.T. Kung. Why systolic architectures? *IEEE Computer*, 15(1):37–46, 1982.
- [6] H.T. Kung and C.E. Leiserson. Systolic arrays (for VLSI). In *Sparse Matrix Proceedings*, pages 256–282. Society for Industrial and Applied Mathematics, 1978.
- [7] S.-Y. Kung, S.-C. Lo, and P.S. Lewis. Optimal systolic design for the transitive closure and the shortest path problems. *IEEE Transactions on Computers*, 36(5):603–614, 1987.
- [8] K.N. Levitt and W.H. Kautz. Cellular arrays for the solution of graph problems. *Communications of the ACM*, 15(9):789–801, 1972.
- [9] P.S. Lewis and S.Y. Kung. Dependence graph based design of systolic arrays for the algebraic path problem. In *Asilomar Conference on Signals, Systems, and Computation*, pages 13–18. IEEE Computer Society, 1986.
- [10] B. Lisper. Computing transitive closure on systolic arrays of fixed size. *Distributed Computing*, 5(3):133–144, 1991.
- [11] Z. Milovanovic, E.I. Milovanovic, and B.M. Randjelovic. Computing transitive closure problem on linear systolic array. In Z. Li, L. Vulkov, and J. Wasniewski, editors, *Numerical Analysis and Its Applications*, volume 3401 of *LNCS*, pages 416–423, Rouse, Bulgaria, 2005. Springer.

- [12] K. Milvang-Jensen and A.J. Hu. BDDNOW: A parallel BDD package. In G. Gopalakrishnan and P.J. Windley, editors, *Formal Methods in Computer Aided Design (FMCAD)*, volume 1522 of *LNCS*, pages 501–507, Palo Alto, California, USA, 1998. Springer.
- [13] M.J. Quinn and N. Deo. Parallel graph algorithms. *ACM Computing Surveys (CSUR)*, 16(3):319–348, 1984.
- [14] G. Rote. A systolic array algorithm for the algebraic path problem. *Computing*, 34(3):191–219, 1985.
- [15] N. Shenoy. Retiming: Theory and practice. *Integration, the VLSI Journal*, 22(1-2):1–21, 1997.
- [16] Tony Stornetta and Forrest Brewer. Implementation of an efficient parallel bdd package. In *Proceedings of the 33rd annual conference on Design automation*, pages 641–644. ACM, 1996.
- [17] C.D. Thompson and H.T. Kung. Sorting on a mesh-connected parallel computer. *Communications of the ACM*, 20(4):263–271, 1977.
- [18] M. Veldhorst. Parallel dynamic programming algorithms. In W. Händler, D. Haupt, R. Jeltsch, W. Juling, and O. Lange, editors, *Algorithms and Hardware for Parallel Processing*, volume 237 of *LNCS*, pages 393–402, Aachen, Germany, 1986. Springer.
- [19] S. Warshall. A theorem on boolean matrices. *Journal of the ACM (JACM)*, 9(1):11–12, 1962.