μ-Genie: A Framework for Memory-Aware Spatial Processor Architecture Co-Design Exploration

Giulio Stramondo*, Manil Dev Gomony†, Bartek Kozicki†, Cees De Laat*, Ana Lucia Varbanescu*
*University of Amsterdam, The Netherlands
{g.stramondo,delaat,a.l.varbanescu}@uva.nl
†Nokia Bell Labs, Belgium
{manil_dev.gomony,bartek.kozicki}@nokia-bell-labs.com

***Index Terms*—spatial processor, memory, framework**

***Abstract*—Spatial processor architectures are essential to meet the increasing demand in performance and energy efficiency of both embedded and high performance computing systems. Due to the growing performance gap between memories and processors, the memory system often determines the overall performance and power consumption in silicon. The interdependency between memory system and spatial processor architectures suggests that they should be co-designed. For the same reason, state-of-the-art design methodologies for processor archiectures are ineffective for spatial processor architectures because they do not include the memory system. In this paper, we present μ-Genie: an automated framework for co-design-space exploration of spatial processor architecture and the memory system, starting from an application description in a high-level programming language. In addition, we propose a spatial processor architecture template that can be configured at design-time for optimal hardware implementation. To demonstrate the effectiveness of our approach, we show a case-study of co-designing a spatial processor using different memory technologies.**

I. Introduction

In modern embedded and high performance computing systems, there is increasing interest in *spatial processors*. These processing architectures consist of physically distributed Processing Elements (PEs), and are more energy efficient than traditional general purpose processors and reconfigurable hardware accelerators [1]–[3]. Despite the growing interest, we are still far from having the right solutions and automated tools for designing efficient, high-performance spatial processors. An important drawback of existing design solutions is their inability to take the memory system into account, despite compelling evidence that the memory system (both on-chip and off-chip) has become a dominant factor affecting the overall performance, power consumption, and silicon area usage [4]–[6]. In this work we argue for *memory-aware design*, because the memory system and the processing elements in a spatial processor architecture are so tightly *interdependent*, they must be *co-designed* to efficiently use the available resources. Co-designing becomes especially important when considering the use of emerging memory technologies, such as MRAM, eDRAM, PCM, or RRAM [7] in spatial processors: they have higher integration density and lower power than SRAM, but also come with additional "quirks" (e.g., MRAM features different read and write latencies).

There are CAD tools [8]–[10] that reduce the increasing design complexity of typical application-specific processors. However, selecting an optimal spatial processor architecture, taking into account the various trade-offs in latency, power consumption, and area usage still requires extensive design-space exploration (DSE) and cannot be performed using the existing CAD tools. In addition, state-of-the-art design flows for application-specific processor DSE focus on processing elements optimization [11]–[13], and do not include the memory system, as illustrated in Figure 1. Instead, co-optimization of the processor and the memory system (including emerging memories) is typically done through the optimization of cache replacement policies [14]–[16].

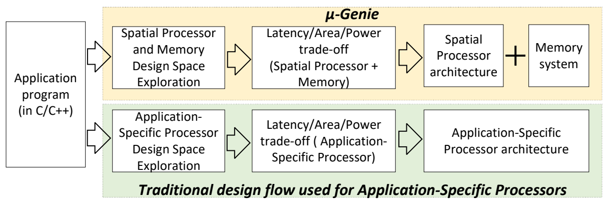
μ-Genie
Application program (in C/C++)
Spatial Processor and Memory Design Space Exploration
Latency/Area/Power trade-off (Spatial Processor + Memory)
Spatial Processor architecture
+
Memory system
Application-Specific Processor Design Space Exploration
Latency/Area/Power trade-off (Application-Specific Processor)
Application-Specific Processor architecture
Traditional design flow used for Application-Specific Processors

Fig. 1: Difference between state-of-the-art design flow typically used for traditional application-specific processors and the proposed μ-Genie design flow for spatial processors.

This paper presents as main contribution μ-Genie (Section II), an automated framework for *memory-aware spatial processor design-space exploration*. The framework allows the user to **customize** the different building blocks to be used in the co-design of the spatial processor and memory system, **explore** different architectures automatically generated for a given application, and **estimate** area, power and latency of each one of the architectures. We highlight as key novel contributions:

1. Unprecedented configuration options for spatial architecture design: memory levels technologies (novel among similar tools), clock frequency (per memory level, also novel), different read/write latencies, and data-widths.

2. A configurable PE architecture template (Section VI) that allows fast prototyping of spatial processor hardware.

3. The Modified Interval Partitioning (MIP) algorithm (Section IV-C), that enables the memory-aware co-Design Space

Exploration (Section V).

To demonstrate the capabilities of μ-Genie, we cover one case-study, showing how a spatial processor can be designed for a representative application and many configurations, including those using MRAM or SRAM for the memory system, can be generated, analyzed, and compared (Section VII).

II. The μ-Genie framework

The μ-Genie framework, illustrated in Figure 2 takes two inputs, the *Configuration Parameters* and an *Application* - described in Section III, and automatically generates a set of hardware architectures, behaviorally equivalent to the input application. The generated hardware architectures can be realized as RTL implementations, using the *architectural templates* described in Section VI. The rest of this section provides an overview of the design and implementation of μ-Genie.

The system architecture we assume in this work has two levels of memory and a spatial processor (Figure 3). Level 1 memory (L1M)[1], the first memory level, and the smaller one in size, uses SRAM as it needs to be physically close to the processor for faster access. The second level - Level 2 memory (L2M)[1] - is larger in size and can be implemented using any memory technology (on-chip or off-chip), even with different access latency for read and write operations. Note that L2M can run at a different clock speed and different IO width than the processor and L1M.

We assume a model of execution following the three steps from Figure 3. Initially, all the required input data for the application are available in L2M. The input data is transferred to the processor, using L1M as intermediate storage (step 1), the data is processed and the results are temporarily stored in L1M (step 2), and, finally, the data from L1M is transferred back to L2M (step 3). The data transfers between L2M and L1M are handled by a Direct Memory Access (DMA) controller. Note that our model of execution performs the steps in a pipelined manner, hence only part of the data will be stored in L1M at any given time.

Because L2M has higher access latency compared to the L1M and spatial processor, we model the L2M assuming its data is accessed in bursts. A read or write burst access to the L2M is controlled by a Direct Memory Access (DMA) controller, with the starting address and size of the burst given as input to the DMA. After an initial *setup latency*, the accessed elements are transferred in sequence from the start address to the end address, from L1M to L2M in case of a write, and from L2M to L1M in case of a read.

III. μ-Genie: Inputs

This section details the two inputs of μ-Genie: the *application* and the *Configuration Parameters*. The applications that can be used as input to μ-Genie are completely defined at compile time, having control-flow instructions not dependent on input data. Such applications allow the static extraction of data dependency information performed by the *Data Dependency Analysis* module (IV-B). We currently support C/C++ applications. However, as the framework uses the LLVM Intermediate Representation [17], it can be easily extended to support other languages as well.

[1]These memories are not to be seen as caches; thus, no cache policies are needed: we schedule data movements at design time. This is why we call them "levels" instead of "layers", and we abbreviate them with L1M and L2M instead of L1 and L2.

The second input to the framework is a configuration file for the different building blocks to be used for hardware architecture realization. Through this file, the user can specify: different compute units (e.g. multipliers, adders), process technology to be used (e.g. 16nm, 28nm), the clock frequency of the processor and L1M, and the clock frequency L2M. Moreover, the user can specify the data-width used by the compute units, L1M and L2M. Information to model the L2M burst accesses is also specified in this file: the setup latency for write/read accesses, the type of L2M to be used (e.g. MRAM, SRAM) and the size of the L2M. The different parameters in the configuration file are then used to access a database containing estimates (obtained by synthesis or from specs) of area usage, static and dynamic power, and latency of each of the building blocks.

IV. μ-Genie: Analysis

This section describes the parts of the framework involved in modeling the data transfers between L2M and L1M, *Model L2 read* and *Model L2 write* (IV-A), the modules that perform data dependence analysis, *Data Dependency Analysis* (IV-B), and the scheduling of the application operations (IV-C).

A. L2 Memory Read and Write Modeling

The first operation performed by the framework is to compute the transfer time of the application's input data from L2M to L1M, implemented in the *Model L2 read* block of Figure 2. Using static analysis, we obtain details regarding the data structures used in the application. For example, in a matrix vector multiplication kernel, the static analysis extracts information about three data structures: an input matrix and an input vector, containing the input elements of the computation, and an output vector containing the results. An address in L2M is given to each input element used by the application; different data structures are placed in consecutive memory addresses. The entire data transfer is modeled as a single burst-read operation from L2M. The information required to compute the arrival clock cycle of each input element to L1M is extracted from the *Configuration Parameters*. Using this information, the exact clock at which each input element arrives in L1M can be computed as seen in (1). The equation symbols are described in Table I.

$$AClk_i = S_r + R_{L2M} * (Add_{L2Mi} + 1) * \frac{B_{L1M}}{B_{L2M}} * \frac{Clk_{L1M}}{Clk_{L2M}} \quad (1)$$

The schedule produced by the Modified Interval Partitioning (MIP), discussed in IV-C, uses the arrival clock cycle computed in this phase to determine when each input element will be available for computation in L1M. The latency of the MIP

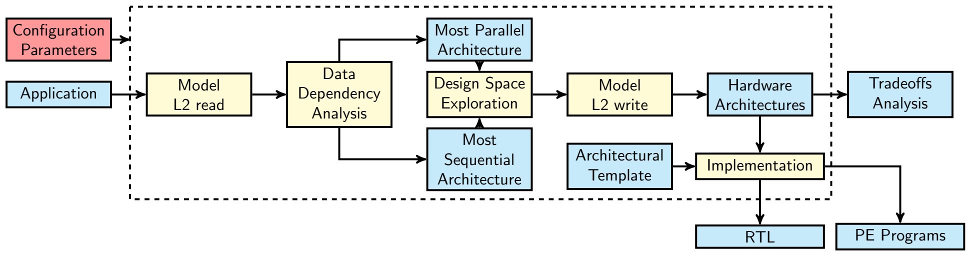

Fig. 2: μ-Genie Framework.

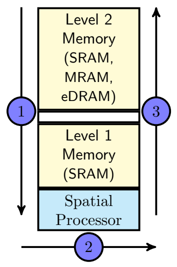

Fig. 3: The system under analysis.

Symbol	Definition
$AClk_i$	Clock at which element i arrives to L1M
S_r	Setup Latency of a L2M burst read
R_{L2M}	L2M read latency (per read), in L2M clock cycles
Add_{L2Mi}	Offset of element i in the burst access
B_{L1M}	Data bitwidth of L1M
B_{L2M}	Data bitwidth of L2M
Clk_{L1M}	Clock Frequency of L1M
Clk_{L2M}	Clock Frequency of L2M
WBL_{L2M}	L2M write burst latency
S_w	Setup Latency of a L2M burst write
W_{L2M}	L2M write latency (per write) in L2M clock cycles
O	Total number of output elements

TABLE I: Definition of symbols used in the equations

schedule includes therefore the L2M→L1M transfer, and the computation; it does not take into account the L1M→L2M transfer of the results (phase 3 in Figure 3). The *Model L2 write* block in Figure 2 computes the latency of the L1M→L2M transfer. The MIP computes the clock cycle at which computation ends (phase 2 in Figure 3) and the last data item is written in L1M. The L1M→L2M transfer can start immediately after the last output is generated. The latency of the L1M→L2M transfer is calculated using (2),

$$WBL_{L2M} = S_w + W_{L2M} * O * \frac{B_{L2M}}{B_{L1M}} * \frac{Clk_{L1M}}{Clk_{L2M}} \qquad (2)$$

where the symbols have been defined in Table I.

B. Data Dependency Analysis

The *Data Dependency Analysis (DDA)* module operates in three stages. The first two stages are the extraction of the *Data Dependency Graph* (DDG) [18] from the application and the schedule of the DDG using the As Soon As Possible (ASAP) and As Late As Possible (ALAP) methodologies. These two steps are core elements in the analysis of high level code for hardware design [19]. Finally, the third step (see IV-C) maps DDG instructions to hardware components - or PEs - using a modified *Interval Partitioning* algorithm [20].

We extract the DGG from the application using some LLVM optimizations and some custom code. We further process the obtained DDG, aiming to reduce the length of the path between the input nodes and the output nodes. This additional transformation is important because the length of these paths is equivalent to the number of sequential operations required to obtain the outputs, which in turn determines the latency of the application. Taking advantage of operation associativity (where possible) we can transform a long sequence of operations - like the one highlighted in Figure 4 - into an equivalent shorter tree.

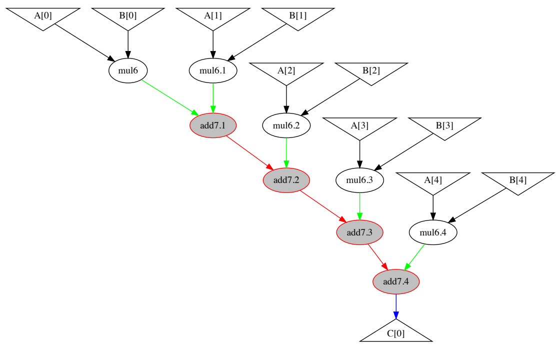

Fig. 4: A Data Dependency Graph: inverse triangles represent input data, obtained from the *load* instructions; ovals describe operations on data; the triangle at the bottom represents the result, derived from a *store* instruction. Highlighted, a chain of associative operations before being optimized by the *DDA* module (IV-B).

Next, we apply the ASAP and ALAP scheduling methodologies [19] to the generated DDG. These schedules will associate to each DDG node a clock cycle where the instruction is executed, and *bound* the design space of possible architectures by determining the *maximally parallel* architectures. We start by scheduling the input nodes of the DDG using the arrival clock time of their input data, computed as explained in Section IV-A, thus taking into account the L2M - L1M transfer time. Next, we determine the minimal latency required to obtain the outputs of the application with the ASAP schedule: starting from the DDG input leafs, each instruction node is scheduled as soon as its dependencies are resolved. Once ASAP is completed, we can perform the ALAP scheduling: starting from the output leaf nodes, each node is scheduled as late as possible according to its dependencies. Once ALAP is completed, every node is annotated with an ASAP clock cycle and an ALAP one. The difference between these two clock cycles, called instruction *mobility*, identifies an interval in which the instruction can be scheduled without changing the overall latency of the application.

The final stage of the *Data Dependency Analysis* module will allocate the DDG nodes to PEs, leveraging the nodes mobility to minimize the number of PEs of the final hardware architecture.

C. PE allocation with Modified Interval Partitioning

To generate a hardware architecture behaviorally equivalent to the input application and with the latency identified during the ASAP-ALAP scheduling, there are two main requirements:

(1) each instruction needs to be computed within its ASAP-ALAP interval, and (2) instructions in the DDG which are executed by the same PE cannot be scheduled at the same time. Our Modified Interval Partitioning (MIP) algorithm - based on the original greedy Interval Partitioning algorithm [20] - is designed to generate, from a DDG, hardware architectures that meet both requirements. The original Interval Partitioning problem addresses the issue of assigning a number of jobs, with known starting and ending time, to the minimum amount of resources, ensuring that the jobs assigned to a resource do not overlap.

To use Interval Partitioning for our problem, we consider instructions as jobs and PEs as resources. There are, however, three main differences between our problem and the canonical Interval Partitioning. First, the original algorithm considers any job can use any resource, while our architecture requires different PEs for different instructions. We therefore perform interval partitioning several times, once for each instruction type (e.g., four times for the graph in Figure 4). This ensures a correct allocation of instructions to PEs performing the same operation. Second, due to *mobility*, instructions do not have a fixed starting time. MIP takes the mobility of an instruction into account by allowing a given instruction to start at any time within its allowed interval. Third, our instructions are dependent on each other, which is not the case for the jobs in the original interval partitioning. To account for this extra constraint, we ensure that any given instruction (a) is only allocated after its dependencies are allocated, and (b) is scheduled to start after the ending time of its dependencies.

D. *Most Parallel and Most Sequential Architectures*

The result of the *DDA* module is the **Most Parallel Architecture (MostPar)**. This architecture takes full advantage of the parallelism of the application and performs the computation with the minimum latency. However, MostPar uses the maximum number of PEs - in the worst case scenario equivalent to the number of instructions in the application - and it will hence have the largest area. At the other end of the spectrum of architectures we can imagine the **Most Sequential Architecture (MostSeq)**, where no parallelism is used and the instruction are scheduled sequentially respecting their dependencies. This architecture will have the worst possible latency, but the minimal impact in area - using only one PE per operation type. Probably none of these two architectures will be of direct interest for the user as they represent two extreme cases. Instead, the interesting architectures are the ones *in between* **MostSeq** and **MostPar**, because they offer interesting trade-offs between power, latency and area. Section V describes how these intermediate architectures can be generated using **MostPar** and **MostSeq**, respectively, as upper and lower bounds of the design space.

V. μ-Genie: Design Space Exploration (DSE)

The *Design Space Exploration* μ-Genie module generates hardware architectures, behaviorally equivalent to the input application, which exhibit area, latency and power tradeoffs. Our DSE is an iterative process which produces, at the end of each iteration, a different hardware architecture. The iterative process starts its sweep from **MostPar**, and ends when **MostSeq** is generated. An iteration consists of three steps. First, the instructions corresponding to output leaf nodes in the DDG are selected. The ALAP schedule of these iterations is increased by 1, and the ALAP scheduling of the rest of the nodes in the DDG is updated accordingly. Consequently, the mobility of each instruction node is increased by one. Finally, the MIP is ran again, using the new ALAP schedule. Due to the increased mobility of each instruction, the generated architecture is likely to use less PEs. The process stops as soon as it generates **MostSeq**, which can be recognized because it contains only one PE per operation type.

Users can tune the DSE process, trading speed for completeness: by increasing the ALAP ”slack” beyond 1, the exploration speeds-up, but less architectures are generated.

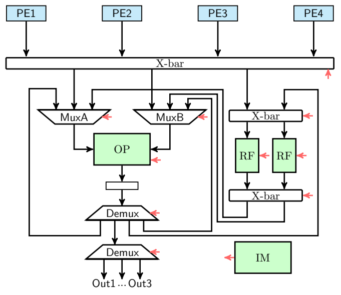

Fig. 5: Functional Unit template. PE1-PE4 represent ”parent” PEs that generate input data. IM is an internal Instruction Memory,where the PE stores the operations to be performed. RFs are internal Register Files, which store reuse data and inputs to be used in the future. OP is the hardware unit actually performing the PE operation.

VI. Architectural Template

To implement the architectures generated by μ-Geniein hardware, we propose a PE template, shown in Figure 5. Each PE has an Instruction Memory (IM) where the operations to be performed at each clock cycle are stored. Each instruction is labeled with the clock cycle in which it should be scheduled. An internal clock counter is compared to the label to decide when to issue the instruction. The internal Register Files (RFs) are used to store input data that needs to be processed in the future, as well as output data that needs to be reused. The *X-bar* in the diagram represent configurable crossbars, which can send data from any input port to any output port. OP is the hardware unit that performs an arithmetic (or logical) operation - e.g. add or multiply. This PE template allows modular implementation of the spatial processor architecture - given that the instructions to execute are stored in the local IM. The inputs to a PE can either be generated by other PEs or the output generated by the same PE in the previous clock cycle. In addition, the inputs to the PE can be used as operands for immediate computation or stored in the RFs for future use.

VII. Case Study

In this section, we demonstrate the capabilities of μ-Genie using one case study. To do so, we analyze the architectures generated by μ-Genie and the energy consumption and

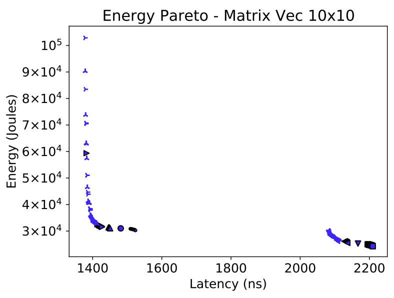

Fig. 6: Each point represents one μ-Genie spatial processor. Different shapes identify different input configurations. We only include the Pareto-optimal designs.

latency (or execution time) of each design. We selected one representative application: matrix-vector (MV) multiplication with matrix sizes 10×10. We used the TSMC 28nm target technology library for generating the database containing the area usage and energy consumption of the different building blocks, as required by our framework. We generated multiple input *Configuration Parameters* to let μ-Genie explore the parameter space, and compute the latency, area usage, and energy consumption of the architectures. Each generated architecture has a known latency imposed in each iteration of the DSE (Section V), which we use to compute its static energy consumption. Moreover, after applying the Modified Interval Partitioning algorithm, the instructions performed by each PE are known and this information is used to compute the dynamic energy consumption of an architecture.

We compare the use of MRAM - modeled according to [21] - and SRAM for L2M (both in 28nm), thus showing the impact of choosing different memory technologies in the L2M.

The comparison results are presented in Figure 6. In the graph, each point represents a hardware architecture generated by μ-Genie; moreover, shapes identify different input configurations. Specifically, we compare a total of 18 configurations: 2 L2M technologies, MRAM and SRAM, clocked at 350MHz, and 9 different clock frequencies (400MHz - 1GHz, in steps of 200) for the processor and L1M ensemble.

We can identify two clusters: LL-HE (low-latency, high-energy) and HL-LE (high-latency, low-energy). Each cluster belongs to one memory technology: LL-HE contains all SRAM designs, while HL-LE contains all MRAM designs. For the considered application (Fig 6) the fastest architecture - using SRAM memory - has a latency of 1375ns and consumes over 1×10^5Joules. The most energy efficient SRAM architecture has instead a latency of 1525ns and consumes under 0.3×10^5Joules, thus being 3x more energy efficient than the fastest, with a latency increase of only 10%. The most energy efficient architecture using MRAM technology has instead a latency of 2210ns and consumes 0.24×10^5Joules, hence having 45% higher latency than the best SRAM counterpart, with 25% lower energy consumption.

VIII. Conclusion and Future Work

In this work, we presented μ-Genie, a novel framework for design-space exploration of memory-aware spatial processors. We have empirically demonstrated that different spatial processor architectures can be quickly implemented using our novel PE hardware template. Lastly, we have shown the capabilities of the framework using case studies, which illustrates the sanity of our DSE approach and its ability to facilitate a comparison between the use of MRAM and SRAM technologies. For example, we were able to conclude that, for a matrix vector multiplication 10x10, the most energy efficient architecture generated with μ-Genie with MRAM L2M has 45% higher latency than the best SRAM counterpart, with 25% decrease in power consumption.

Our future work focuses on the use μ-Genie to analyze more applications. In addition, we plan to enhance our framework adding the capability of automatically *merging* multiple spatial processor architectures, to generate a single *multi-application spatial processor*.

References

[1] A. Parashar *et al.*, "Efficient spatial processing element control via triggered instructions," *IEEE Micro*, vol. 34, no. 3, pp. 120–137, 2014.
[2] S. Smets *et al.*, "2.2 a 978gops/w flexible streaming processor for real-time image processing applications in 22nm fdsoi," in *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, Feb 2019, pp. 44–46.
[3] Y. Chen *et al.*, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE JETCAS*, June 2019.
[4] S. Williams *et al.*, "Roofline: An insightful visual performance model for floating-point programs and multicore architectures," Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), Tech. Rep., 2009.
[5] M. Dayarathna *et al.*, "Data center energy consumption modeling: A survey," *IEEE Commun. Surv. Tutor.*, vol. 18, no. 1, pp. 732–794, 2015.
[6] T. Oh *et al.*, "An analytical model to study optimal area breakdown between cores and caches in a chip multiprocessor," in *2009 IEEE ISVLSI*. IEEE, 2009, pp. 181–186.
[7] A. Chen, "A review of emerging non-volatile memory (nvm) technologies and applications," *Solid-State Electronics*, vol. 125, 07 2016.
[8] Synopsys Inc., "Synopsys IP designer," in *https://www.synopsys.com/dw/ipdir.php?ds=asip-designer*, 2019.
[9] Cadence Design Systems, Inc., "Tensilica customizable processors," in *https://ip.cadence.com/ipportfolio/tensilica-ip/xtensa-customizable*.
[10] Codasip Ltd., "Codasip Studio," in *https://www.codasip.com/custom-processor/*, 2019.
[11] P. Meloni *et al.*, "Enabling Fast ASIP Design Space Exploration: An FPGA-based Runtime Reconfigurable Prototyper," *VLSI Des.*, Jan. 2012.
[12] J. Eusse *et al.*, "Pre-architectural performance estimation for ASIP design based on abstract processor models," in *SAMOS XIV*, July 2014.
[13] L. Jozwiak *et al.*, "ASAM: Automatic architecture synthesis and application mapping," *Microprocessors and Microsystems*, vol. 37, no. 8 PARTC, 2013.
[14] G. Sun *et al.*, "A novel architecture of the 3D stacked MRAM L2 cache for CMPs," in *2009 IEEE HPCA*, Feb 2009.
[15] M. P. Komalan *et al.*, "System level exploration of a stt-mram based level 1 data-cache," in *2015 DATE*, March 2015.
[16] S. Lee *et al.*, "Hybrid cache architecture replacing sram cache with future memory technology," in *2012 IEEE ISCAS*, May 2012.
[17] C. Lattner *et al.*, "Llvm: A compilation framework for lifelong program analysis & transformation." IEEE Computer Society, 2004.
[18] S. Isoda *et al.*, "Global compaction of horizontal microprograms based on the generalized data dependency graph," *IEEE Trans. Comput.*, no. 10, 1983.
[19] C.-T. Hwang *et al.*, "A formal approach to the scheduling problem in high level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, no. 4, pp. 464–475, 1991.
[20] D. Mount. (2017) Greedy algorithms for scheduling. [Online]. Available: http://www.cs.umd.edu/class/fall2017/cmsc451-0101/Lects/lect07-greedy-sched.pdf
[21] Q. Dong *et al.*, "A 1Mb 28nm STT-MRAM with 2.8ns read access time at 1.2V VDD using single-cap offset-cancelled sense amplifier and in-situ self-write-termination," in *2018 IEEE ISSCC*, Feb 2018.