APmap
An Open-Source Compiler for Automata Processors

Yu, Jintao; Lebdeh, Muath Abu; Du Nguyen, Hoang Anh; Taouil, Mottaqiallah; Hamdioui, Said

DOI
10.1109/TCAD.2021.3062328

Publication date
2021

Document Version
Final published version

Published in
IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

APmap: An Open-Source Compiler for Automata Processors

Jintao Yu, *Student Member, IEEE*, Muath Abu Lebdeh, *Student Member, IEEE*, Hoang Anh Du Nguyen, Mottaqiallah Taouil, *Member, IEEE*, and Said Hamdioui, *Senior Member, IEEE*

***Abstract*—A novel type of hardware accelerators called automata processors (APs) have been proposed to accelerate finite-state automata. The bone structure of an AP is a hierarchical routing matrix that connects many memory arrays. With this structure, an AP can process an input symbol every clock cycle, and hence achieve much higher performance compared to conventional architectures. However, the design automation for the APs is not well researched. This article proposes a fully automated tool named *APmap* for mapping the automata to APs that use a two-level routing matrix. APmap first partitions a large automaton into small graphs and then maps them. Multiple transformations are applied to the automaton by APmap to meet hardware constraints. The experiments on a standard benchmark suite show that our approach leads to around 19% less storage utilization compared to state-of-the-art.**

***Index Terms*—Automata processor (AP), design automation, graph partitioning, mapping.**

I. Introduction

Finite-state automata (FSA) are widely used in domains such as network security [1], bioinformatics [2], and artificial intelligence [3]. Some innovative hardware designs repurpose memory array for accelerating FSA execution, e.g., micron automata processor (MAP) [4], Cache Automaton [5], and RRAM-AP [6]. These accelerators store many states in memory arrays and distribute each input symbol to all the states simultaneously. Based on the input symbol, a state activates other states via a hierarchical routing matrix. These actions are repeated every clock cycle, and hence these accelerators achieve much high throughput [4], [5], [7]. We refer to these accelerators as automata processors (APs). The routing matrix mimics the transition function of FSA. It is implemented with memory arrays that are connected with rich wiring. The routing matrix is configured for specific FSA by writing configurable bits to the memory array. For example, the routing matrix of Cache Automaton connects 32 k states and contains 10 M configurable bits [5]. Therefore, design automation is required for mapping FSA to the APs.

Currently, there are no open-source design tools available for the APs. The authors of Cache Automaton described their methodology of mapping FSA to the hardware. However, not all the details are explained, and their tool is not publicly available. Following their methodology, some FSA cannot be mapped directly due to the constraints on the routing matrix. As a result, these FSA have to be transformed into other equivalent forms [5]. This step is iterative and requires experience. As for other related works, Micron provides a commercial software development kit (SDK) for MAP. Since this SDK is closed-source, it cannot be adapted for other architectures such as Cache Automaton. The compiler of RAPID can generate mapping by duplicating an initial result, e.g., produced by MAP SDK [8]. Therefore, it cannot be used alone. Wadden *et al.* [9] have developed an open-source tool named *ATR* to estimate the resource needed for mapping an application to MAP. This tool is based on *VPR*, a routing tool that targets a 2-D-mesh structure such as FPGAs. This structure is different from the hierarchical routing matrix of APs, and hence ATR cannot produce accurate results. While open-source tools, such as REAPR [10] and Grapefruit [11], have been proposed to map applications to FPGAs, a similar one that targets APs is still needed.

This article addresses the above issues and presents *APmap*[1] (automata processor mapping tool), an open-source compiler for APs that are based on a two-level routing matrix, such as Cache Automaton and RRAM-AP. Note that APmap cannot be applied to MAP due to its algorithm limitation. APmap uses multiple strategies to change given FSA to equivalent forms so that they can meet the constraints of the routing matrix. Therefore, the compilation process does not require any user involvement. The main contributions of this article are as follows.

1) A methodology to automatically map automata to APs that are based on a two-level routing matrix. The methodology optimizes the *storage utilization.*
2) An *open-source* tool APmap based on the proposed methodology. This tool can be adapted to various designs by altering its parameters.
3) An evaluation of APmap and comparison with state-of-the-art.

The remainder of this article is organized as follows. In Section II, we explain the working principle and the routing matrix of APs. Section III presents the methodologies of APmap. Next, Section IV evaluates APmap's performance using ANMLzoo. After a brief discussion in Section V, Section VI concludes this article.

II. Background

A. Automata Processors

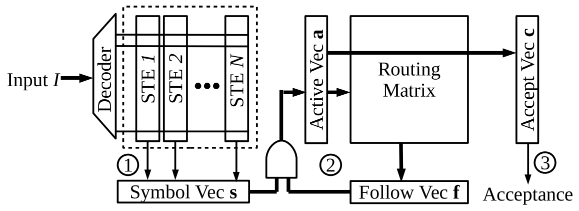

Fig. 1. General architecture of APs [6].

The APs share a generalized architecture as shown in Fig. 1 [6]. In every clock cycle, an input symbol I is processed using three major steps.

1) *Input Symbol Matching:* All the states that have incoming transitions occurring on I are identified in this step. Each state is presented by column vectors called state transition element (STE) that are preconfigured based on the targeted automaton. The decoder activates one of the word

Manuscript received August 27, 2020; revised November 20, 2020 and February 10, 2021; accepted February 13, 2021. Date of publication February 26, 2021; date of current version December 23, 2021. This article was recommended by Associate Editor A. Gamatie. *(Corresponding author: Jintao Yu.)*

The authors are with the Laboratory of Computer Engineering, Delft University of Technology, 2628 CD Delft, The Netherlands (e-mail: j.yu-1@tudelft.nl; m.f.m.abulebdeh@tudelft.nl; h.a.dunguyen@tudelft.nl; m.taouil@tudelft.nl; s.hamdioui@tudelft.nl).

Digital Object Identifier 10.1109/TCAD.2021.3062328

[1]APmap can be downloaded at https://github.com/yjt98765/apmap.

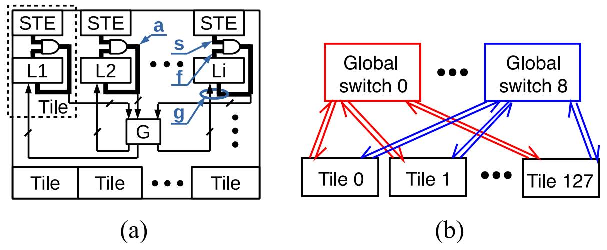

Fig. 2. Detailed structure of cache automaton and RRAM-AP. (a) AP chip structure. (b) Routing matrix.

lines according to the input symbol I. If an STE has an incoming transition occurring on I, its output is logic 1; otherwise, the output is logic 0. The outputs of all STEs are mapped to a vector called Symbol Vector **s**.

2) *Active State Processing:* It generates all the possible states that can be reached from the currently active states (stored in Active Vector **a**) based on the transition function (stored in the routing matrix), and stores the result in the Follow Vector **f**. This step also generates the next active states by bit-wise ANDing **s** and **f**.
3) *Output Identification:* Accept Vector **c** is preconfigured based on the automaton's accepting states C. This step checks the intersection of **a** and **c** to decide whether the input sequence is accepted.

Multiple components, including STEs, the routing matrix, and Accept Vector **c**, need to be configured based on the targeted FSA. The configuration will be generated by APmap.

B. Routing Matrix

The routing matrix implements the transition function of an automaton. Its input and output are Active Vector **a** and Follow Vector **f**, respectively. The lengths of these two vectors are both N, i.e., the state number of the automaton. Each member in the vectors is a Boolean value, corresponding to an automaton state. The routing matrix of the existing APs all consist of multiple components that are linked in a hierarchical style. The routing matrix of MAP contains four levels; Cache Automaton and RRAM-AP contain two, i.e., global and local switches. In Cache Automaton and RRAM-AP, the global switches are located at the center of the chip while the local switches are distributed. 256 STEs, a local switch, 256 AND gates, and a decoder are grouped as a *tile*, as shown in Fig. 2(a). The input symbol I is sent to all the tiles in parallel.

APmap targets the routing matrix of the space-optimized design of Cache Automaton, which consists of 128 tiles, eight 1-way global switches (G1), and a 4-way global switch (G4). The connection between the tiles and global switches is shown in Fig. 2(b). Each tile has two input wires from and two output wires to every G1. Each tile also has eight input wires from and eight output wires to the G4. In total, a tile has 24 input and 24 output wires.

III. APMAP METHODOLOGIES

We suggest several other tools to be used together with APmap to develop applications targeting APs. The application can be coded in RAPID, a high-level programming language designed for pattern-recognition processors such as APs [8]. RAPID's compiler generates automata network markup language (ANML), an XML-based format for describing automata [12], files as output. The ANML can be parsed by *VASim* [13], a tool that simulates the execution of a homogeneous automaton. It also supports some important automata transformations, such as prefix merging. We modified VASim to preprocess the automata and generate the file formats used by APmap. These files describe the automata as a collection

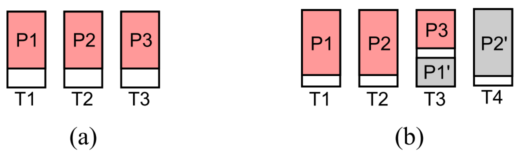

Fig. 3. Different styles of partitioning a graph. (a) Balanced partitioning. (b) Unbalanced partitioning.

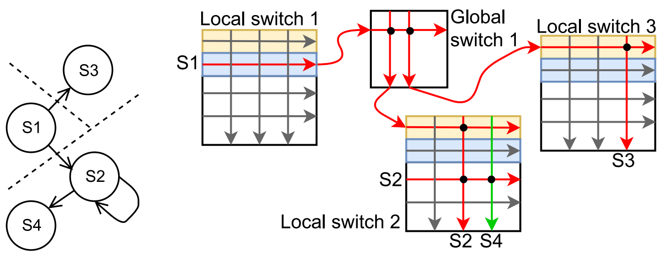

Fig. 4. Mapping an example automaton to AP. The automaton is partitioned into three parts, which are mapped to Tiles 1 to 3, respectively. The input signal of the global switches origin from the blue region of the local switches while the output signals enter the yellow region of the local switches. The black dots indicated that the row can activate the corresponding column.

of connected components (CCs), i.e., nonoverlapping subsets of the original automata. Finally, APmap produces the configuration files for APs.

APmap first sorts the CCs by their state numbers and then maps them one by one. In each iteration, APmap picks the largest unmapped CC and maps it to one or multiple tiles. In some cases, not all the space of these tiles are occupied by this CC. Therefore, APmap tries to find some small CCs to fill in the remaining space. This process repeats until all the CCs are mapped.

When a CC contains more than 256 states, it has to be mapped to multiple tiles. First, this CC is partitioned into several parts, which will be presented in detail in Section III-A. To increase the chance of mapping success, the partitioning process produces multiple solutions. Then, APmap examines the partitioning results with the hardware constraints on a tile. If there are conflicts, an extra process is applied to resolve the conflicts, which will be presented in Section III-C. Next, APmap tries to generate the configuration for the global switches and the tiles, which will be presented in Section III-B. If the configuration of global switches cannot be generated, APmap selects the next partitioning solution and repeats the previous processes. If none of these partitioning solutions leads to a valid configuration, the mapping flow fails.

A. CC Partitioning

APmap partitions a CC with the help of *METIS* [14], a widely used graph partitioning tool. METIS divides an undirected graph into k nonoverlapping parts while trying to cut the least number of edges. An edge is *cut* in a division means that the two nodes linked by the edge are assigned to different parts. First, we transform the CC to an undirected graph to make it acceptable for METIS. Self-loops are removed in this transformation. Then, we invoke METIS with two types of input parameters: 1) the number of parts and 2) the size constraints on those parts. The size of a part means the number of nodes contained in that part, and a size constraint is the desired size of a part. METIS produces the same number of parts as required; however, the size constraints are not guaranteed to be satisfied. Therefore, we need to check the size of each part after the partitioning. If any part contains more than 256 nodes, we need to modify parameters and invoke METIS again.

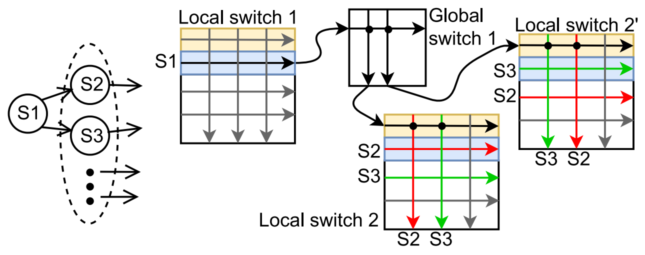

Fig. 5. Example of resolving an output constraint conflict. Assume a part contains more outgoing states than the constraint. To resolve this conflict, Local switch 2, including input signals (e.g., coming from $S1$), is duplicated as Local switch 2'. The outgoing states (e.g., $S2$ and $S3$) are split between these two tiles.

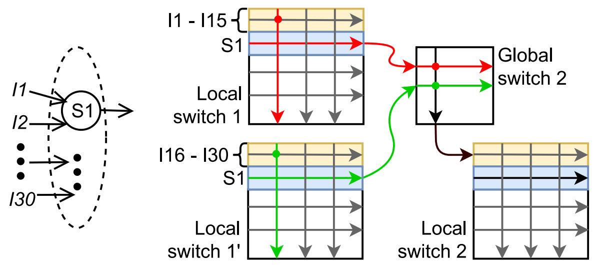

Fig. 6. Example of resolving an input constraint conflict. Assume a part requires 30 incoming signals (i.e., $I1$ to $I30$), which exceeds the constraint. To solve this conflict, Local switch 1 is duplicated as Local switch 1'. The incoming signals are split between these two switches. The outgoing states (e.g., $S1$) in these two switches activate other states together.

This partitioning process is iterated with different parameters to lower the *storage utilization*, or *utilization* for short, which is referred to as the number of tiles that the automaton is mapped to. It is influenced by the partitioning result from three aspects. First, each partitioned part will be mapped to different tiles; therefore, the number of parts is the most important factor for the final utilization. Second, the cutting edges will be mapped to *global wires*, i.e., the wires connecting tiles and global switches. The wire resource is limited. Therefore, a partitioning result that contains many cutting edges may lead to overhead for resolving the constraint conflict. Third, the balancing of partitioned parts may also affect utilization. Fig. 3 shows two possible partitioning styles of a CC. It is partitioned into three parts, i.e., $P1$, $P2$, and $P3$, represented by gray rectangles. The size of the rectangle indicates the size of the part, and the total size of the three parts are equal in the two partitioning styles. These parts are mapped to three tiles, i.e., $T1$, $T2$, and $T3$, represented by transparent rectangles. In the balanced partitioning, as shown in Fig. 3(a), all the parts have similar sizes. In the unbalanced partitioning, as shown in Fig. 3(b), the sizes of the first two parts are close to the size of a tile, while the third part is relatively small. This is an important feature as the whitespace in $T3$ can be used for mapping a small CC. Alternatively, it can also be used to map a part of another large CC [indicated by $P1'$ and $P2'$ in Fig. 3(b)]. On the contrary, the whitespace in Fig. 3(a) is only enough for fitting tiny CCs, which are rare in automata benchmarks. We prefer the unbalanced style during partitioning since it provides more optimizing opportunities.

B. Mapping Method

This section focuses on the configuration of the routing matrix, i.e., the global and local switches. The configuration of STEs, i.e., their associated input symbols and whether they are the start or final states, is generated by VAsim. APmap simply copies this information to the final configuration file.

The transitions among the states in different tiles are mapped to multiple global and local switches in two steps: first to configure global switches and then the local switch part. Fig. 4 uses an example to illustrate these steps. Assume the four states in the automaton are partitioned into three parts, i.e., {$S2$}, {$S2$ and $S4$}, and {$S3$}, and these parts will be mapped to Tiles 1, 2, and 3, respectively. For both global and local switches, the input signals connect to the rows, and the columns generate the output signal. Dots indicate that the column is connected to the row, i.e., when the row is activated, the column also activates. First, APmap selects a global switch, e.g., Global switch 1, that has at least one *free* wire with Tile 1. Here, free means that it has not been assigned for mapping other transitions. Next, APmap checks the connections between Global switch 1 and Local switch 2 and 3. Similarly, it requires at least one free wire. If the above requirements are all satisfied, then the global switch part is successfully mapped. $S1$ will be placed at the slot that connects Global switch 1. Note that only 24 slots can output its signal to global switches, and these slots are illustrated by the blue region. Similarly, only 24 slots receive signals from the global switches, and they are colored yellow. All the wires assigned in this step are colored red in Fig. 4. If any condition is not satisfied, then APmap selects the next global switch and checks again. If none of the global switches meet the requirement, the mapping process fails.

After all the transitions being mapped to global switches, detailed mapping to local switches consists of two parts. The first part is to map the transitions within a tile, e.g., two dots are placed in the row of $S2$ in Fig. 4, implementing the transitions from $S2$ to $S2$ and $S4$. The second part is to map the signals that come from global switches, i.e., configuring the dots on the red region in Local switch 2 and 3 in Fig. 4. They can be conducted similarly.

C. Meeting Constraints

Constraint violation is a result of hardware resource limits and unsatisfactory partitioning. When a large CC is partitioned into several parts, the number of transitions among these parts is unconstrained. Although METIS tries to minimize the total transition numbers, it is possible that the transition number exceeds the number of wires that connect a tile with global switches. In this case, we need to resolve this conflict before mapping it to the tile.

In this section, we first present the methods for resolving output constraint conflicts and then the input. As introduced in Section II-B, a tile has only 24 output wires that connect to global switches. Any state that transits to states in other tiles, referred to as an *outgoing* state, has to be mapped in those 24 slots. Assume that a partitioned part contains more outgoing states than the constraint, including two states, $S2$ and $S3$. We duplicate this part and keep only half of them as outgoing states in each copy. Fig. 5 shows a possible mapping result where these two parts are mapped to Tile 2 and 2', respectively. $S2$ is regarded as an outgoing state in Local switch 2 but not in Local switch 2'. $S3$ is the opposite. In this way, they can both activate other states. Note that all the input signals are also duplicated. Assume that $S1$, which is mapped to Tile 1, activates $S2$ and $S3$. After the duplication, the global switch maps the output of $S1$ to both Local switch 2 and 2'. Therefore, the execution process of the CC is unchanged.

APmap resolves input constraint conflicts by duplicating as well. However, comparing with output constraint resolving, one additional configuration is required. Assume a part contains 30 input signals named from $I1$ to $I30$, which exceeds the 24-input constraint. It is duplicated and assigned to Tile 1 and 1', respectively. The 30 inputs are also divided into two groups and assigned to those tiles. For the outgoing states in this part (e.g., $S1$), the duplicates activate the states in other parts together as shown in Fig. 6. This configuration guarantees the correctness of automata execution. Assume $S1$ can be

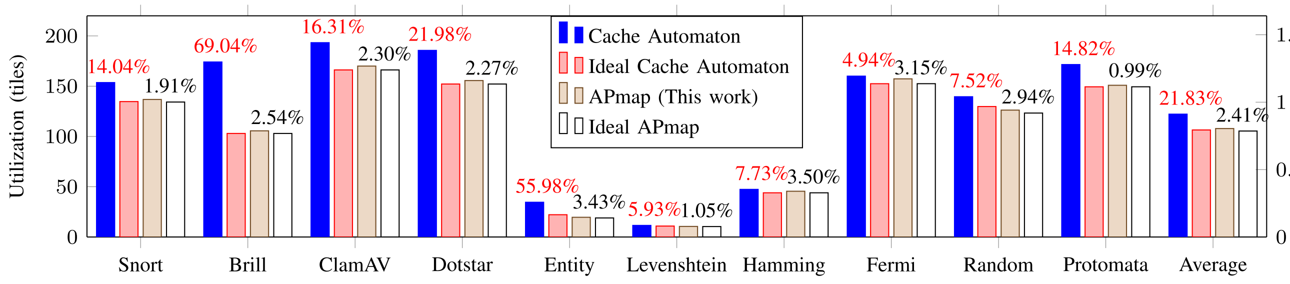

Fig. 7. Utilization comparison between APmap and Cache Automaton. The numbers above the bars illustrate the percentages that the actual utilization exceeds the ideal one.

TABLE I
REQUIRED TILE NUMBERS WHEN MAPPING THE APPLICATIONS UNDER THREE DIFFERENT CONFIGURATIONS

Benchmark	Snort	Brill	ClamAV	Dotstar	Entity	Levenshtein	Hamming	Fermi	Random	Protomata	Average
8 G1 + G4	136.8	105.6	170.0	155.6	19.6	10.5	45.5	157.3	126.9	150.9	107.87
4 G1 w/ OPT	137.6	107.8	170.0	156.6	22.6	11.8	46.5	157.3	126.9	151.2	108.83
4 G1 w/o OPT	NA	NA	170.0	156.6	NA	11.8	46.5	157.3	126.9	151.2	NA

activated by (one of) the input signals $I1$ to $I30$. After the duplication, the $S1$ in either Local switch 2 or 3 is (or both of them are) activated, and it (or they) will further activate other states through the red or the green paths in the figure.

In general, if a part contains N outgoing states or incoming signals, it will be duplicated $\lceil (N/24) \rceil - 1$ times, and these states or signals are distributed equally in these duplicates. If a part has both output and input constraint conflicts, it is duplicated for resolving the output constraint conflict first, and then the input constraint conflict in each duplicate is resolved individually.

IV. Evaluation

A. Evaluation Methodology

We adopt ANMLzoo as the benchmark suite in the evaluation since it is widely used for evaluating APs, especially Cache Automaton [5]. Two benchmarks in this suite, i.e., BlockRings and CoreRings, contain large CCs that exceed the capacity of an RRAM-AP or Cache Automaton chip. Therefore, they are excluded from this evaluation.

We use the latest commit of VASim to parse and optimize ANMLzoo benchmarks. Only the optimized automata will be used in the evaluation because their CC sizes are larger and hence more challenging for mapping tools. The state numbers after optimization are slightly different from that in Cache Automaton's paper [5], probably because of the usage of different VASim versions. The two notable exceptions are PowerEN and SPM. The state numbers are so different in the two works that they are not representing the same benchmark. Therefore, we will exclude them from the evaluation.

In the first experiment, we evaluate the performance of APmap. The hardware target for the mapping is two AP chips with a full routing matrix. No wires are connecting these chips. In the second experiment, we demonstrate the effect of constraint conflict resolving optimizations by mapping the applications to a routing matrix with only four G1.

B. Experimental Results

When mapping the benchmarks to the full routing matrix, the utilization of the results is shown in Fig. 7. The utilization of Cache Automaton's mapping tool [5] and the *ideal* utilization for these two cases are also shown as a comparison. The results in [5] are reported using MBs (see the right y-axis) and it is interchangeable with the number of tiles. A tile contains 256 STEs whose size is 32 bytes each, and hence a tile occupies 8 kB. Ideal utilization is referred to the minimum amount of memory required for mapping an automaton in theory, i.e., the product of the state number and the STE size. The ideal utilization may never be achieved due to the input and output constraints and the imperfect partitioning. However, it can be used to measure the ability of the mapping tools. The *overhead*, the percentage by which the actual utilization exceeds the ideal, of both tools is illustrated above the bars. As the overhead is calculated using separate bases, it is a fair comparison for these two tools. For all the benchmarks, APmap achieves a lower overhead than Cache Automaton. In addition, APmap's overhead is always below 4%, with an average of 2.41%. On the Cache Automaton side, however, the average overhead is 21.83%, and the highest is more than 50% (Brill and Entity). No tiles are duplicated in this evaluation as all the input/output constraints are satisfied.

We map the applications to a routing matrix consisting of only four G1, with and without constraint conflict resolving optimizations. The results are listed in Table I with a comparison to the previous one. With fewer global switches, more tiles are needed due to constraint conflict resolution and less compact partitioning. When the optimizations are disabled, Snort, Brill, and Entity cannot be mapped. It suggests that these optimizations are essential for some complex applications. Note the partitioning a graph to more parts does not necessarily decrease the number of cutting edges in a part. Therefore, without these optimizations, some automata cannot be mapped.

V. Discussion

This section highlights two main advantages brought with APmap: 1) allowing more applications deployed in a chip and 2) assisting hardware design space exploration.

Deploy More Applications: When two applications share a subset of an automaton, they can be merged and optimized as a single application. In Section IV, we have shown APmap's benefit in mapping a complex application. When two applications share no common states, they can be mapped to the same chip without conflicts in global switches. Assume application A is mapped to Tiles 1 and 2, which are connected by Global switch 1. Now, we map application B to Tiles 3 and 4, which are connected by Global switch 1 as well. As indicated by different colors in Fig. 8, the transitions in an application do not affect the other. For example, APmap successfully mapped Snort and Brill together to two AP chips (without co-optimization),with

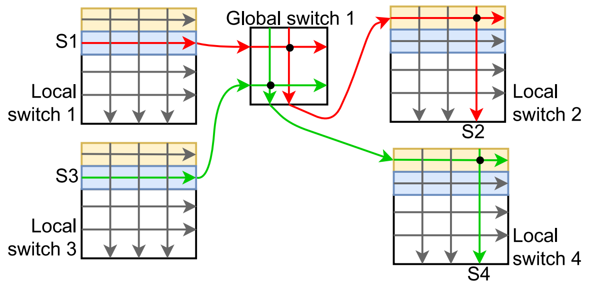

Fig. 8. Example of two applications sharing a global switch. There is no interference between the two groups of paths as indicated by the red and green colors.

the utilization of 242.4 tiles (94.7% of all the tiles in two chips). As the sum of Cache Automaton's reported utilization of Snort and Brill exceeds the capacity of two chips, APmap will most likely offer capacity benefits versus Cache Automaton.

Assist Hardware Design: APmap provides methodologies to solve hardware constraints on the number of input and output signals. As shown in Section IV, APmap can map all the benchmarks to a routing matrix with only four G1 and no G4. It allows the hardware to be more compact, and hence achieving better performance. Especially, G4 is much slower than G1 and STEs [5], which affects the throughput of the whole chip [7]. Therefore, APmap can become a crucial member of a hardware/software co-design toolchain.

VI. Conclusion

In this article, we proposed an open-source tool named APmap for mapping automata to AP chips. It employs multiple optimizations to automate the mapping process and decrease storage utilization. An evaluation with ANMLzoo benchmark suite shows that APmap achieves low overhead and significantly outperforms state-of-the-art.

References

[1] M. Roesch, "Snort—Lightweight intrusion detection for networks," in *Proc. 13th USENIX Conf. Syst. Admin.*, 1999, pp. 229–238.

[2] I. Roy, A. Srivastava, M. Nourian, M. Becchi, and S. Aluru, "High performance pattern matching using the automata processor," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, Chicago, IL, USA, May 2016, pp. 1123–1132.

[3] K. Zhou, J. J. Fox, K. Wang, D. E. Brown, and K. Skadron, "Brill tagging on the micron automata processor," in *Proc. 9th Int. Conf. Semantic Comput.*, Anaheim, CA, USA, 2015, pp. 236–239.

[4] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 12, pp. 3088–3098, Dec. 2014.

[5] A. Subramaniyan, J. Wang, E. R. M. Balasubramanian, D. T. Blaauw, D. Sylvester, and R. Das, "Cache automaton," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchit.*, Boston, MA, USA, Oct. 2017, pp. 259–272.

[6] J. Yu, H. A. D. Nguyen, L. Xie, M. Taouil, and S. Hamdioui, "Memristive devices for computation-in-memory," in *Proc. Design Autom. Test Eur. Conf. Exhibit. (DATE)*, Mar. 2018, pp. 1646–1651.

[7] J. Yu, H. A. D. Nguyen, M. A. Lebdeh, M. Taouil, and S. Hamdioui, "Time-division multiplexing automata processor," in *Proc. Design Autom. Test Eur. Conf. Exhibit. (DATE)*, 2019, pp. 794–799.

[8] K. Angstadt, W. Weimer, and K. Skadron, "RAPID programming of pattern-recognition processors," in *Proc. 21st Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2016, pp. 593–605.

[9] J. Wadden, S. M. Khan, and K. Skadron, "Automata-to-routing: An open-source toolchain for design-space exploration of spatial automata processing architectures," in *Proc. 25th IEEE Annu. Int. Symp. Field Program. Custom Comput. Mach. (FCCM)*, Apr. 2017, pp. 180–187.

[10] T. Xie, V. Dang, J. Wadden, K. Skadron, and M. Stan, "REAPR: Reconfigurable engine for automata processing," in *Proc. 27th Int. Conf. Field Program. Logic Appl. (FPL)*, Sep. 2017, pp. 1–8.

[11] R. Rahimi, E. Sadredini, M. Stan, and K. Skadron, "Grapefruit: An open-source, full-stack, and customizable automata processing on FPGAS," in *Proc. IEEE 28th Annu. Int. Symp. Field Program. Custom Comput. Mach.*, 2020, pp. 138–147.

[12] K. Wang et al., "An Overview of MicronŠs Automata Processor," in *Proc. 11th IEEE/ACM/IFIP Int. Conf. Hardware/Software Codesign and Syst. Synthesis*, Pittsburgh, Pennsylvania, 2016, pp. 14:1–14:3, doi: 10.1145/2968456.2976763.

[13] J. Wadden *et al.*, "VASim: An open virtual automata simulator for automata processing application and architecture research," Dept. Comput. Sci., Univ. Virginia, Charlottesville, VA, USA, Rep. CS2016-03, 2016.

[14] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, pp. 359–392, Dec. 1998.