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 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 consisting of physically distributed Processing Elements (PEs), and are more energy efficient than 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 efficiently use the available resources. A memory-aware design produces spatial architectures having bandwidth 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].

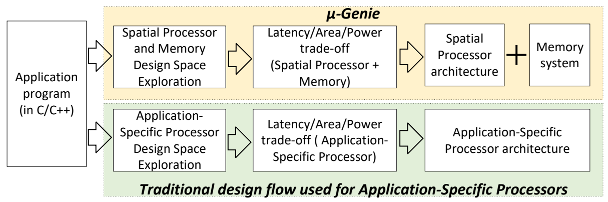

μ-Genie
Application program (in C/C++)
Spatial Processor and Memory Design Space Exploration
Latency/Area/Power trade-off (Spatial Processor + Memory)
Spatial Processor architecture
+
Memory system
Application-Specific Processor Design Space Exploration
Latency/Area/Power trade-off (Application-Specific 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 novel contributions:

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, analyzed, 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

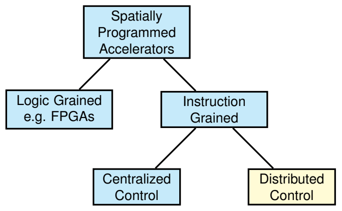


Fig. 2: Spatial Architectures classification [1].

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 architectures 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 architectures are instead programmable at instruction level and their PEs implement simplified ALUs. The functionality of an *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 architectures 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 28 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 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 bandwidth close to the bandwidth of the L2M. This effectively reduces the instantiation of unrequired resources in both the L1M and the spatial processor. 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

This section details the two inputs of μ-Genie: the *application* and the *Configuration Parameters*.

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 Analysis* 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.

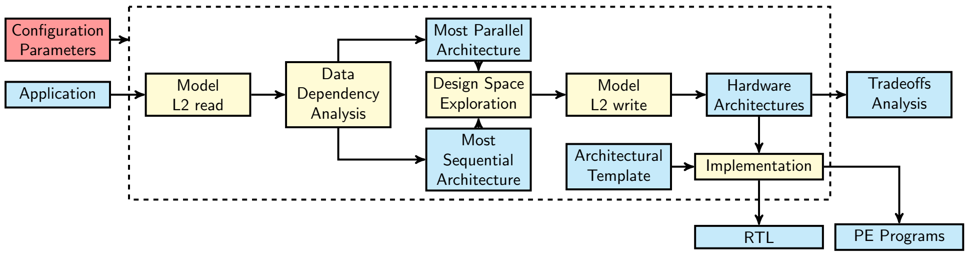


Fig. 3: μ-Genie Framework.

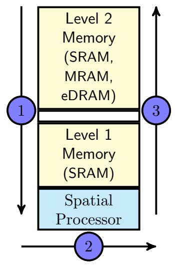


Fig. 4: The system under analysis.

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 it to be accurate - less then 2% error in the area and static energy model and less than 28% error in the dynamic energy model.

V. 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* (V-A),the modules that perform data dependence analysis, *Data Dependency Analysis*(V-B), and the scheduling of the application operations (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 28. 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 Interval 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 28 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

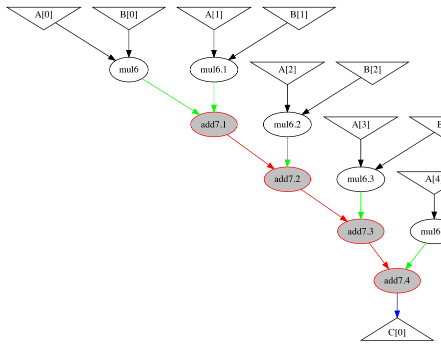


Fig. 5: A Data Dependency Graph: inverse triangles represent input data, obtained from the *load* instructions; ovals describe operations on data; the triangle at the bottom represents the result, derived from a *store* instruction. Highlighted, a chain of associative operations before being optimized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

Next, we apply the ASAP and ALAP scheduling methodologies [26] 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 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 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 [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 Figure 5). 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) 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 (line 4), and (b) 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, 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 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 starts its sweep from **MostPar**, and ends when **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 **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 parallelized, 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).

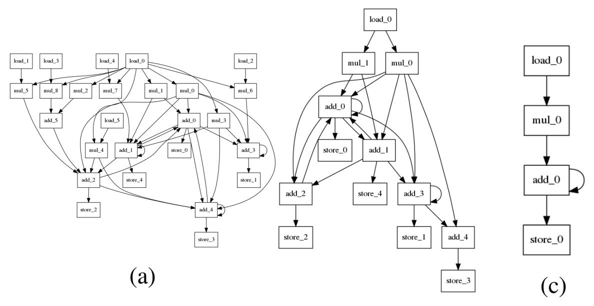


Fig. 6: Example of architectures generated from a matrix vector multiply application of size 5x5. The MostPar (a), an intermediate architecture (b) and the MostSeq (c).

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,

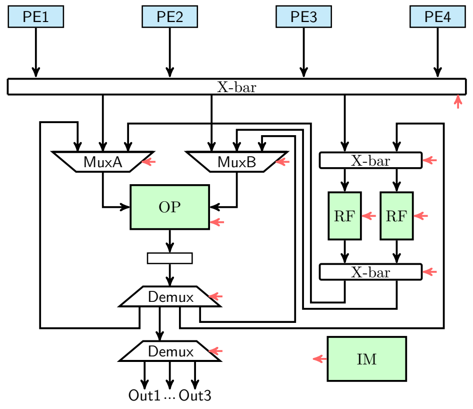


Fig. 7: Functional Unit template. PE1-PE4 represent "parent" PEs that generate input data. IM is an internal Instruction Memory,where the PE stores the operations to be performed. RFs are internal Register Files, which store reuse data and inputs to be used in the future. OP is the hardware unit actually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

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 - Section VIII-B - compares the use of MRAM - modeled according to [28] - and SRAM for L2M (both in 28nm) for the two applications, thus showing the impact of choosing different memory technologies in the L2M. The last case study - Section VIII-C - compares μ-Genie architectures for the different MV sizes.

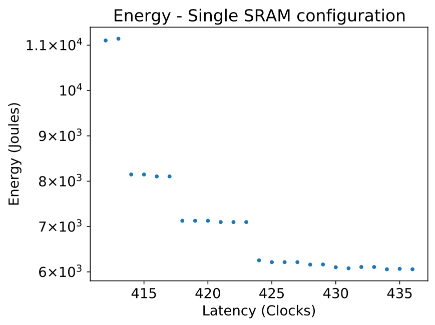
A. Single configuration DSE

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.

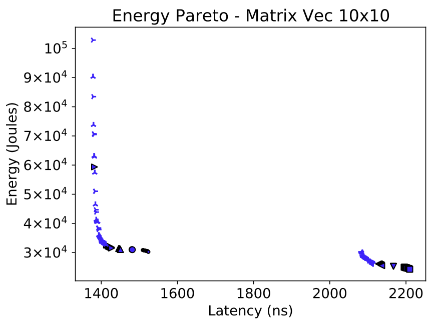
B. MRAM vs SRAM Level 2 Memory

In this case study we compare the energy efficiency of two alternative technologies to implement L2M: 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.

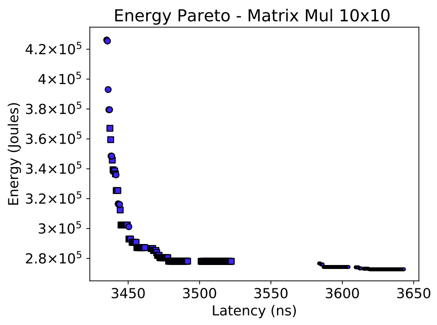
In both figures we can identify two clusters: LL-HE (low-latency, high-energy) and HL-LE (high-latency, low-energy). Each cluster belongs to one memory technology: LL-HE contains all SRAM designs, while HL-LE contains all MRAM designs. For the MV application (Fig 8b) the fastest architecture - using 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 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. The matrix multiplication, Figure 8c, performs 10 times more operations 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 scales 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 behavior caused by the increased number of write operations that have high energy impact when the MRAM technology is used.

IX. Related Work

Previous work on designing 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 scratch-pad 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.

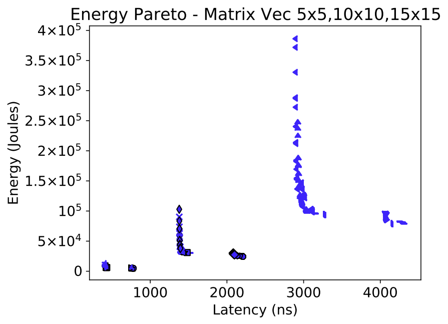


Fig. 9: Energy Pareto optimal architectures generated by μ-Genie for different sizes of Matrix Vector multiplication 5x5 - with latency ranging from 0 to 1000,10x10 - having latency between 1000ns and 2500ns , and 15x15- with latency above 2500ns. Each point corresponds to an architecture generated by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

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 configuration, 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 asynchronous, and operation dependencies are handled using a token-based mechanism which is implemented in hardware. The memory system of the architecture consists of a monolithic 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 back-to-back, forming systolic arrays which are then interconnected using an on-chip interconnect. Thus, 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 interesting approach is Catena [5], an ultra-low-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 performance 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 architectures 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 μ-Genie 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*.

XI. Method

This chapter presents our method for constructing efficient multi-mode architectures. We start by considering possible efficiency metrics and their applicability in our case to answer RQ1. Next, we present the actual merging method, which combines two application-specific architectures, thus answering RQ2. Finally, we extend this method with a DSE, to find pairings within the sets of input architectures that result in efficient merged architectures, answering RQ3.

XII. Efficiency metrics

The first step in our research is to establish which metrics to consider for efficiency. Doing so helps guide the creation of our method towards a shared goal, helping us decide the input and steps our method should take in order to best optimise these metrics.

We base our discussion on latency, area usage, and energy consumption, as these metrics are the most common in the HLS and DSE literature seen so far. The latency of an architecture represents the time needed to execute the program it implements, measured in clock cycles. Area usage refers to the amount of physical area required to implement the architecture on a chip. Lastly, the energy consumption is the amount of energy required to execute the application, and is divided into the passive energy, determined by passive leakage of electrical energy in hardware, and the dynamic energy, required to activate the components.

Out of these, area is the most important metric to us, as we are specifically aiming to design a merging method that exploits resource sharing. Because area is determined by component count and size, and the goal of resource sharing is to avoid unnecessary resource duplication, minimising the area of the merged architecture implies maximising resource re-use. Thus, our method aims to optimise the area of the merged architecture.

In contrast, the latency of the input applications in the output architecture is directly related to their latency in the original architectures. By selecting architectures with higher latencies, we can use less parallel and thus lower area architectures in our merge. To explore the different combinations of latency and area, we rely on the latency of the original architecture, and make use of the DSE process to select the pairs of architectures to merge.

Finally, energy is made up of two components, the static and dynamic energy. The dynamic energy of an application is determined mostly by the amount of operations in the application, and is thus largely constant regardless of architecture. On the other hand, the static energy is determined by the passive leakage of the hardware, and only depends on the area and latency. Because a merged architecture is larger than its inputs, running a program on a merged architecture incurs an energy overhead compared to running it on its original architecture. Thus, the energy consumption of the architecture is highly dependent on the values of the latency and area.

In conclusion, when merging input architectures, we prioritise minimising the area of the merged architecture, while preserving the latency of the input architectures, and minimising energy consumption.

XIII. Merging method for architectures

Our goal is to devise a method to merge two application-specific architectures as output by the μ-Genieframework. The resulting architecture must be able to run each application individually, and minimise area and static energy while preserving the latency and dynamic energy of the original input architectures. We note that it is not necessary for applications to be able to run concurrently, as our goal is to be able to run *either* application on the same hardware, rather than *both* at the same time. We focus on creating the necessary *hardware*, relying on a separate step to create the necessary hardware instructions to run the input applications on the merged hardware.

To meet these requirements, our method that merges two architectures, represented as graphs of FUs, has three major steps:

1) Isolate the maximum common subgraph *mcs* of the input graphs.
2) Mapping the 'leftover' nodes where possible.
3) Completing the graph with the remaining 'leftovers'.

We further present the motivation and procedure for each of these steps.

A. Isolating the mcs

We begin the merge by isolating the *mcs* of the input graphs to use as the 'core' of our merged architecture. By definition, this is a subgraph that exists within both input graphs that is also maximum in its size compared to other common subgraphs. Thus, the *mcs* represents the largest part of the input architectures that we can directly reuse in our result without introducing any additional hardware or overhead, as it is used in both original architectures. We note that multiple *mcs* of equal size may exist for any pair of architectures, any one of these are valid for our method.

We solve the *mcs* by transforming it to the maximum clique problem on the compatibility graph of the input graphs (as outlined in Section **??**). We impose an additional restriction to the *mcs* found in this manner: two FUs can only map onto each other if they perform the same operation in hardware. Because we rely on a separate step to implement the input applications on the merged hardware, we do not consider the instruction memory in the FUs for this restriction.

B. Mapping leftovers

After isolating the *mcs*, we are left with two sets of 'leftover' nodes. Our next step is to map together—that is, map on shared resources—any nodes of the same type between the two leftover sets. Pairing two nodes from the two different sets, and mapping each pair on a new shared resource creates *one* new node in our merged graph, with all of the connections of both original nodes.

Matching these leftovers results in new nodes with edges that should only be active when one of the two applications is running, meaning that we must be able to ignore certain hardware connections at runtime. This is possible by adding switching hardware where these edges meet, allowing the hardware to select which of multiple inputs to connect to an output. In the FUs used in our framework, this functionality is provided by the crossbar.

Compared to the previous step, mapping FUs in this way creates additional hardware overhead, in the form of hardware connections that are used only in one of the two applications. These added connections increase the amount of control logic required in both the memory and hardware of the FU. When an FU has more connections, instructions require more bits to address the different connections to select the inputs, increasing the area of the instruction memory. More hardware is also needed in the crossbar of the FU itself, contributing to the overall area increase. To minimise the cost of this overhead, we prioritise mapping together FUs with overlapping edges leading to the same node.

Following this, we also note that, due to the way the partitioning algorithm used for allocation in μ-Genie, the amount of components for a certain operation in an architecture is always equal to the maximum amount of operations of that kind that run in parallel at any point in the application. Consequently, the merged architecture should never need more of one kind of component than any of the original architectures. With maximum sharing, this means that the amount of components of a certain type in the merged architecture is exactly equal to the maximum amount of components of that type found in any of the two input architectures. Thus, we continue mapping the leftovers until we run out of compatible FUs implementing the same operation.

C. Completing the graph

As a final step, we *add* all remaining leftovers—i.e., the FUs not mapped in the previous step—along with all their connections. Any nodes added at this point are guaranteed to belong to only one architecture. This leaves us with a supergraph containing both of the input graphs as non-induced subgraphs. This supergraph is capable of running both input programs by mapping the instruction memories from the input units to the corresponding units in the output architecture.

XIV. Design-space exploration for efficiency

Our method so far allows us to merge any two architectures together. In order to explore the merge of the two input applications, one question remains: how we can select *the right* pairs of architectures for the input applications, such that we can create efficient merged architectures? To address this problem, we perform a design-space exploration (DSE), wherein the space represents all possible pairings, while the selection of efficient ones is made using the proposed efficiency metrics. The full merging flow is shown in Figure 10.

For DSE, we first define the design space as the complete set of possible architectures for the input applications, i.e., the Cartesian product of the sets of architectures for each input application. Doing so gives us access to pairs of architectures with different parameters for all of our efficiency metrics (area, latency, and energy consumption), ensuring the completeness of our design space.

Next, we analyse each merged architecture using our efficiency metrics. Because each resulting architecture is multimode, it has separate latency and energy metrics for each individual mode, because both of these metrics depend on the execution. By construction (as designed in Section XIII), the latency and dynamic energy of each mode are the same as in

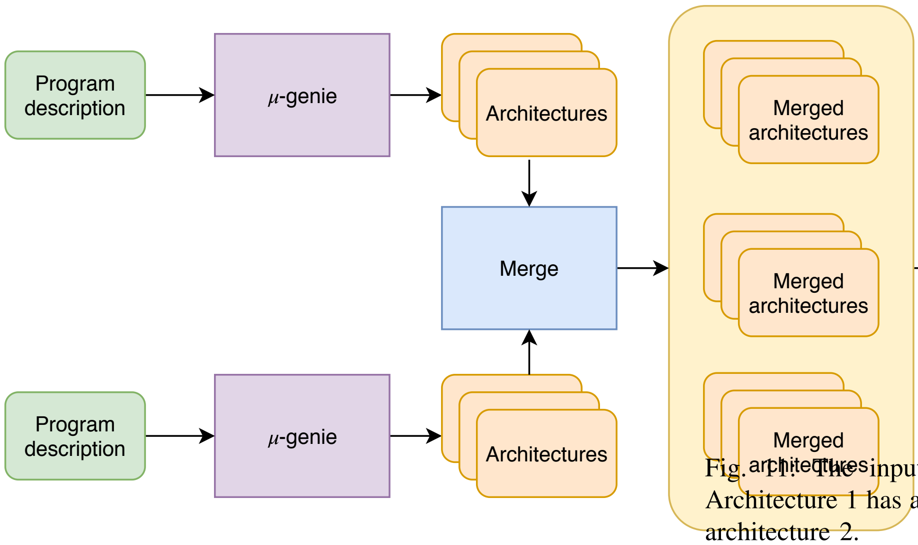
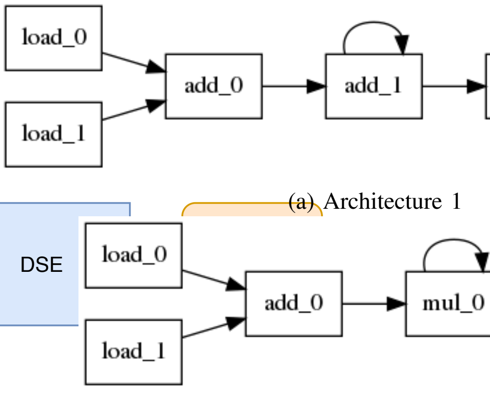


Fig. 10: A view of our full merging flow. We use μ-Genieto generate sets of architectures from the program descriptions. After merging these sets, we use a DSE process to find the merged architectures that are efficient.



(a) Architecture 1

(b) Architecture 2

Fig. 11: The input architectures of our running example. Architecture 1 has an extra `add_1` node inserted compared to architecture 2.

the original architecture, whereas the static energy *changes*, because it depends on the area of the merged architecture.

Finally, in our exploration, we find the Pareto set of the merged architectures. To do this, we assume a scenario in which all modes run equally often, i.e., for every execution of application 1 there is also an execution of application 2. Under this scenario, we can approximate the latency and energy of the merged architecture as the sum of the latency and energy of both modes. We note that with this method, different exploration strategies can be used by changing the way the metrics are estimated according to the scenario.

Using this DSE approach, in which we explore merges of *all* the individual architectures implementing *each* of the two given applications, and *select only the efficient ones*, we generalise from a method of merging two architectures to a method for merging two applications.

XV. Implementation

This section presents the pseudocode and implementation details of the methods formulated in this chapter, accompanied by a running example that demonstrates the architecture merging method in practice, step by step. The input architectures for this example are shown in Figure 11.

We calculate the *mcs* on the input graphs with any self-edges removed. Since self-edges in the μ-Genieoutput represent reads or writes from the component's own memory, these loops are irrelevant to the overall topology. However, the existence of loops in the graph would automatically mean that nodes containing them are non-isomorphic to nodes without them, and therefore cannot be matched as part of the *mcs* algorithm. By removing such edges, we correctly maximise the coverage of the *mcs*.

We find the *mcs* by computing the maximum clique [29]. on the compatibility graph between our two input graphs. We construct the edges of the compatibility graph by iterating over each pair of mappings in the compatibility graph, skipping any that map nodes of different types together. We add an edge between two mappings if they are *compatible* (as defined in Section **??**). This construction is illustrated in Figure 12. Each node in the maximum clique represents a pair of nodes, one from each input graph, which we map to each other to create the *mcs*. Figure 13 illustrates the *mcs* for our running example.

```
for vertex v1 in graph1:
    for vertex u1 in graph1, where v1 != u1:
        for vertex v2 in graph2, where v1.type == v2
    .type:
            for vertex u2 in graph2, where u1 != u2
    and u1.type == u2.type:
                if edge_exists(v1, u1) ==
    edge_exists(v2, u2):
                    add_edge(<v1,v2>, <u1,u2>)
```

Compatibility graph construction

Fig. 12: Pseudocode of the compatibility graph construction from two input graphs, where `v1` and `u1` are vertices in graph 1 and `v2` and `u2` are vertices in graph 2.

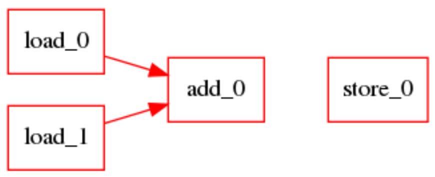


Fig. 13: The disconnected *mcs* of the running example.

With the *mcs* ready, we create the two leftover sets of nodes from each architecture by iterating over each node, adding any that are not part of the *mcs* to the leftovers. We map the leftover nodes by type, and minimise the creation of new edges by selecting nodes with matching edges in the process. We do so by counting the amount of overlapping edges for each pair, and selecting the pairs with the most overlap. Edges are considered overlapping if the nodes connected to them map to each other, ignoring nodes that are not yet mapped. Figure 14 shows the pseudocode for the matching algorithm, and Figure 15 illustrates the graph resulting from applying this to our running example, with the *mcs* highlighted in red and the extended leftovers in blue.

```
while new pair possible:
    for vertex v in leftovers_in_graph1:
        for vertex u in leftovers_in_graph2, where v
    .type == u.type:
            overlap = 0

            for vertex a in adjacent_vertices(v):
                if equivalent_vertex_in_graph_2(a)
    in adjacent_vertices(u):
                    overlap += 1

            if overlap > max_overlap:
                best_pair = <v,u>
                max_overlap = overlap

    add_vertex_from_mapping(best_pair)
```

Mapping leftovers

Fig. 14: Pseudocode of the mapping procedure for the leftovers.

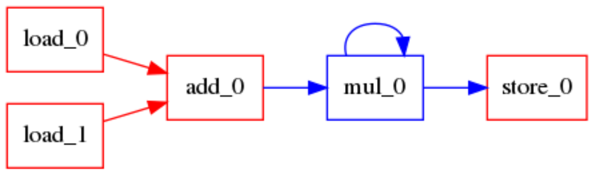


Fig. 15: The *mcs* of the running example extended with the mapped leftovers.

Finally, we add any unmatched leftover nodes to the result graph along with all of their edges. The result is a complete supergraph, as illustrated in Figure 16. We see that both original input graphs are present in this final result graph.

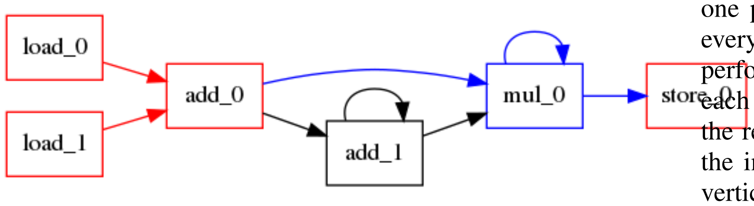


Fig. 16: The final result of the running example.

We perform our exploration over the Cartesian product of the sets of architectures created by μ-Geniefor each of the two input applications. For each pair in this set, we apply our merging method, resulting in a merged architecture, along with the mapping of each FU in the output architecture to one or two FUs in the input architectures. This information allows us to compute the metrics of the merged FU from its original FUs, enabling the exploration.

FUs differ in area depending on the area of the instruction memory, register files, and operation hardware. To be able to run both input programs, a merged FU must naturally have at least the area of the largest FU it was merged from. We compute the area of the merged FU as the maximum of the areas of the two original FUs. Similarly, the static energy per latency, which is based on the area, is the maximum of the original FUs. Finally, for the dynamic energy, which is only dependent on the FU's program, we use the original values.

To compute the total metrics for the merged architecture, we sum the individual metrics of all FUs in the architecture. In accordance with the scenario outlined previously (see Section XIV), we sum the latency of both input architectures to obtain the latency we use for exploration, multiply it with the sum of the static energy per latency to obtain the overall static energy, and add to it the sum of the dynamic energy to obtain the total energy of the architecture.

We note that the area calculation for FUs outlined above is slightly simplified. As the area of the FU is the sum of the areas of the OP, RFs, and IM, each of which can vary depending on the bit sizes used and the FUs program, it would be more accurate to sum the individual maximum area of each sub-component. This is potentially relevant in certain edge cases where one sub-component is larger in one input FU and another sub-component is larger in the other. Additionally, we do not yet take into account the area increase created by increasing the amount of inputs to an FU, either due to increased crossbar hardware or due to the increased instruction memory size required for addressing. These discrepancies may minimally affect the result of the area calculation.

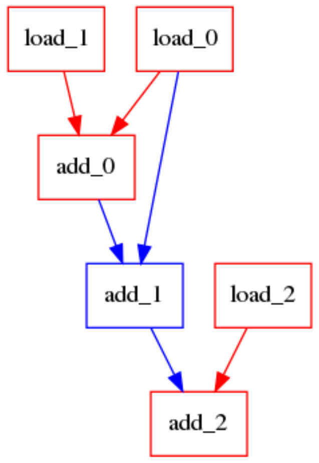
XVI. Empirical Evaluation

In this chapter, we present the validation of our merging method in terms of the correctness of the merge. We further present a thorough evaluation of the performance of our method in terms of the efficiency of the merged architectures.

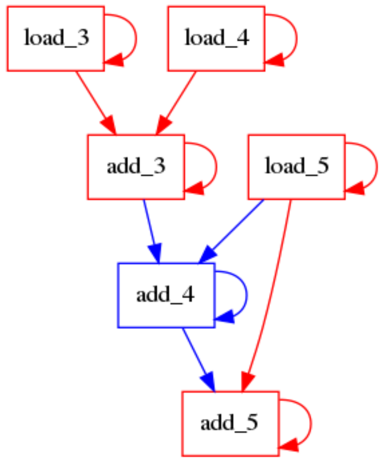
XVII. Architecture merging validation

We evaluate the correctness of our merging method by manually assessing its results on three test cases, each targeting one part of the algorithm. For each test case, we verify that every vertex in the input graph maps to a unique vertex performing the same operation in the output graph (such that each operation in the application can be bound to an FU in the result), and that for every edge connecting two vertices in the input, there is a corresponding edge between the merged vertices in the output (such that each operation has access to the correct inputs). We show the input and output architectures of each of our test cases, with the *mcs* found by our algorithm marked in red, the mapped leftovers marked in blue, and the final set of leftovers marked in black.

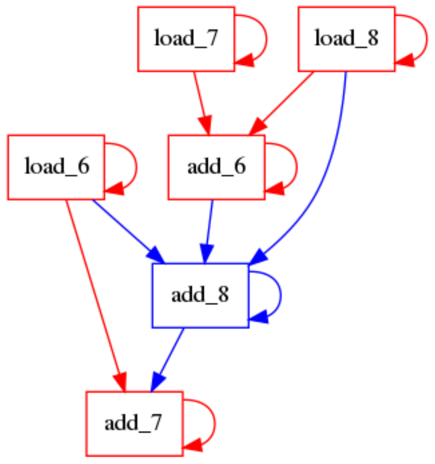
Our first case tests the capability of the merging method to find an *mcs* that is disconnected and has self-edges in only one



(a) Architecture 1



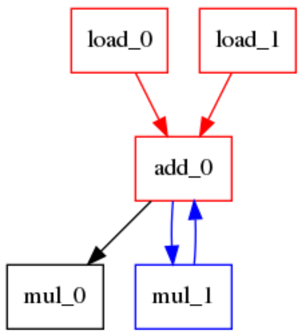
(b) Architecture 2



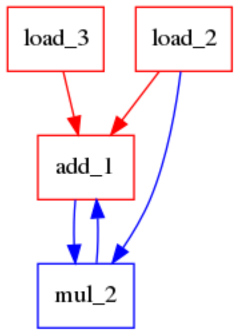
(c) Result

Fig. 17: `add_1` and `add_4` are not part of the *mcs* due to being connected to `load_0` and `load_5` respectively. `load_2` and `add_2` map to `load_5` and `add_5` respectively, forming one part of the *mcs*. The cluster of two loads and one add forms the other part of the *mcs*.

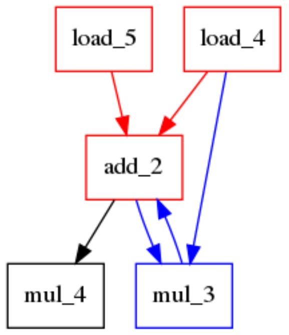
input. To this end, we construct two architectures based on an *mcs* made up of two disconnected subgraphs, one of which has a self-edge on every node. We connect the two parts of the *mcs* together with an extra node, alternatively connected to one or the other part of the *mcs* depending on the application so that it cannot be part of the *mcs*. The construction is shown in Figure 17. We see that the method correctly finds the disconnected *mcs*.



(a) Architecture 1



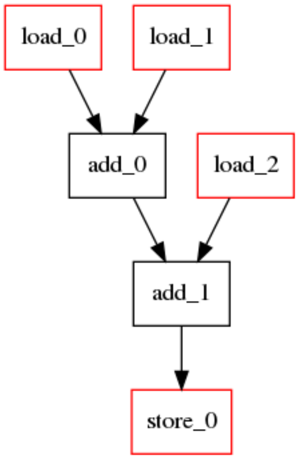
(b) Architecture 2



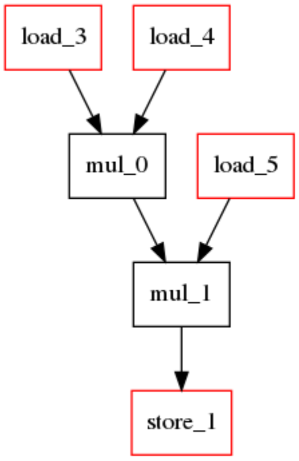
(c) Result

Fig. 18: `mul_1` and `mul_2` cannot be part of the *mcs* due to `mul_2` connecting to `load_2`. Both `mul_1` and `mul_2` have two connections to the add in the *mcs*, and are thus merged together.

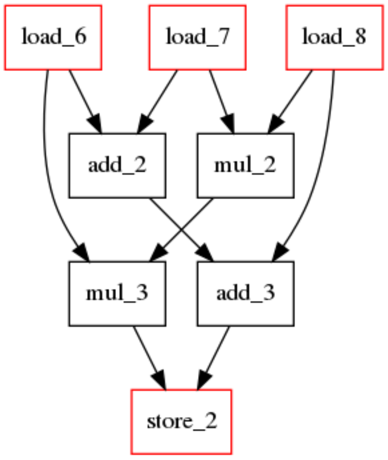
To test the leftover matching algorithm, we construct a pair of input graphs where a leftover node in one graph has two possible nodes to map to in the other graph. Of these nodes, one node has an edge overlap of one, while the other has an overlap of two with the node to be mapped. This construction is shown in Figure 18. We see that the node is merged to the



(a) Architecture 1



(b) Architecture 2



(c) Result

Fig. 19: All three load nodes and the store node are identical, and thus part of the *mcs*, though disconnected. We see that all four unique nodes are added to the result as leftovers, and all appropriate edges are added.

correct node that it overlaps most with.

Our final test case, shown in Figure 19, tests the addition of multiple leftovers to the final graph. It consists of a sequence of two identical arithmetic operations, where the output of the first operation is used as an input of the second one, with all corresponding loads and stores. The difference between both architectures is only the type of operation. We see that our algorithm correctly identifies these components as different types, merging only the loads and stores, while putting the arithmetic operations next to each other in the merged architecture.

We limit our validation of the merging method to three experiments, showing that the individual parts of the merging process behave correctly, illustrating the correctness of our method in practice.

XVIII. Method evaluation

We evaluate the results of the DSE by applying the merging method on two representative applications, a 5×5 matrix-vector multiplication and a 5×5 matrix-matrix multiplication, illustrated in Figure 20. These programs give us 950 total merged implementations, are reasonably similar, and are likely to be used in the same context.

```
int sum;
for(int i=0;i<5;i++){
    sum=0;
    for(int j=0;j<5;j++){
        sum+=A[i*5+j]*B[j];
    }
    C[i]=sum;
}


```

5×5 matrix-vector multiplication

```
int sum;
for(int i=0;i<5;i++){
    for(int j=0;j<5;j++){
        sum=0;
        for(int k=0;k<5;k++){
            sum+=A[i*5+k]*D[k*5+j];
        }
        E[i*5+j] = sum;
    }
}
```

5×5 matrix-matrix multiplication

Fig. 20: Input program source code.

To evaluate the effectiveness of our method, we consider the efficiency of the merged architectures in terms of resource sharing. We quantify the resource sharing efficiency as the area reduction of the merge, based on the ratio of the merged area to the sum of the areas of the original architectures, showing the amount of area saved by merging the architectures. We also consider the energy overhead created by the merge, quantified as the ratio of the merged energy to the sum of the unmerged energy, showing the static energy costs caused by the increased area relative to the architecture for any single application. The metrics we evaluate are summarised in Table III.

Metric	Descri
Latency	Sum of the latency of both inp clock c
Area	Area of the merged archite
Energy	Sum of the static and dynar architectures (mea
Area Reduction	1 minus the output area divided (rendered as percenta
Energy Increase	Output energy divided by the su 1 (rendered as percent

TABLE III: List of metrics that we evaluate for ou architectures.

Based on the area reduction and energy overheac we are able to analyse which situations create the me the most improvement. Additionally, by plotting and the efficiency metrics of each architecture resulting merge, we can gain a better understanding of the trade-offs resulting from merging applications.

A. Findings

We summarise the results of our experiments in th ing six findings:

1. Pairs of architectures with similar area anc perform better in terms of area reduction and energy than pairs that differ more.
2. Lowest-latency architectures result in the highest area reduction and lowest energy overhead, whereas the area reduction and energy overhead of the highest-latency architectures is highly dependent on application size.
3. The area reduction for effective merges generally lies between 20 and 40%. For Pareto-optimal architectures, the range is 25 to 40%.
4. The energy increase for effective merges generally lies between 10 and 20%.
5. High area reduction combined with low energy overhead is a good indicator of the Pareto-optimality of the resulting architectures.
6. The Pareto set of merged architectures shows meaningful design points with different trade-offs in terms of latency, area, and energy.

The following sections show the empirical evidence and the accompanying analysis that has lead to these findings.

B. Relative merging efficiency

To assess the relative efficiency of the merged architectures compared to the input architectures, we plot the pairs of input architectures together with their achieved area reduction and their energy increase overhead. The results are shown in Figures 21 and 22.

We immediately notice, from the shape of the plots, that all results are largely divided into 4 sections, with merges in the low-latency $\times$ low-latency or high-latency $\times$ high-latency domains performing better than the mixed-latency parts, having both higher area reduction and lower energy overhead. We also notice a similar trend along the diagonal, starting from the origin and growing towards the top right,

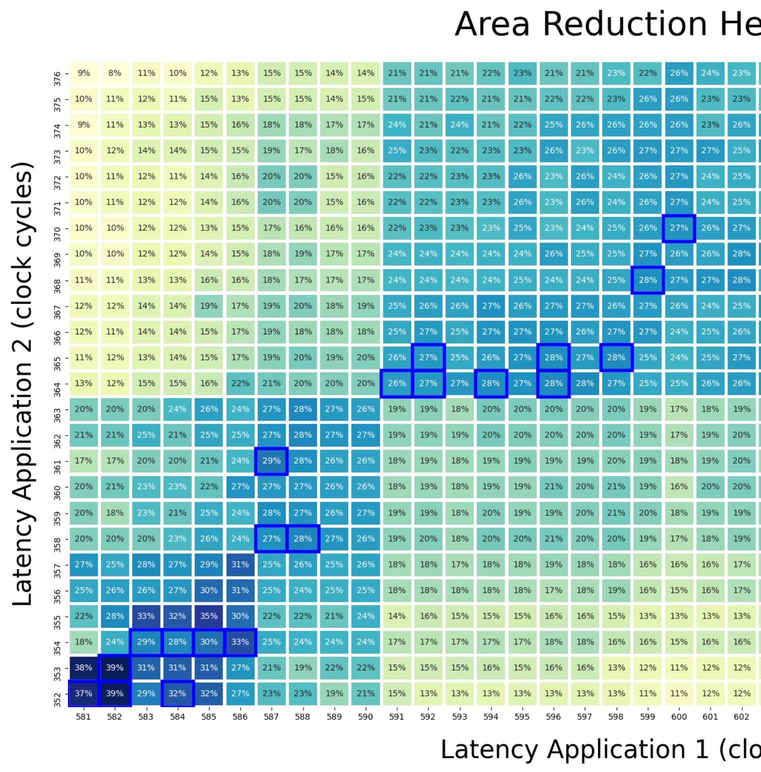


Fig. 21: Area reduction for each pair of input architectures. Highlighted with blue borders are the architectures that are Pareto-optimal for latency, area, and energy.

where merging performance is highest, most noticeable in the low-latency high-parallelism domain. Thus, we see that merging performance depends largely on the ‘closeness’ in the amount of parallelism of the input architectures.

This observation is in line with our expectation, as when merging a large and parallel architecture with a small and sequential one, the resulting architecture is at least as large as the large architecture. This means that the maximum achievable area reduction of such a merge is equal to the size of the small one, which is a small gain compared to merging architectures of the same size. Similarly, because the static energy of the architecture is a product of the area and latency, such a merge results in a larger energy overhead where the large architecture is active for the high latency associated with the smaller architecture. This leads us to Finding **??**.

We also notice that among our efficient merges, the best performing merges are also the ones with the lowest latency. Since our area ratio scales with individual FU area differences, a likely explanation is that for these architectures, mostly FUs that are close in size are being matched. Because these lowest-latency architectures are also maximally parallel, each individual FU performs a minimal, and thus similar amount of work, equally requiring a similar amount of area. Conversely, because each FU in the most-sequential architectures scales with the overall amount of operations in the application, the merging performance of these highest-latency architectures

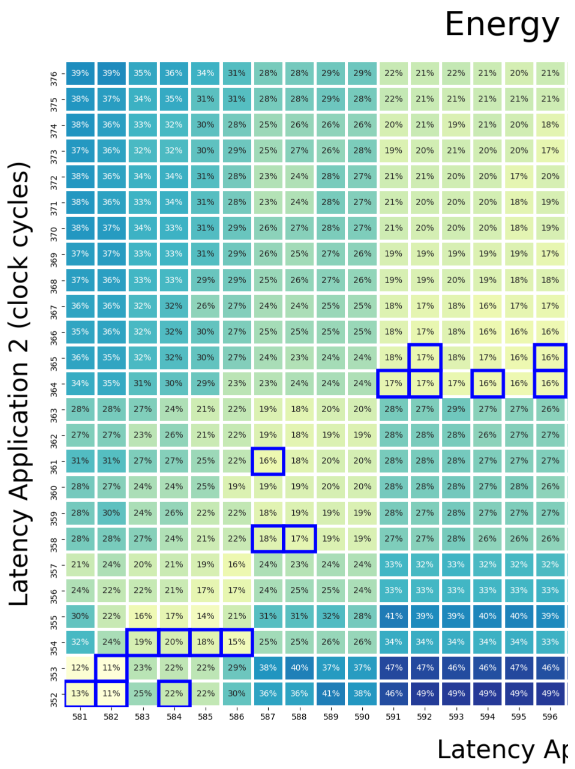


Fig. 22: Energy increase for each pair of input architectures. Highlighted with blue borders are the architectures that are Pareto-optimal for latency, area, and energy.

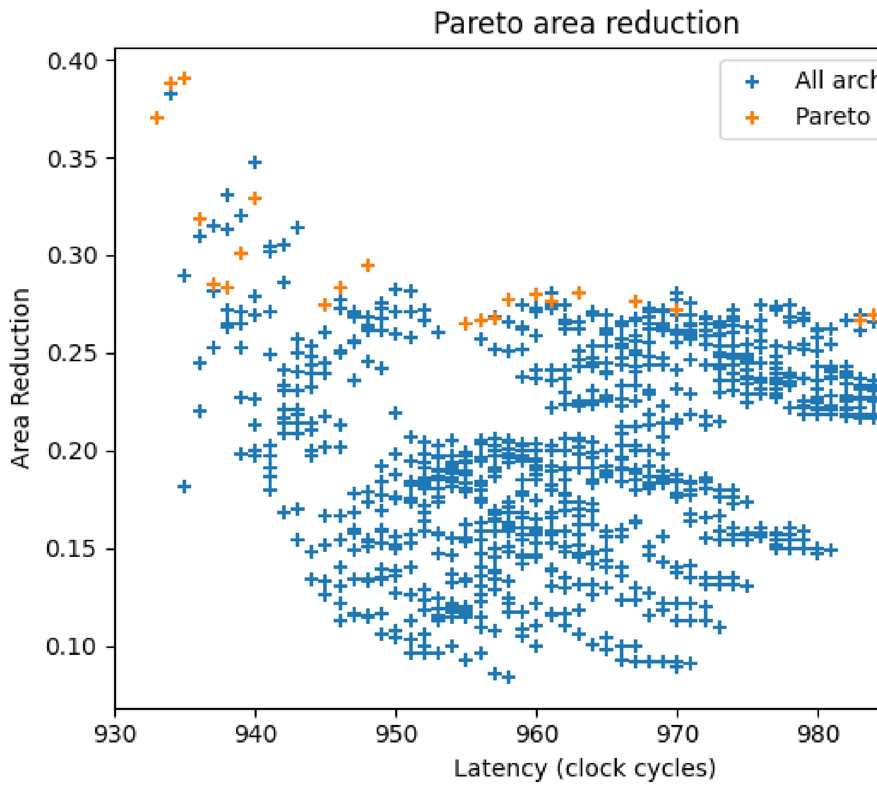


Fig. 23: Comparison of the area reduction between the Pareto set of architectures and all architectures.

likely scales with the difference between the application sizes. This leads to Finding **??**.

Ignoring the pairs of architectures with the worst merging performance present in the mixed-latency regions of our graph, we see that the merging performance generally lies between 20 and 40% with few outliers. Furthermore, if we consider only those pairs resulting in architectures Pareto-optimal in either latency, area, or energy, highlighted in blue, this range further reduces to 25 to 40%. Similarly, the energy overhead for efficient merges lies between 10 and 20%, giving us Findings **??** and **??**.

Finally, we also see that the Pareto-optimal architectures tend to coincide with the best-performing merges at the various latency points. Once again this is in line with our expectations, as more efficient merges allow us to attain lower latencies at the same area and energy costs compared to worse performing merges. Plotting the latency and area reduction separately gives us another view of this, shown in Figure 23. In this figure, the Pareto architectures are shown to be at or near the highest area reduction for each latency, leading us to Finding **??**. This means that we can aggressively prune the design space towards higher-performance merges following our previous findings, without loss of optimal architectures.

C. Merged architecture trade-offs

To assess the efficiency trade-offs of the merged architectures themselves, we create similar plots for the absolute area and energy numbers of the architectures, shown in Figures 24 and 25. We also observe the Pareto plots of the area and energy compared to the latency in the final architectures, shown in Figures 26 and 27.

We immediately see that both the area and energy scale with latency, with low latency for either program resulting in both high area and high energy, and vice-versa for high latency. We also see that both graphs are nearly identical, ignoring magnitude. This tells us that in this case, the total energy is dominated by the static energy, which scales with area, rather than the dynamic energy, which scales with program size. We also see that the area and energy numbers change less as latency increases.

The Pareto plots show us a different view of the area and energy compared to the latency. Here, once again, the results are in line with our expectations, as decreasing the latency leads to a trade-off requiring increased area and energy. Additionally, we see that both forms of trade-offs have diminishing returns, i.e., decreasing latency further requires larger increases in area or energy, and vice-versa for decreasing area or energy.

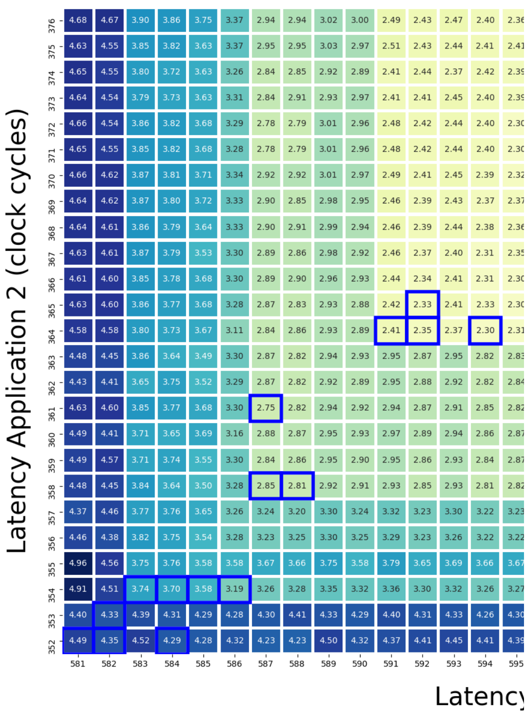


Fig. 24: Resulting area for each pair of architectures. Area measured in $1e^5$.

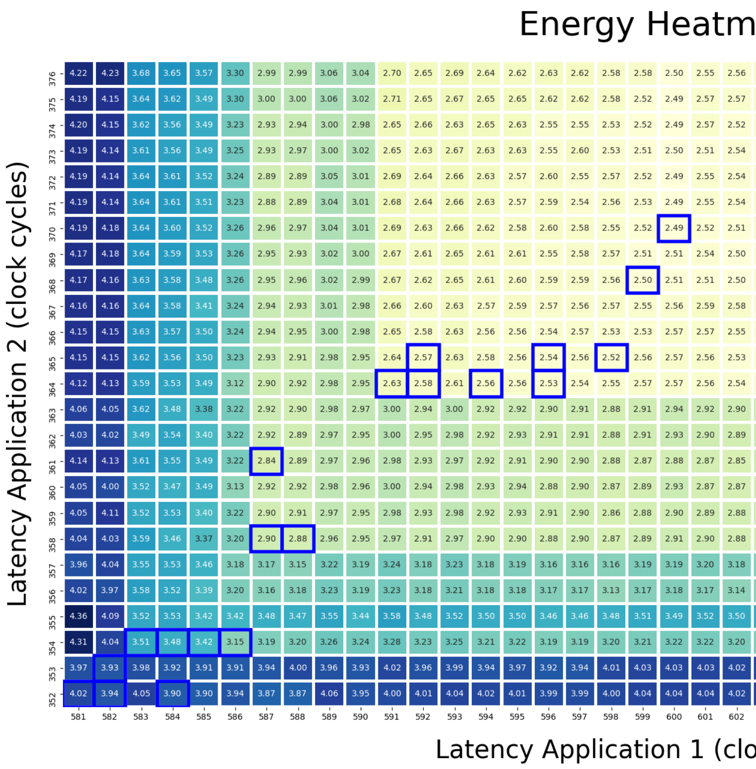


Fig. 25: Resulting energy for each pair of architectures. Energy measured in $1e^4$.

Accordingly, the most effective trade-offs exist in the middle of the curve. This shows that we can, through our merging and DSE process, find meaningful design points with different trade-offs in terms of latency, area, and energy, giving us our final Finding **??**.

XIX. Programming the PEs

In this chapter we discuss the requirements and the design for our automated programming of functional units, as well as the assembly language. Specifically, we go over the requirements the methodology needs to meet in XX, we discuss the way the assembly has been defined in XXI, and we provide an overview of the methodology in XXII, where we go over the individual parts of the methodology separately.

XX. Requirements

The output of the methodology is a list of FU instructions, discussed in **??**, for every component in the architecture, such that the components behave as described by their respective FUs in the annotated DFG.

To guarantee the correctness of the execution according to the input application, the main requirement is that, for every operation in the annotated DFG, the component needs to be programmed to let the correct two inputs flow through the OP component at the time specified in the annotation of the operation. To achieve this we define three more requirements which relate to the source for the input data for the OP component and the timing of the arrival of that input data. First, we note that input can arrive on the input ports from other components, as described in **??**. Thus, to be able to correctly read input data, we need to know which component is connected to which input port. Second, we note that input data might arrive earlier than the operation on this data is scheduled to occur in the annotated DFG. Thus, we need to be able to temporarily store data that cannot be processed right away. Third, and last, we need to fetch the data when the component is ready to perform the corresponding operation.

To allow us to design a methodology that meets these requirements we create an abstraction of the complex FU instructions. This abstraction is created by defining a high-level assembly language consisting of assembly instructions that each determine a subset of the fields of the FU instruction described in **??**. We co-design the high-level assembly with the methodology such that each part of the methodology that meets one of these requirements will have a separate corresponding assembly instruction that it generates.

After the assembly instructions have been generated, they need to be compiled into FU instructions. To compile the assembly instructions we first need to merge them into full FU instructions, in which all fields are defined. We do this because the assembly instructions separately only determine subsets of the fields of an FU instruction. Next, because the sizes of the fields in the FU instructions differ per component, the fields of these FU instructions need to take up the correct number of bits such that they correspond with how the component is

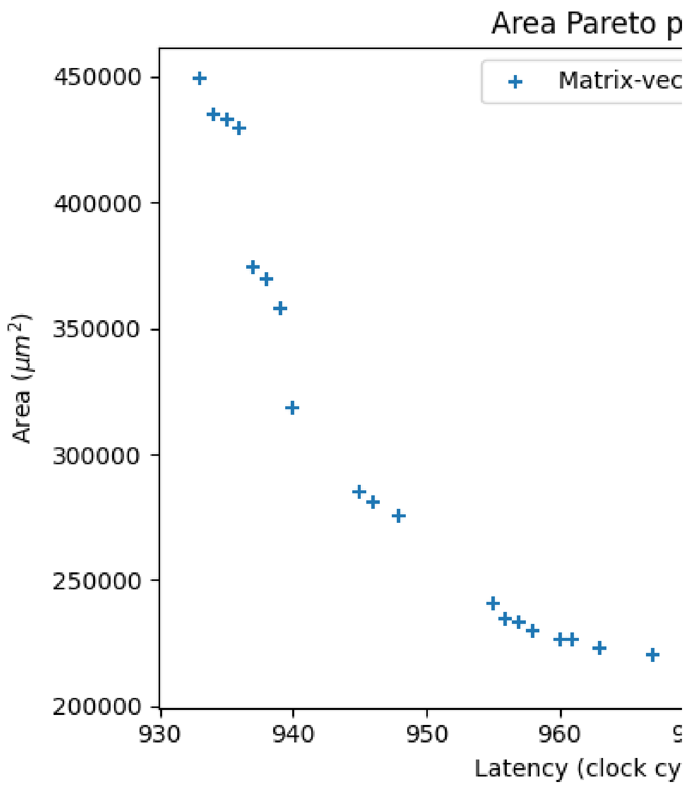


Fig. 26: Area Pareto plot for the architectures resulting from our exploration.

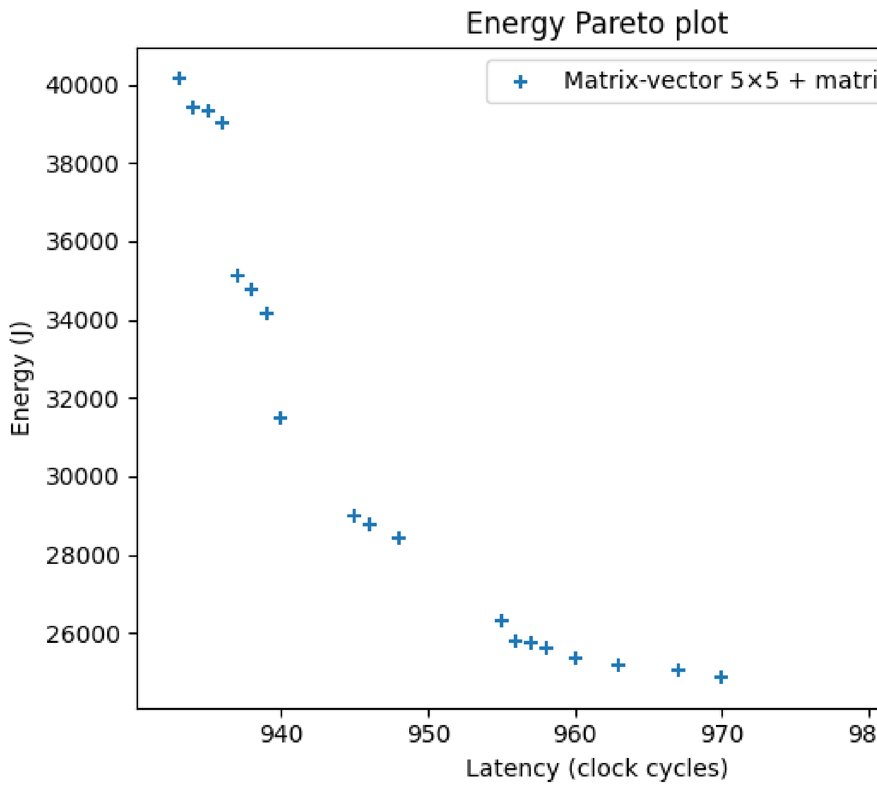


Fig. 27: Energy Pareto plot for the architectures resulting from our exploration.

input name	fields	description
OpInput	-	Input from the result of a previous *Op* instruction
FUInput	port	Input from one of the FU input ports
RFInput	-	Input from the register file in the component

TABLE IV: The input types which function as an abstraction to control the multiplexers, as well as which register in the RF data, coming from the input ports or as a result from a previously executed *Op* instruction, is written to.

configured. Lastly, the sequence of FU instructions all need to be loaded into the IM of the component by generating the control signals as described in **??** for each of them.

instruction name	fields	description
Op	clock, $input_1$, $input_2$	loads $\$input_1$ and $\$input_2$ into the OP part of the component at clock cycle \$clock. This input may or may not have a \$port value associated with it
Fetch	clock, address, mux	Loads the bits from the RF at \$address and sends them to multiplexer \$mux at clock cycle \$clock
Store	clock, input, address	Stores the bits from \$input in \$address at clock cycle \$clock

TABLE V: All instructions in the Architecture Template Assembly we specified.

XXI. Architecture Template Assembly Specification

The Architecture Template Assembly (ATA) is designed to be an abstraction of the FU instruction discussed in **??**. It is split into three different Architecture Template Assembly Instructions (ATAIs): *Op*, *Fetch* and *Store*. Because all the instructions are executed at a specific, predetermined, clock cycle we define our ATA to always have a \$clock field.

To control the OP of the component, discussed in **??**, we created the *Op* instruction. This instruction focuses on the main requirement of letting two input flow through the OP part at the right time. It takes two input sources whose values are defined in IV. $FUInput$ means that the data comes directly from the output of a different component through one of the input ports described in **??**, thus it needs to have the field \$port to denote which of the ports should be read from. $OpInput$ means that the result of the previous computation of the component is used as input for the next. $RFInput$ means that the input comes from the RF, this only sets the multiplexer for the input described in **??**. The actual fetching of the data from the RF happens through the $Fetch$ instruction.

The $Fetch$ instruction takes an RF address, which specifies the location where the data to fetch is located, as well as a multiplexer to send the data to.

The $Store$ instruction is used to store data in the RF. This instruction takes an input source as defined in V. The values

ATA field	MI fields	
clock	clock	
Op		
input$_1$	muxA	cbOut0
input$_2$	muxB	cbOut1
$Fetch$: mux=x		
address	r_REGx_s	
$Store$		
address	w_REGx_s	
input	REG0 REG1	cbOut2

TABLE VI: Relation of the fields in the ATA instructions to the fields in the FU instructions.

Input	Multiplexer value	cbOut value
OpInput	0	0
FUInput	1	port
RFInput	2	0

TABLE VII: Values for the multiplexers and their correspondig cbOut ports depending the value of one of the ‘input’ fields of the Op instruction.

this field can have are the $FUInput$ and the $OpInput$ entries in IV.

The different ATA instruction types listen in VI are designed to never have any overlap in the fields they influence in the FU instruction. Op determines the muxA, muxB, cbOut0 and cbOut1 fields depending on the values of the \$input$_1$ and \$input$_2$ fields as described in VII. $Fetch$ determines r_REG0_s or r_REG1_s. $Store$ determines w_REG0_s or w_REG1_s and their corresponding write-enable bits REG0 and REG1 as well as cbOut depending on the value of the \$input field as described in VIII.‘

XXII. Methodology overview

To meet the requirements laid out in XX, we designed a methodology that takes as input an annotated DFG and an *Allocation administration*, and produces a series of FU programs with some VHDL boilerplate code so they can be loaded into the IM of a component.

An overview of our methodology can be seen in 28. The methodology first analyses the interconnections between the FUs in the *FU input mapper*, then looks at each FU in the annotated DFG separately. Next is the graph exploration phase, in which the FU in the annotated DFG is transformed into ATA instructions. The *RF allocation generator* generates the $Store$ instructions using both the annotated DFG and the *Allocation administration* as inputs. The *Op and Fetch generator* then generates the Op instructions as well as any required $Fetch$ instructions. Next, the compiler phase starts, in which the ATA instructions are eventually turned into a list of FU instructions. The *Sorter* then sorts all the ATA instructions the methodology generates in the graph exploration phase by clock cycle, such that the *ATA merger-assembler* can merge the ones occurring at the same clock cycle into a single FU instruction. Finally, for every generated FU instruction, the *Control signal generator* sets all the ports and signals, as well as and the VHDL boilerplate code needed for loading the instruction into the IM (as described in **??**).

Input	REG0	REG1	cbOut value
FUInput	-	1	port
OpInput	1	-	0

TABLE VIII: Values for REG0, REG1 and cbOut depending on the value of ‘input’ of the $Store$ instruction.

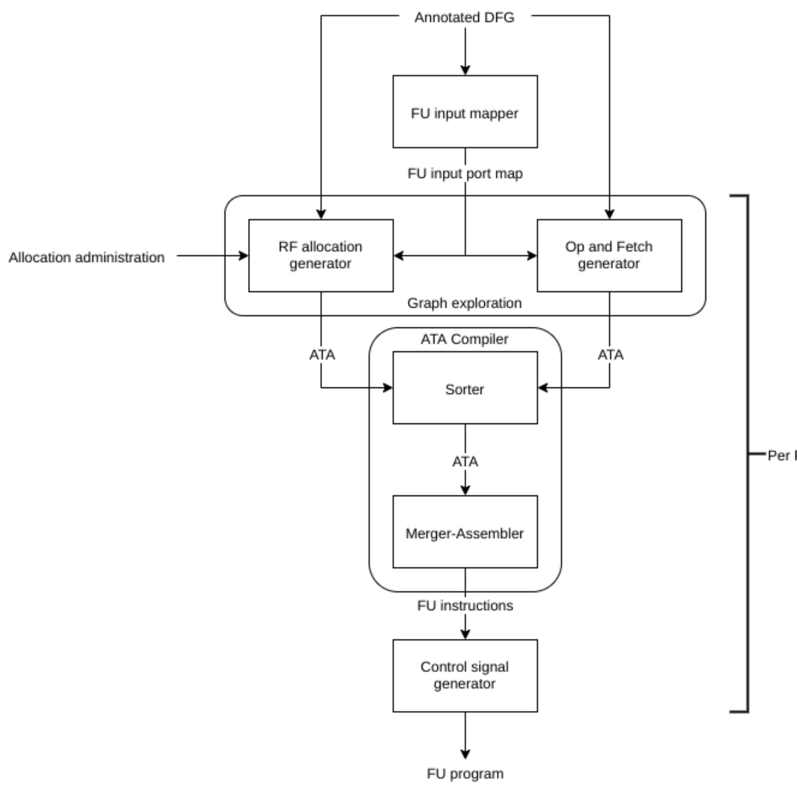


Fig. 28: Overview of the methodology (read from top to bottom).

Allocation administration

The allocation administration is an analysis that is performed by the μ-Genieframework and provides an input for the methodology. It traverses the graph and determines if the output of an operation should be stored in the RF of the component that receives the output. The allocation administration then uses a customised interval partitioning algorithm to optimise the amount of memory being used by overwriting the data in the RF if the data is no longer needed. The allocation administration generates a map for each FU that describes what its memory layout should be. This map describes, for each piece of data in the RF, at which address it should be stored at, and from which node the data comes.

FU input mapper

The *FU input mapper*’s job is to map, for every component, all its output ports to the input ports of other components. For every FU in the annotated DFG, the input mapper produces an alphabetically sorted list of FUs whose output port is connected to one of the input ports. The index of each FU in this list corresponds to the port number to which it is connected. The output of the FU input mapper consists of this list.

RF allocation generator

This phase generates $Store$ instructions, to ensure the storage of incoming data that cannot be immediately processed. The *RF allocation generator* does this by consulting the map generated by the *Allocation administration* for the current FU. A $Store$ instruction is generated for every entry in the map. The $address field is found by simply consulting the entry in the map. The $clock field is found by first finding the operation for which the entry in the map was generated. The latency of the OP of the component, as described in **??**, is then added to the clock cycle for which it was scheduled. The $input field is determined by looking if the operation resides in a foreign FU or not. The *RF allocation generator* outputs the $Store$ instructions as a list, for which no order is guaranteed.

Op and Fetch generator

As the name indicates, this module generates all the Op instructions, as well as the $Fetch$ instructions to load the Op input data from the RF. The *Op and Fetch generator* takes as input the annotated DFG, and (1) finds the FU describing the current component, and (2) generates an Op instruction for every operation in this FU. By looking at the annotated DFG vertices representing operations inputs, and comparing them against the map generated by the *allocation administration*, the Op and Fetch generator determines, for every Op input, whether it is stored in the RF or not. If the input is not in the RF, we look if the input vertex is in a foreign FU; in this case, we consult the map that the *FU input mapper* generates to find the port this foreign FU is connected to. We then set the input field of the Op to $FUInput$ with the $port field corresponding to the one found in the map. If the vertex is not in a foreign FU, and it is not stored in the RF, we set it to $OpInput$; if it is stored in the RF, we set it to $RFInput$ and generate a $Fetch$ instruction. For this $Fetch$ instruction we first determine the $clock field by subtracting the known fetch latency from the the clock cycle at which the Op is scheduled. Next, we consult the allocation map and set the $address field to the address where the data is stored. Finally, we set the $mux field to 0 if the $Fetch$ instruction was generated as input for muxA, or 1 if it was generated for muxB.

The Op and Fetch generator outputs a list containing Op and possibly $Fetch$ instructions. For this list no order is guaranteed.

Sorter and ATA merger-assembler

The *sorter* takes as input the lists of ATA instructions that the graph traversal phase generates. And outputs a single list of ATA instructions, all sorted by clock cycle from low to high. The *merger-assembler* converts the ATA instructions into FU instructions, and merges FU instructions occurring at the same clock cycle into a single one.

As described in **??**, within one clock cycle we can store up to two words, fetch up to two words, and perform one operation. As such, the merger is able to merge at most: one Op instruction, one $Store$ instruction for data from another component, one $Store$ instruction for data coming from an earlier Op within the component, and two $Fetch$ instructions. Because we may not always merge the maximum of one Op, two $Store$ and two $Fetch$ instructions, some fields in a FU instruction may not be defined by the assembly. These fields are simply set to 0, which means that the two write-enable bits $REG0$ and $REG1$ are set to 0 if the assembly does not specify a $Store$ instruction at that clock cycle. This ensures we do not unintentionally overwrite any data in the RF. The other unspecified fields do not influence the behaviour of the component, no matter what value they contain; however setting them to 0 provides a more complete definition of what the *merger-assembler* does.

Additionally, the *merger-assemble* keeps track of the highest clock cycle, the RF address, and the amount of input ports, so it can format the FU instructions such that all the fields take up the correct amount of bits by padding them with 0s where necessary.

Control signal generator

After all the ATA instructions have been converted to FU instructions, the FU instructions still need to be loaded into the IM. For this, the control signals described **??** need to be set. To facilitate this operation, the *Control signal generator* first sets the r_LOAD_INST signal to 1. Then the control signal generator adds the VHDL syntax for writing the FU instruction to the r_INPUT_INST port for every FU instruction. Additionally, the generator increments the IM address pointer after writing each instruction by setting the r_LOAD_NEXT_INST to 1. Lastly, the *Control signal generator* keeps track of how many clock cycles the components take to load their instructions into the IM, to make sure all components start counting clock cycles at the same time, and none is still busy loading instructions; this synchronisation is achieved by setting the r_COMPUTING signal to 1. The *control signal generator* outputs a string of VHDL code, which can be inserted into the template of every component in the network.

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.
[29] J. Konc *et al.*, "An improved branch and bound algorithm for the maximum clique problem," *proteins*, vol. 4, no. 5, 2007.