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

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 as 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 a increasing interest in *spatial processors*, processing architectures consisting of physically distributed Processing Elements (PEs), as they are more energy efficient traditional general purpose processors and reconfigurable hardware accelerators. [1]–[7]. 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 [8]–[10]. In this work we argue that, given that the memory system and the processing elements in a spatial processor architecture are so tightly *interdependent*, they must be *co-designed* to efficently use the available resources. A memory-aware design produces spatial architectures having bandwitdth close to the bandwidth of the memory system, effectively reducing the instantiation of unrequired resources. Co-designing becomes especially important when considering the use of emerging memory technologies, such as MRAM, eDRAM, PCM, or RRAM [11] 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 [12]–[14] 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 [15]–[18], 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 [19]–[22].

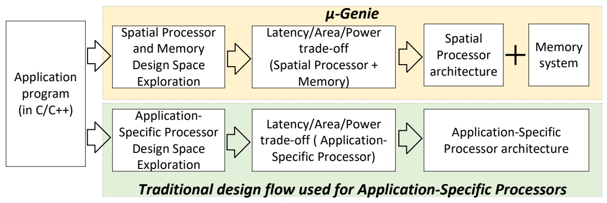

μ-Genie
Application program (in C/C++)
Spatial Processor and Memory Design Space Exploration
Latency/Area/Power trade-off (Spatial Processor + Memory)
Spatial Processor architecture
+
Memory system
Application-Specific Processor Design Space Exploration
Latency/Area/Power trade-off (Application-Specific 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 III), an automated framework for *memory-aware spatial processor design-space exploration*. The framework presented in this work 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 novelties:

1. Unprecedented configuration options: 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 VII) that allows fast prototyping of spatial processor hardware.

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

To demonstrate the capabilities of μ-Genie, we cover three case-studies, showing how a spatial processor can be designed for two different applications and many configurations, including those using MRAM or SRAM for the memory system, can be generated, analysed, and compared (Section VIII).

II. Background

A *spatial processor* architecture consists of a set of physically distributed PEs with dedicated control units interconnected using an on-chip interconnect. The operations that need

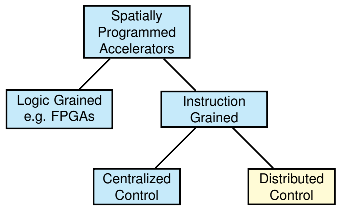


Fig. 2: Spatial Architectures classification [1].

to be performed by an algorithm are mapped on the PEs, which compute in a fine-grained pipeline fashion. There are different kinds of spatial architectures, one possible classification is shown in Figure 2 [1]. FPGAs are an example of spatially programmed architectecture in which the PEs implement basic logic operations, and hence are classified as Logic-Grained. To change the functionality of a Logic-Grained architecture, the hardware design needs to be modified and re-synthesized. Instruction-Grained spatial architecture are instead programmable at instruction level and their PEs implement simplified ALUs. The functionality of a *Instruction-Grained* spatial accelerator can change by modifying the sequence of instructions it executes. The advantage of using *Instruction-Grained* over *Logic-Grained* programmable architectures lies in their higher computational density, which results in a higher operational frequency and lower power consumption [1]. The Instruction-Grained class is itself composed of architecture having Centralized Control [23] , where a single control unit manages all the PEs, and Distributed Control, where each PE has a built-in control mechanism [1], [2], [5]. Intuitively, an architecture with distributed control is more scalable and has a simpler interconnection network. In this work we introduce μ-Genie, an automated framework that *generates distributed control spatial architectures optimized for input applications*.

III. The μ-Genie framework

The μ-Genie framework, illustrated in Figure 3 takes two inputs, the *Configuration Parameters* - described in Section IV-B - and an *Application* - detailed in Section IV-A, 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 VII. The rest of this section provides a detailed analysis of the design and implementation of μ-Genie.

A. Model of Execution

The system architecture we assume in this work has two levels of memory and a spatial processor (Figure 4). 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 4, shown by arrows representing the direction of data flow. Initially, all of 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 transfer 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.

μ-Genie lets the user specify the parameters of the L2M through the *Configuration Parameters*. The L2M parameters are used to model the data transfer between L2M and L1M (see III-B and V-A). The L2M model is used to compute arrival time of input elements in the L1M. The arrival time of the element in the L1M is then used by the Modified Interval Partitioning (MIP) algorithm - see V-C, to produce spatial architectures having bandwitdth close to the bandwidth of the L2M. This effectively reduces the instantiation of unrequired resources in both the L1M and the spatial procesor. The L1M is composed of multiple banks having different depths. The number and depths of the banks composing the L1M is determined by the MIP as described in VI-A.

B. The L2 Memory Model

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.

IV. μ-Genie: Inputs

The framework takes two inputs, an *application* (IV-A) and *Configuration Parameters* (IV-B).

A. Application

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 enable the static extraction of data dependency information performed by the *Data Dependency Analisys* module (V-B). We currently support C/C++ applications. However, as the framework uses the LLVM Intermediate Representation [24], it can be easily extended to support other languages as well.

B. Configuration Parameters

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

[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.

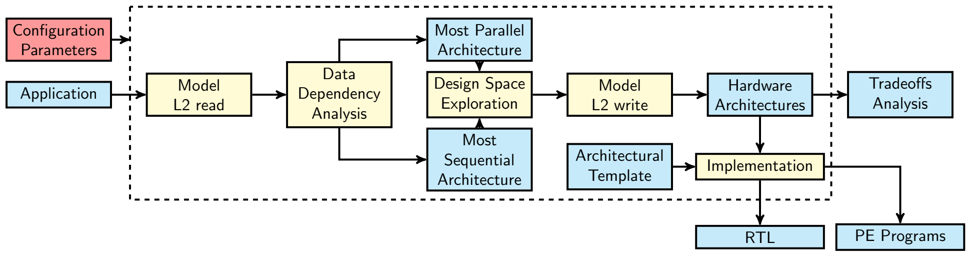


Fig. 3: μ-Genie Framework.

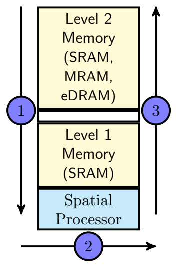


Fig. 4: The system under analysis.

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. Our L1M implementation uses multiple memory banks of different sizes (see VI-A). To estimate the resource usage of the different types of these memories we built a linear model, using synthesis data. We compared the ability of our linear model to predict area, latency and power consumption against the data generated using the synthesis tool and we found our linear model to be accurate.

V. μ-GENIE: ANALYSIS

This section describes the parts of the framework involved in modeling of the data transfers between L2M and L1M, *Model L2 read* and *Model L2 write* (V-A),the modules that perform data dependence analysis, *Data Dependency Analysis*(V-B), and how the operations of the input application are scheduled (V-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 3. 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 output elements of the computation. 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.

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

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

The schedule produced by the Modified Inteval Partitioning (MIP), discussed in V-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 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 4). The *Model L2 write* block in Figure 3 computes the latency of the L1M→L2M transfer. The MIP computes the clock cycle at which computation ends (phase 2 in Figure 4) 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),

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

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) [25] 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 [26]. Finally, the third step (see V-C) maps DDG instructions to hardware components - or PEs - using a modified *Interval Partitioning* algorithm [27].

To extract the DDG from an application, we use LLVM and custom transformations. We first convert the input application code to its LLVM Intermediate Representation. We then transform the code into static single assignment (SSA) form and perform full-loop unrolling on all of the application loops. After these transformations, there will be no control flow instructions in the application body, and each variable will be defined only once. It is now possible to follow the definition

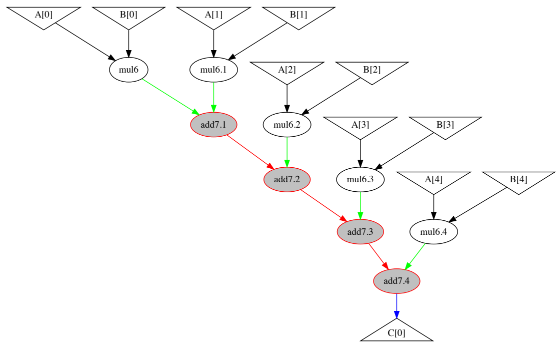


Fig. 5: A Data Dependency Graph: inverse triangles represent input data, obtained from the *load* instructions; ovals describe operations on data; the triangle at the bottom represents the result, derived from a *store* instruction. Highlighted, a chain of associative operations before being optimized by the *DDA* module (V-B).

and use chain of the variables to produce a Data Dependency Graph like the one shown in Figure 5. The DDG represents each operation as a node - in Figure 5 the input and output nodes represent respectively load and store instructions, while oval nodes represent computations - and each edge represent a dependency between operations.

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 5 - into an equivalent shorter tree.

Once we have the DDG, we apply the ASAP and ALAP scheduling methodologies to the generated DDG. These schedules will associate to each DDG node a clock cycle where the instruction is executed, and *bound* the 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 element, computed as explained in Section V-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 - described in Section V-C - 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 original 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 [27] - is designed to generate, from a DDG, hardware architectures that meet both requirements. Listing 1 presents MIP, in pseudo-code. 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.

1) 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 (lines 5-20), once for each instruction type (e.g., four times for the graph in Fig. 4). This ensures a correct allocation of instructions to PEs performing the same operation.
2) 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 (lines 11-13).
3) and final difference, is that 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 (1) is only allocated after its dependencies are allocated (line 4), and (2) is scheduled to start after the ending time of its dependencies (line 7-8).

```
1  # ASAP[i] and ALAP[i] contain the scheduled cycles for instruction i
2  # SetPEs is the set of Processing Elements in the architecture
3  SetPEs=[]
4  sort instructions by ASAP[i]
5  for each instruction i
6    allocated = False
7    dep_deadline = maximum end-time of all instructions depending on i
8    schedule[i] = max(ASAP[i], dep_deadline)
9    for each PE in SetPEs
10     if instruction i matches PE
11       if ALAP[i] >= next_free_slot[PE]
12         add instruction i to PE
13         schedule[i] = max(schedule[i], next_free_slot[PE])
14         next_free_slot[PE] = schedule[i] + latency(i)
15         allocated = True
16   if not allocated
17     create new PE with type(i)
18     add instruction i to PE
19     next_free_slot[PE] = schedule[i] + latency(i)
20     add PE to setPEs
```

Listing 1: Modified Interval Partitioning (MIP) Algorithm

The MIP algorithm returns `setPEs` and a `schedule` for the current design: `setPEs` is the list of processing elements that form the architecture, with each PE containing the instructions it has to execute, while `schedule` contains the clock cycle at which each instruction is scheduled to be executed.

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, the 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 worst 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, however 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 VI describes how these intermediate architectures can be generated using **MostPar** and **MostSeq** respectively as upper and lower bounds of the design space.

VI. 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 - described in Listing 2 - is an iterative process which produces, at the end of each iteration, a different hardware architecture. The iterative process start its sweep from the **MostPar**, and ends when the **MostSeq** is generated.

```
1  currentArchitecture = MostPar
2  found_MostSeq=False
3  GeneratedArchitectures=[]
4  while(!found_MostSeq)
5      type_count={}
6      found_MostSeq=True
7      for each PE in currentArchitecture
8          type_count[type(PE)]+=1
9              if type_count[type(PE)] > 1
10                 found_MostSeq=False
11                 break
12     if found_MostSeq
13         break
14     for each instuction i in DDG
15         if type(i) == 'store'
16             ALAP[i] = ALAP[i]+1
17     ALAP=performALAPschedule(instructions ,dependencies)
18     SetPEs=MIP(instructions ,ASAP,ALAP)
19     GeneratedArchitectures+=[SetPEs]
20     currentArchitecture=SetPEs
```

Listing 2: Design Space Exploration

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 (lines 14-16), and the ALAP scheduling of the rest of the nodes in the DDG is updated accordingly (line 17). Consequently, the mobility of each instruction node is increased by one. Finally, the MIP is ran again (line 18), 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 one iteration generates the **MostSeq**, which can be recognized because it contains only one PE per operation type (lines 6-13).

There are two side benefits of our DSE approach. First, the user can tune the granularity of the exploration: by increasing the ALAP "slack" beyond 1, the exploration speeds-up, but less architectures are generated. Second, the DSE process can be easily parallelised, because its iterations are independent.

A. Architecture Tradeoffs

For a given input application and a given configuration, μ-Genie outputs a set of hardware architectures - composed of spatial processor and L1M - with different area and latency trade-offs. Figure 6 shows three of the architectures generated during the DSE. Each box represents a PE. The different load and store PEs are implemented as separate L1M banks. As an example, the architecture in Figure 6b has 1 L1M bank to store the input data and 5 banks to store the results. This architecture can therefore receive 1 input element per clock cycle from the L2M, and store up to 5 results per clock in the L1M store banks. The remaining PEs are obtained from computing instructions. We allow the architectures to have cycles because the circuit is synchronous, and every instruction has been carefully scheduled. A self-loop in a PE indicates data reuse (see Section VII for implementation details).

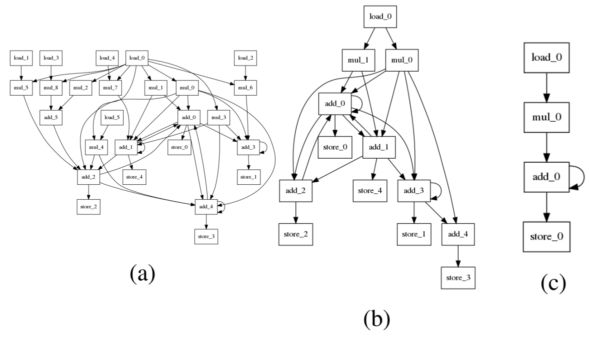


Fig. 6: Example of architectures generated from a matrix vector multiply application of size 5x5. The MostPar (a), an intermediate architecture (b) and the MostSeq (c).

B. Multi-Configuration Design Space Exploration

μ-Genie can also perform design space exploration on the configuration parameters (Section IV-B), generating one configuration file for each combination of configuration parameters and aggregating the architectural trade-offs. This allows, as an example, to estimate the effect that different clock frequencies or different memory technology have on the area,latency and energy trade-offs. The use cases discussed in VIII-B and VIII-C are examples of multi-configuration DSEs.

VII. ARCHITECTURAL TEMPLATE

To implement in hardware the architectures generated by μ-Genie, we propose a PE template, shown in Figure 7. 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 white rectangles in the diagram represent configurable crossbars,

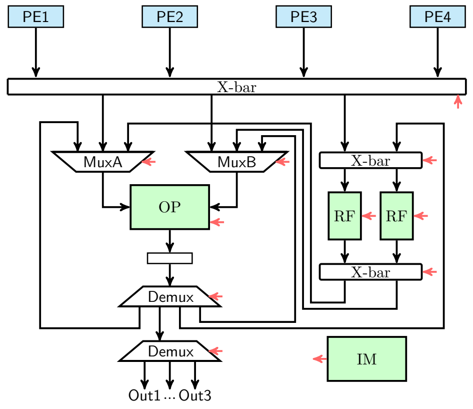


Fig. 7: Functional Unit template. PE1-PE4 represent "parent" PEs that generate input data. IM is an internal Instruction Memory,where the PE stores the operations to be performed. RFs are internal Register Files, which store reuse data and inputs to be used in the future. OP is the hardware unit actually performing the PE operation.

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 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. We have implemented the template PE in RTL that can be configured to build all types of PEs found in the architectures generated by μ-Genie.

VIII. Case Studies

In this section, we demonstrate the capabilities of μ-Genie using three case studies. To do so, we analyze the architectures generated by μ-Genie and the energy consumption and latency (or execution time) of each design. We selected two representative applications: matrix-vector (MV) and matrix-matrix (MM) multiplication with matrix sizes 5×5, 10×10, and 15×15. 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* - Section IV-B - to let μ-Genie explore the parameter space (see VI-B), 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 VI), which we use to compute its static energy consumption. Moreover, after applying the Modified Interval Partitioning algorithm (see Section V-C), the instructions performed by each PE are known and this information is used to compute the dynamic energy consumption of an architecture.

In the first case study - Section VIII-A - we illustrate how DSE works for 5x5 MV and a single configuration - i.e. one configuration parameter file, see IV-B. The second case study compares the use of MRAM - modeled according to [28] - and SRAM for L2M (both in 28nm) for the two applications, that shows the impact of choosing different memory technologies in the L2. The last case study compares μ-Genie architectures for the different MV sizes.

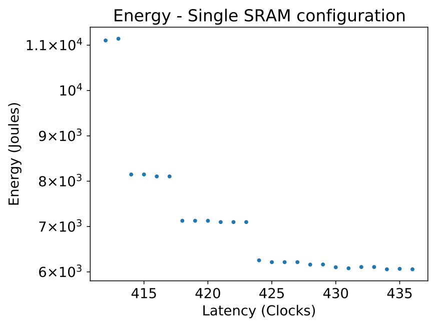
A. Single configuration DSE

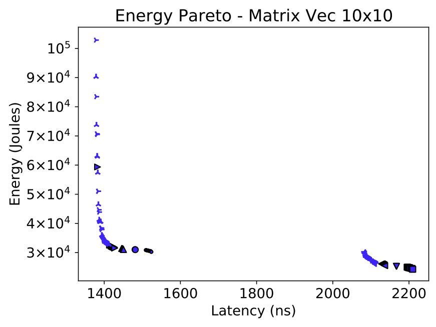
The goal of this case-study is to illustrate the ability of μ-Genie to generate, given a single configuration, architectures with different energy consumption. We selected a configuration that uses SRAM in both levels. L2M is clocked at 350MHz, while L1M and the spatial processor are clocked at 1GHz. Figure 7a shows the energy consumption of 30 different pareto-optimal spatial processor architectures, with different latency and energy consumption. We make two observations: (1) the latency and energy consumption of each design range between the min and max latency, as given by the **MostPar** and **MostSeq** architectures, and (2) as expected, faster designs result in higher energy consumption, due to their larger numbers of PEs.

B. MRAM vs SRAM Level 2 Memory

In this case study we compare the energy efficiency of two alternative technologies to implement L2: MRAM and SRAM. The comparison is performed using both applications - MV and MM - with 10×10 matrices (see Fig 8b and 8c, respectively). In both graphs, each point is relative to 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.

In both figures 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 containts all MRAM designs. For the MV application (Fig 8b) 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 10% increase in latency. 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. The matrix multiplication, Figure 8c, performs 10 times more operation than the matrix vector multiplication, hence there is a clear overall increase in latency - about 30% - and energy consumption - about 4 times - in comparison to the previous application. In this case the SRAM architecture consuming the least amount of energy has 2% higher latency than the fastest SRAM architecture, but consumes 50% less energy. However, the introduction of MRAM technology in the L2M is not as beneficial as it was for the matrix vector application. The MRAM architecture consuming the least amount of energy has a 3% slowdown compared to the most energy efficient SRAM, while attaining only a 2.2% improvement in energy consumption.


(a) 5x5 MM


(b) MV 10x10

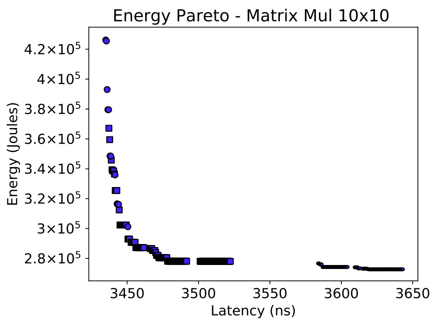

(c) MM 10x10

Fig. 8: Each point represents one μ-Genie spatial processor. Different shapes (in 8b and 8c) identify different input configurations. 8a shows the architecture's Energy over Latency in clock cycles generated from a single configuration of a matrix vector multiplication of size 5×5. Note that (a) presents all designs, while (b) and (c) only include the Pareto-optimal designs.

C. Different Matrix Dimensions

In this case study we compare three different matrix sizes for MV - 5×5, 10×10 and 15×15, using the same configurations used in VIII-B, to evaluate how the energy consumption of an L2M MRAM scale with respect to an L2M SRAM. Figure 9 shows the Pareto-optimal architectures generated from each input application. The increase in the matrix size is reflected by an increase in latency: all MV-5×5 application-specific processors have latency below 1000ns, the MV-10×10 have latency between 1250ns and 2500ns, and the latency of all MV-15×15 is beyond 2500ns. However, the increased number of operations results instead in wider trade-offs possibilities. Thus, the normalized latency gap between the most energy efficient SRAM and MRAM, decreases with the matrix size, from 82% for the 5×5, to 42% for the 10x10 and even 32% for the 15×15. The reduction in energy consumption between the same pair of results is instead 20% for the 5×5 and 10×10, while for the 15×15 drops to 16%. Therefore, as the size of the matrix grows, the benefits in energy consumption diminish when using MRAM technology in the L2M. This is a behaviour caused by the increased number of write operations that have high energy impact when the MRAM technology is used.

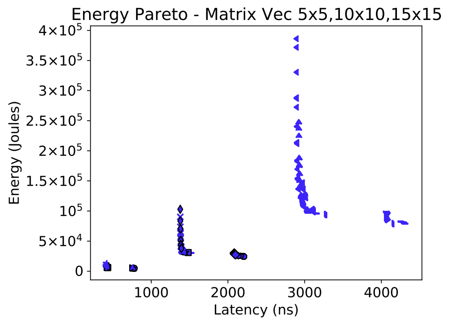


Fig. 9: Energy Pareto optimal architectures generated by μ-Genie for different sizes of Matrix Vector multiplication 5x5 - with latency ranging from 0 to 1000,10x10 - having latency between 1000ns and 2500ns , and 15x15- with latency above 2500ns. Each point corresponds to an architecture generated by the framework.

IX. Related Work

Previous work on design of spatial processor focuses on the hardware architecture of the processor, while the optimization of the memory system is only partially taken into account. In [1] a spatial processor with distributed control across PE using triggered instructions is presented. Their architecture is built around the guarded-action programming paradigm, where guards - boolean expressions specifying if an action is legal - are evaluated by a scheduler and trigger computations. Support for high level languages is missing, so this spatial processor needs to be programmed in a low level guarded-action language and the computation needs to be manually mapped on the PEs. Their memory system consist of two levels of memories (L1 and L2) and distributed scratchpad memories located within the PEs. The design is not tailored for a specific set of applications and they do not perform analysis on the interactions between the memory and processing systems, leaving the modeling of the memory system as future work.

Plasticine, a spatial processor optimized for the acceleration of parallel patterns is presented in [2]. Their memory system is composed of Pattern Memory Units (PMUs) which are connected through a network to Pattern Compute Units (PCUs). Although it allows some degree of configurations, Plasticine is not meant to be optimized around specific applications. The input application needs to be written in a language exposing its parallel patterns - Delite Hardware Definition Language (DHDL) - and then it is mapped on the Plasticine architecture. Hence, the number of memory units (PMUs) and processing units (PCUs), and their interconnections are not optimized around specific applications.

In [3], a framework to generate Application Specific Hardware (ASH) from a C application is presented. The final architecture it produces is asyncrounous, and operation dependencies are handled using a token-based mechanism which is implemented in hardware. The memory system of the architecture consists of a monolitic memory. To handle concurrent memory requests the design uses a hierarchy of busses and arbiters, which creates a bottleneck. This means that their memory system is overwhelmed because it is not tailored for the PEs it uses.

Spatially distributed PEs with a dedicated configuration register allow to configure the PEs to one of the operating modes [4] at compile time. Few PEs are connected in back-to-back, forming systolic arrays which are then interconnected using an on-chip interconnect. Hence, the processor architecture is quite general-purpose, i.e, the interconnect allows a PE array to be connected to any another PE array. However, there is no automated design flow to efficiently map algorithms to the processor architecture.

An intresting approach is Catena [5], an ultralow-power spatial processor with a distributed architecture, where multiple techniques - *clock gating*, *power gating* and *voltage boosting* - are applied in a fine-grained way to optimize energy efficiency. These techniques can be used to explore the power/latency tradeoff of specific applications. However, the impact of the memory system on the performace of the design is not modeled and the memory system is not co-designed with the spatial processor, potentially resulting in an inefficient utilization of the hardware resources; moreover, Catena lacks high-level language support.

In summary, a comparison of μ-Genie against existing work is presented in Table II.

Framework	Type (see II)	Application Optimized	Memory Co-Design	Architectural DSE	High Level Language
μ-Genie	Dist. Control	Yes	Yes	Yes	Yes, C
[1]	Dist. Control	No	No	No	No
[2]	Dist. Control	No	No	No	Yes, DHDL
[4]	Dist. Control	No	No	No	No
[3]	Logic Grained	Yes	No	No	Yes, C
[5]	Dist. Control	Yes	No	Yes	No

TABLE II: Comparison with related work.

X. Conclusion and Future Work

In this work, we presented μ-Genie, a novel framework for co-designing memory-aware custom processors. The framework enables design-space exploration of spatial processor archiectures including the memory system. 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 three case studies, which illustrate 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 our automated framework to analyze multiple 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] R. Prabhakar *et al.*, "Plasticine: A reconfigurable architecture for parallel patterns," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 389–402.

[3] M. Budiu *et al.*, "Spatial computation," in *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, 2004, pp. 14–26.

[4] 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.

[5] J. P. Cerqueira *et al.*, "Catena: A near-threshold, sub-0.4-mw, 16-core programmable spatial array accelerator for the ultralow-power mobile and embedded internet of things," *IEEE Journal of Solid-State Circuits*, 2020.

[6] Z. Du *et al.*, "ShiDianNao: Shifting vision processing closer to the sensor," in *2015 ACM/IEEE 42nd ISCA*, June 2015.

[7] Y. Chen *et al.*, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE JETCAS*, June 2019.

[8] 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.

[9] M. Dayarathna *et al.*, "Data center energy consumption modeling: A survey," *IEEE Commun. Surv. Tutor.*, vol. 18, no. 1, pp. 732–794, 2015.

[10] 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.

[11] A. Chen, "A review of emerging non-volatile memory (nvm) technologies and applications," *Solid-State Electronics*, vol. 125, 07 2016.

[12] Synopsys Inc., "Synopsys IP designer," in *https://www.synopsys.com/dw/ipdir.php?ds=asip-designer*, 2019.

[13] Cadence Design Systems, Inc., "Tensilica customizable processors," in *https://ip.cadence.com/ipportfolio/tensilica-ip/xtensa-customizable*.

[14] Codasip Ltd., "Codasip Studio," in *https://www.codasip.com/custom-processor/*, 2019.

[15] P. Meloni *et al.*, "Enabling Fast ASIP Design Space Exploration: An FPGA-based Runtime Reconfigurable Prototyper," *VLSI Des.*, Jan. 2012.

[16] J. Eusse *et al.*, "Pre-architectural performance estimation for ASIP design based on abstract processor models," in *SAMOS XIV*, July 2014.

[17] L. Jozwiak *et al.*, "ASAM: Automatic architecture synthesis and application mapping," *Microprocessors and Microsystems*, vol. 37, no. 8 PARTC, 2013.

[18] K. Karuri *et al.*, "A Generic Design Flow for Application Specific Processor Customization through Instruction-Set Extensions (ISEs)," in *SAMOS*, K. Bertels *et al.*, Eds. Springer Berlin Heidelberg, 2009.

[19] G. Sun *et al.*, "A novel architecture of the 3D stacked MRAM L2 cache for CMPs," in *2009 IEEE HPCA*, Feb 2009.

[20] M. P. Komalan *et al.*, "System level exploration of a stt-mram based level 1 data-cache," in *2015 DATE*, March 2015.

[21] S. Lee *et al.*, "Hybrid cache architecture replacing sram cache with future memory technology," in *2012 IEEE ISCAS*, May 2012.

[22] S. Mittal, "A technique for efficiently managing SRAM-NVM hybrid cache," *CoRR*, vol. abs/1311.0170, 2013. [Online]. Available: http://arxiv.org/abs/1311.0170

[23] S. Swanson *et al.*, "The wavescalar architecture," *ACM Transactions on Computer Systems (TOCS)*, vol. 25, no. 2, pp. 1–54, 2007.

[24] C. Lattner *et al.*, "Llvm: A compilation framework for lifelong program analysis & transformation." IEEE Computer Society, 2004.

[25] S. Isoda *et al.*, "Global compaction of horizontal microprograms based on the generalized data dependency graph," *IEEE Trans. Comput.*, no. 10, 1983.

[26] 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.

[27] D. Mount. (2017) Greedy algorithms for scheduling. [Online]. Available: http://www.cs.umd.edu/class/fall2017/cmsc451-0101/Lects/lect07-greedy-sched.pdf

[28] 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.