μ-Genie: A Framework for Memory-Aware Custom Processor Architecture Design-Space Exploration

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

***Abstract*—Custom processor architectures are essential to meet the increasing demand in performance and power consumption of embedded and high performance computing systems. Due to the massive amount of data handling by the processors, the memory system determines the overall performance and power consumption in silicon. Since the memory system and the processing system are interdependent, they must be co-designed. State-of-the-art tool flow for custom processor design-space exploration focus on processor architecture optimization but does not include memory system. In this paper, we present μ-Genie: an automated framework for co-design-space exploration of custom processor architecture and memory system starting from an application description in a high-level programming language. In addition, we propose an 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 custom-processor architecture using different memory technologies.**

I. Introduction

The interest for application-specific hardware is growing in different fields of computer science, from embedded to High Performance Computing systems. This is mostly due to the end of Dennard Scaling [1] and the ever increasing demand for performance and lower power consumption. Application-specific or *custom processor* hardware seems to be the solution to face current silicon challenges allowing energy savings and performance increase over general purpose counterparts [2]. With the massive increase in the data handling needed in custom hardware, the *memory system* (both on-chip and off-chip) is becoming a dominant factor affecting the overall performance, power consumption and area usage of the silicon. Since the memory system and the processing system are strongly *interdependent*, they should be *co-designed*. This is especially important for emerging memory technologies, such as MRAM, eDRAM, PCM, RRAM [3], that has high integration density and lower power than SRAM, but with different latency for read/write operations

Commercial CAD tools [4]–[6] are available to help with the increasing complexity of custom processor design. However, selecting an optimal hardware architecture taking into account the various trade-offs in latency, power consumption and area usage of all the possible design choices is still a challenging task. State-of-the-art design flows for custom processor architecture design-space exploration focus on processor architecture optimization [7]–[10] and does not include the Memory System, as illustrated in Figure 1. Co-optimization of processor and memory system (including emerging memories) is typically done by co-simulation of the processor and the memory system and optimizing the cache replacement policy [11]–[14]. On the other hand, emerging *spatial architecutres* [15], [16] for custom processors helps to distribute the program memory and processing elements to achieve maximum performance, but uses "non-custom" and complex interconnect.

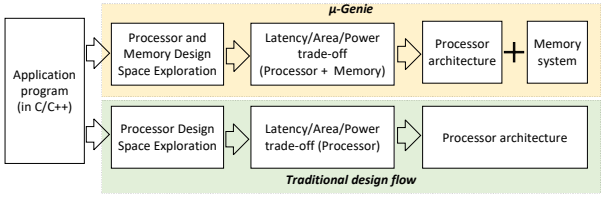


Fig. 1: Difference between state-of-the-art and the proposed μ-Genie design flows for custom processor architecture design.

Our main contributions in this paper are:

- μ-Genie: An automated framework for memory-aware custom processor architecture design-space exploration and provide different tradeoffs, as shown in Figure 1. The framework supports different kind of memory technologies and configuration of different parameters such as, levels of memory, clock frequency, read/write latency and data width.
- A novel spatial processor architecture template that can be configured at design-time (based on the trade-off from the framework) and allows a faster implementation of an application-specific hardware.
- Case-study demonstrating the effectiveness of the framework for a custom processor design using state-of-the-art MRAM and SRAM for memory system.

The rest of this paper is organized as follows. We present the μ-Genie and describe its modules in section II. We describe our novel spatial processor architecture in section III. We conclude the paper with three case studies demonstrating the capability of μ-Genie.

II. Framework

The framework presented in this work allows the user to **customize** the different building blocks to be used in the design of the custom processor and memory system, **explore** custom architectures automatically generated for a given application and **estimate** area, power and latency of each of the architectures. Figure 2 gives an overview of the different steps in the framework. The framework takes two inputs, the *Configuration Parameters* - described in Section II-D - and an

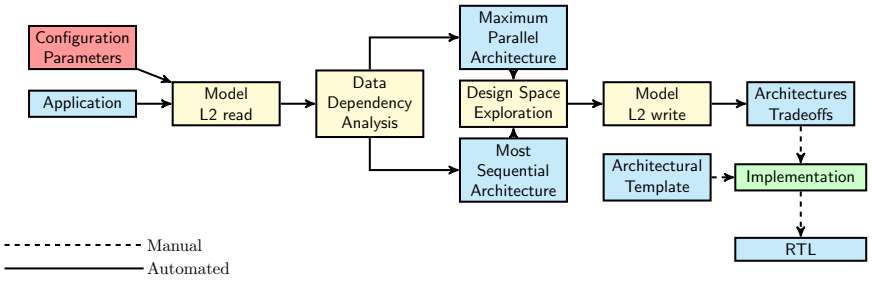


Fig. 2: μ-Genie Framework.

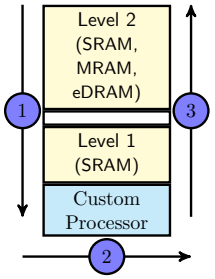


Fig. 3: The system under analysis.

Application - detailed in Section II-C. A set of hardware architectures, behaviorally equivalent to the input application, is automatically generated by the framwork and the area, power and latency trade-off is generated. The generated hardware architectures can be used to generate an RTL implementation using the *architectural templates* described in Section III. The framework is composed of three main steps. The first step - *L2 Model* - models the transfer of the input data between the *Level 2* memory and the *Level 1* memory. The second step, *Data Dependency Analysis* performs static analysis on the input application. The last step, *Design Space Exploration*, uses the information extracted by the previous step to procedurally generate different hardware architectures. The rest of this section will describe more in detail the different inputs and steps of the framework.

A. Model of Execution

The system architecture we assume in this work is composed of two levels of memory and a custom processor, as shown in Figure 3. The first level of memory - *Level 1* - smaller in size and uses SRAM as it needs to be physically close to the processor for faster access. The second layer - *Level 2* - larger in size and can be any memory technology (on-chip or off-chip) with diffferent access latency for read and write operations. Note that the processor and *Level 1* memory runs at a different clock speed than the *Level 2* memory. We assume a model of execution, as illustrated in Figure 3, by the three steps. Initially, all of the required input data for one iteration of the application are available at the *Level 2* memory. In the first step, part of the input data is transfered to the custom processor using the *Level 1* memory as intermediate storage location. In the second step, the computation is peformed in the processor and the output elements are stored in the *Level 1* memory. Once the computation is complete, in the third step, the output data stored in the *Level 1* memory is transfered back to the *Level 2* memory. Note that our model of execution performs all the three steps in a pipelined manner.

B. Level 2 Memory Model

Since the *Level 2* memory has longer access latency, we model them assuming the data is accessed in larger bursts. A read or write burst access to the *Level 2* memory 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 transfered in sequence from the start address to the end address, to the *Level 1* memory if a read access is being performed or to the *Level 2* memory for a write access.

C. Application

The framework performs exact data dependency analysis on the input application. This entails that the behaviour of the application needs to be completely specified at compile-time and independent from the input data. Streaming applications meet these requirement and can be analyzed by our framework. The input application language that our framework currently supports is C/C++, however the framework uses the LLVM Intermediate Representation [17], so it can easily be extended to support other languages as well.

D. Configuration Parameters

The second input to the framework is a configuration file that contains description of the different building blocks to be used in the realization of the hardware architecture. Through this file, the user can specify - different compute units - e.g. multipliers, adders, - process technology to be used - e.g. 16nm, 28nm - the clock frequency of processor & Level 1 memory and Level 2 memory. Moreover, the user can specify the datawidth used by the compute units and the Level 1 and Level 2 memories. Information requred to model the Level 2 burst accesses are as well specified in this file: the setup latency for write/read accesses, the type of Level 2 memory to be used -e.g. MRAM, SRAM etc..- and the size of the Level 2 memory. 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.

E. Level 2 Read

The first operation performed by the framework is to compute the transfer time of the application's input data from the *Level 2* memory to the *Level 1* memory. An address in Level 2 memory is given to the each input element used by the application; different datastructures are placed in consecutive memory addresses. The entire data transfer is modeled as a single burst read operation to the *Level 2* memory. The information required to compute the arrival clock time of each input element to the Level 1 memory is extracted from teh

Configuration Parameters and performing static analysis on the input *Application*. Using this information, the exact clock at which each input elements arrives in the Level 1 memory can be computed as:

$$AClk_i = SL_r + RL_{L2} * (Add_{L2i} + 1) * \frac{B_{L1}}{B_{L2}} * \frac{Clk_{L1}}{Clk_{L2}}$$

Where $AClk_i$ is the clock at which element i arrives to the Level 1 memory, SL_r is the setup latency of a Level 2 burst read, RL_{L2} is the Level 2 read latency - i.e. latency of a single read operation - expressed in number of *Level 2* clock cycles, Add_{L2i} is the offset of element i from the beginning of the burst access, B_{L1} is the data bitwidth of the Level 1 memory, B_{L2} is the data bitwidth of the Level 2 memory, Clk_{L1} is the clock frequency of the Level 1 memory and Clk_{L2} is the clock speed of the Level 2 memory.

F. Data Dependency Analysis

The *Data Dependency Analysis* module performs three main operations. First, it extracts *Data Dependency Graph* (DDG) from the application. Then, it schedules the DDG using the As Soon As Possible (ASAP) and As Late As Possible (ALAP) methodologies. Finally, it maps instructions in the DDG to hardware components - or Functional Units (FUs) - using a modified version of the *Interval Partitioning* algorithm [18]. 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 transformation there will be no control flow instructions in the application body and each variable will be defined only once. Following the chain of use and definition of the variables is then possible to produce a Data Dependency Graph like the one shown in Figure 4. We further process the obtained DDG in the aim to reduce the length of the path between the input nodes and the output nodes. The reason is that the length of such paths is equivalent to the number of sequential operations required to obtain the outputs which determines the latency of the application. Taking advantage of the associativity of some operation we can transform a long sequence of operations - like the one highlighted in Figure 4 - into an equivalent shorter tree.

In the second step we apply the ASAP and ALAP scheduling methodologies to the generated DDG. These schedules will associate each node of the DDG to a clock cycle where the instruction is executed. We start by scheduling the input nodes of the DDG using the arrival clock time of their element, computed as explained in Section II-E. This allows us to take into account the transfer time between the Level 2 memory and Level 1 memory. We then derive 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 a clock is assigned to the output leaf nodes of the DDG we can perform the ALAP scheduling: starting from

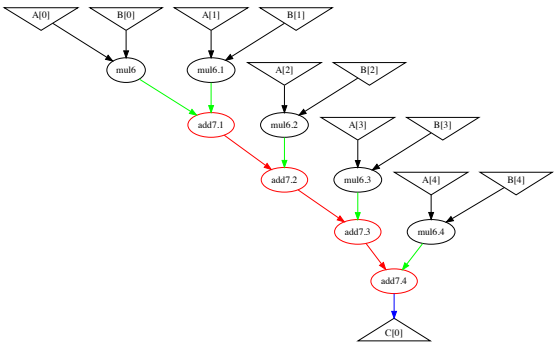


Fig. 4: Example of Data Dependency Graph. The inverse triangles represent the input data, obtained from the *load* instructions. The ovals describe operations on data, while the triangle triangle at the bottom represents the result, derived from a *store* instruction.

the output leaf nodes each dependency node is scheduled as late as possible. After this second step, each node will then have an associated ASAP schedule and ALAP schedule. The difference between these two schedules is called *mobility* of the node. The mobility of a node identifies an interval of clocks in which the instruction can be scheduled without changing the overall latency of the application. The final step of the *Data Dependency Analysis* module - described in Section II-G - will allocate the DDG nodes to FU, leveraging the nodes mobility to minimize the number of FUs of the final hardware architecture.

G. Modified Interval Partitioning

To generate an hardware architecture behaviorally equivalent to the input application and with the latency identified during the ASAP-ALAP scheduling, there are two main requirements. The first is that each instruction needs to be computed within its ASAP-ALAP interval. The second requirement is that instructions in the original DDG which are executed by the same FU cannot be scheduled at the same time. Our Modified Interval Partitioning algorithm - based on the original greedy Interval Partitioning algorithm [18] - is able to generate hardware architectures from a DDG meeting the two requirements described above. 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 map the Interval Partitioning formulation to the allocation of instructions to FU, we need to consider application instructions as jobs and FUs as resources. There are however three main differences between our problem and the canonical Interval Partitioning. First, while in the original Interval Partitioning problems jobs are equivalent to each other and can be assigned to any resource, the instructions represent computations. For example a multiply instruction can only be assigned to a multiply FU. To address this issue, we divide our initial problem and we perform multiple time the

Interval Partitioning, once per operation type. This ensures a correct allocation of the instruction to FUs performing the same operation. The second difference is due to the *mobility* of our instructions. Each instruction does not have a predefined start and end time, like a job has in the original Interval Partitioning, but it has a time interval - derived with ASAP-ALAP - in which it can be scheduled and a fixed latency. Hence, the start time of an instruction might vary, but once this is fixed, its end time can be derived. The modified Interval Partitioning takes the mobility of an instruction into account allowing a given instruction to start at any time within its allowed interval. The last difference regards the dependencies between instructions. While the original jobs in the Interval Partitioning are intependent from each other, the instructions of an application need to be executed according to their dependencies. This is addressed by ensuring that a given instruction is allocated after its dependencies are allocated and by verifying that its starting time is scheduled after the ending time of its dependencies. Listing 1 shows the pseudocode of our modified interval partitioning algorithm.

```
FunctionalUnits=[]
sort i in Instructions by ASAP[i]
for each i in Instructions
  allocated = False
  schedule[i] = ASAP[i]
  for d in dep(i)
    end_time_d = schedule[d] + latency(d)
    schedule[i] = max(schedule[i],end_time_d)
  for each fu in FunctionalUnits
    if type(fu) ==  type(i)
      if ALAP[i] >=  next_free_slot[fu]
        fu += [i]
        schedule[i] = max(schedule[i],
          next_free_slot[fu])
        next_free_slot[fu] = schedule[i] +
          latency(i)
        allocated = True
  if not allocated
    create new Functional Unit fu
    type(fu) = type(i)
    fu += [i]
    next_free_slot[fu] = schedule[i] + latency(i)
    FunctionalUnits += [fu]
```

Listing 1: Modified Interval Partitioning Algorithm

We assume that ASAP and ALAP schedules have been previously performed, hence we can have two datastructures that return the ASAP and ALAP schedules for a given instruction i - i.e. ASAP[i] and ALAP[i]. Instructions is an ordered list of instructions initialized with the instructions of the input application. A Functional Unit is represented as a set containing the instructions that have been allocated to it, and FunctionalUnits is a set containing the Functional Units of the resulting architecture which is initialized as an empty set. The functions latency and dep take an instruction i as input and return respectively the latency of i and a list of intructions i depends on. The function type takes as input a Functional Unit or an Instruction and returns the type of operation performed. The Modified Interval Partitioning algortihms returns the FunctionalUnits and schedule datastructure. FunctionalUnits contains a set of Functional Units which define the generated architecture, each Functional Unit is a set that contains the instrucitons to be executed. The schedule datastructure will contain the clock at which each instruction of the input application has to be executed.

H. Maximum Parallel Architecture and Most Sequential Architecture

We refer to output hardware architecture obtained after the *Data Dependency Analysis* module as **Maximum Parallel Architecture**. This architecture will take full advantage of the parallellism of the application and will perform the computation with the minimal possible latency. However, the Maximum Parallel Architecture will use the maximum number of functional units - in the worst case scenario equivalent to the number of instructions in the application - and it will hence have the worst area. At the other end of the spectrum of architectures we can imagine the **Most Sequential Architecture**, where no parallelism is used and the instruction are scheduled sequentially respecting their dependencies. This architecture will have the worst possible latency, however the minimal impact in area - using only one Functional Unit per operation type. Probably none of these two architectures will be of direct interest for the user as they represent two extreme cases. The architecture that are most likely to be interesting are the ones between the Maximum Parallel and Most Sequential that offer tradeoffs between power, latency and area. Section II-I describes how these intermediate architecures can be procedurally generated.

I. Design Space Exploration

The *Design Space Exploration* module procedurally generates hardware architectures, behaviorally equivalent to the input application, which exibit area, latancy and power tradeoffs. The DSE performs an iterative process and outputs, at the end of each iteration, a different hardware architecture. The iterative process start from the **Maximum Parallel Architecture** and ends when the **Most Sequential Architecture** is generated. An iteration consists of three steps. First, the instructions corresponding to output leaf nodes in the DDG are selected. The ALAP schedule of these iterations is increased by 1. After, the ALAP scheduling of the rest of the nodes in the DDG is updated accordingly. Doing so, the mobility of each instruction node is increased by one. The last step of the DSE consists in performing the Modified Interval Partitioning using the new ALAP schedule. Due to the increased mobility of each instruction the generated architecture is likely to use less Functional Unit. The process stops as soon as one iteration generates the **Most Sequential Architecture** which can be recognised because it contains only one Functional Unit per operation type. There are two side benefit of our DSE approach. The first is that it can be tuned. By changing the value that is used to increase the ASAP latency - which is one by default - we can tradeoff the speed of the DSE process for the accuracy. The second benefit is that each iteration is independent from the others hence the DSE process can be easily parallelized.

J. Level 2 Write

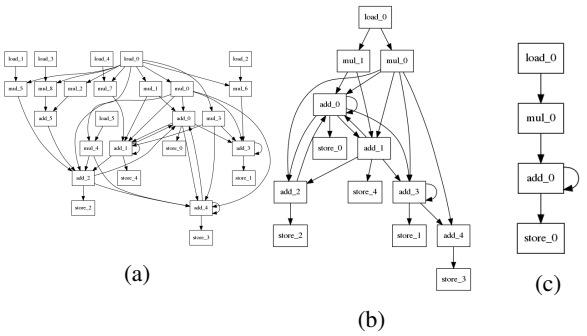
The schedule produced by the Modified Interval Partitioning algorithm already takes into account the transfer of the input data between the Level 2 memory and the Level 1 memory as explained in II-E. However, it does not include the transfer of the output data from the Level 1 memory to the Level 2 memory - which represents the end of the computation (see II-A). But, the result of the scheduling process determines the clock cycle at which the computation is over and the last output is stored in Level 1 memory. The transfer of the outputs to the Level 2 memory can therefore start right after the Custom Processor generates the last output. We model the transfer back to Level 2 as a burst write access, using the formula below we can derive the overall latency.

$$L2WBL = wSL + L2WL * O * \frac{L2B}{L1B} * \frac{L1Clk}{L2Clk}$$

Where $L2WBL$ is the Level 2 Write Back Latency, wSL is the setup latency of a Level 2 burst write, $L2WL$ is the Level 2 write latency - i.e. latency of a single write operation - expressed in number of L2 clock cycles, O is the total number of output elements,$L2B$ is the data bitwidth of the Level 2 memory, $L1B$ is the data bitwidth of the Level 1 memory, $L1Clk$ is the clock frequency of the Level 1 memory and $L2Clk$ is the clock speed of the Level 2 memory.

K. Architecture Tradeoffs

For a given input application and a configuration the framework outputs a set of hardware architectures having different area and latency tradeoffs. Figure 5 show three examples of architecture generated during the DSE. Each box represents a FU; The FUs generated from load and store instructions of the DDG (see Figure 4) will become separate memory banks of the Level 1 memory. The remaining FU are obtained from operation nodes of the DDG. We allow the architectures to have cycles as the circuit is synchronous and every instruction has been carefully scheduled. A self-loop in a FU idendifies data reuse, section III clarifies how this functionality can be achieved.



(a) (b) (c)

Fig. 5: Example of architectures generated from a matrix vector multiply application of size 5x5. The Maximal Parallel Architecture (a), an intermediate architecture (b) and the Most Sequential Architecture (c).

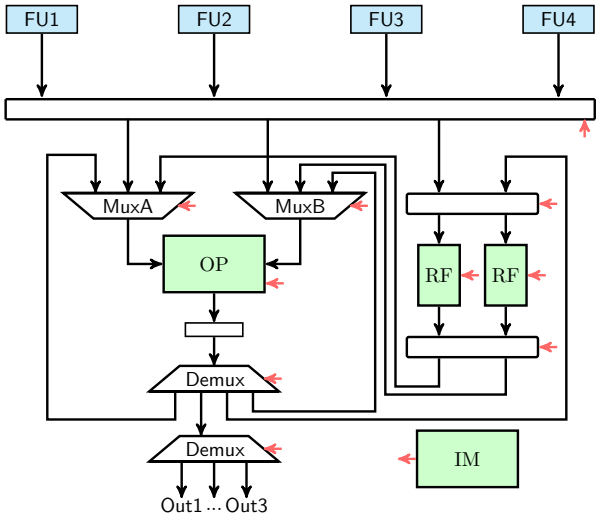


Fig. 6: Functional Unit template. The blocks at the top of the diagram represent other FUs that generate the inputs data. IM is the Instruction Memory, an internal memory where the FU stores the operation to perform. RFs are internal Register Files that the FU needs to reuse data and store inputs that needs to be used in the future. OP is the hardware unit actually performing the FU operation.

III. Architectural Template

To implement in hardware the architectures generated by μ-Genie as a spatial processor, we designed and implemented a FU template, shown in Figure 6. Each FU has an Instruction Memory (IM) where the operation to be performed at each clock cycle are stored. To compress the size of the IM, each instruction is labeled with the clock cycle in which it should be executed, an internal clock counter is compared to the label to decide when to issue the instruction. The RFs are internal Register Files used to store input data that needs to be processed in the future, as well as output data that need to be reused. The rectangles in the diagram represent configurable crossbars, these can send data from any of its input port to any of its output ports. OP is the hardware unit that performs the FU operation - e.g. add, multiply. This implementation allows any FU to be independed from the rest of the architecture - given that the instructions to perform are stored in the IM. The inputs generated by other FUs and the output generated by the same FU in the previous clock cycle can be used directly as operand or they can be stored in the RFs for future use. We have used our FU template to verify the correctness of some representative architectures through VHDL simulations.

IV. Case Studies

In this section we demostrate the capabilities of μ-Genie using three case studies. We selected as representative input application a matrix-vector multiplication having dimension 5x5,10x10 and 15x15 and a matrix-matrix multiplication having dimensions 10x10. We used TSMC 28nm target technology library for generating the database containing the area usage and power consumption of the differeng building blocks required as an input by our framework. In our experiments, we

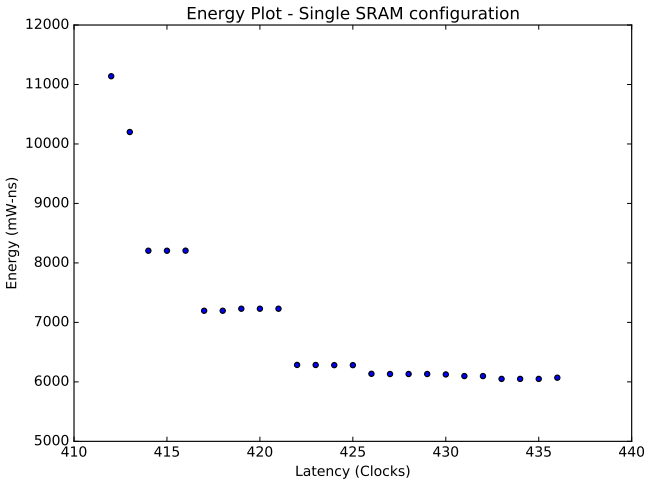


Fig. 7: Energy over Latency in clock cycles generated from a single configuration of a matrix vector multiplication of size 5x5. Each point corresponds to a generated hardware architecture. This configuration uses an SRAM Level 2 memory clocked at 350MHz, while the SRAM Level 1 memory and the Custom Processor are clocked at 1GHz.

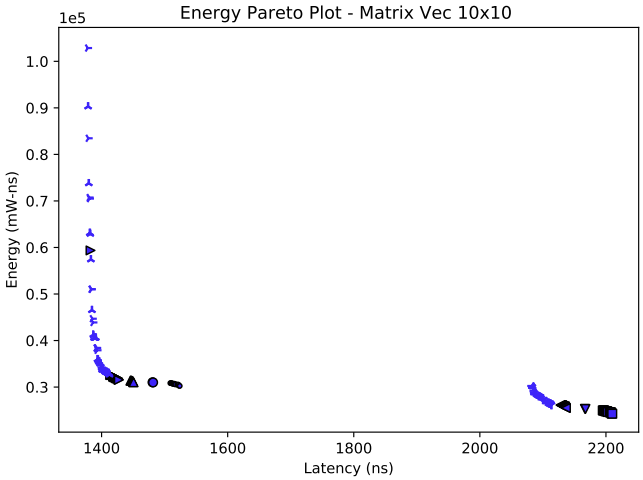


Fig. 8: Energy Pareto optimal hardware architectures generated by μ-Genie for a Matrix Vector multiplication 10x10, each point corresponds to an architecture generated by the framework.

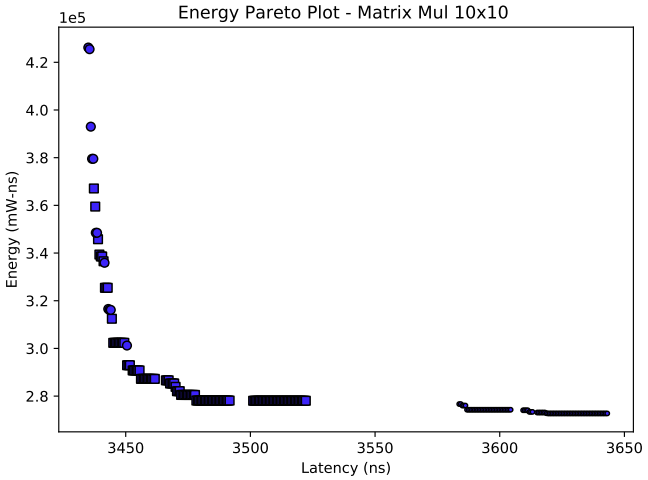


Fig. 9: Energy Pareto optimal hardware architectures generated by μ-Genie for a Matrix Vector multiplication 10x10, each point corresponds to an architecture generated by the framework. The different shapes identify different input configurations: we compare two memory technologies at the Level 2 memory, MRAM and SRAM, clocked at 350MHz, while the SRAM Level 1 memory and the custom processor have clock frequencies ranging from 400MHz to 2GHz with steps of 200MHz. In both figures we can identify two clusters, one has higher energy consumption but lower latency, while the other has higher latency but lower latency consumption. These represent configurations using respectively SRAM and MRAM technology at the Level 2 memory.

analyse the architectures generated by μ-Genie and we derive the energy consumption and latency of each design. We use the information contained in the input *Configuration Parameters* - Section II-D - to derive the dynamic and static energy consumed by each FU. Each generated architecture has a known latency imposed in each iteration of the DSE(Section II-I), which we use to compute its static energy consumption. Moreover, after applying the Modified Interval Partitioning algorithm (see Section II-G), the instructions performed by each FU are known and this information is used to compute the dynamic energy consumption of an architecture. In the first case study - Section IV-A - we show the result of the DSE methodology presented in Section II-I using a matrix vector multiplication with dimension 5x5 with a single configuration. The second case study compares the use of MRAM - modeled according to [19] - at the Level 2 memory against the use of SRAM (both in 28nm) for the two input applications. The last case study compares different matrix dimensions for the Matrix Vector application, 5x5, 10x10 and 15x15.

A. *Single SRAM configuration*

In this case study we used a 5x5 matrix vector multiplication as input application and we show the energy consumption of the hardware architectures generated by μ-Genie from a single configuration. The configuration uses SRAM in both levels and the Level 2 memory is clocked at 350MHz, while the Level 1 and Custom Processor are clocked at 1GHz. Figure 7 shows the energy consumption and latency of the results. Each point represents a generated hardware architecture. The Maximum Parallel Architecture - starting point of our DSE - is the fastest architecture completing the computation in 412 clock cycles of the Custom Processor and it consumes 11.1^3 mW-ns. The Most Sequential architecture - ending point of the DSE - is the slowest, compleating in 436 clock cycles and consuming 6^3 mW-ns. Between these two extreme design points, the DSE generates intermediate architectures with various energy-latency tradeoffs.

B. *MRAM vs SRAM layer 2 memory*

In this case study we compare a Level 2 MRAM memory against Level 2 SRAM memory. We used two input application

a Matrix Vector 10x10, shown in Figure 8 and a matrix multiply 10x10, in Figure 9. Each point is relative to a hardware architecture generated by μ-Genie. The different shapes identify different input configurations: we compare two memory technologies at the Level 2 memory, MRAM and SRAM, clocked at 350MHz, while the SRAM Level 1 memory and the processor have clock frequencies ranging from 400MHz to 2GHz with steps of 200MHz. In both figures we can identify two clusters, one has higher energy consumption but lower latency, while the other has higher latency but lower latency consumption. These represent configurations using respectively SRAM and MRAM technology at the Level 2 memory. In the case of the matrix vector application (Figure 8) the fastest architecture - using SRAM memory - has a latency of 1375ns and consumes over 1^5mW-ns. The most energy efficient SRAM architecture has instead a latency of 1525ns consuming under 0.3^5mW-ns, begin 3x more energy efficient then the fastest with a 10% increase in latency. The most energy efficient architecture using MRAM technology has instead a latency of 2210ns and consumes $0,24^5$mw-ns, hence having 45% higher latency than the best SRAM counterpart, with 25% decrease in power consumption. The matrix multiplication, Figure 9, performs 10 times more operation than the matrix vector multiplication, hence there is a clear overall increase in latency - about 30% - and energy consumption - about 4 times - in comparison to the previous application. In this case the architecture using SRAM consuming the least amount of energy is has 2% higher latency compared to the fastest one, but consumes 50% less energy. However, the introduction of MRAM technology in the Level 2 memory 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.

C. Different Matrix Vector Dimensions

In this case study we compare three different dimensions of a matrix vector multiplication: 5x5, 10x10 and 15x15, using the same configurations used in IV-B . Figure 10 shows the pareto-optimal architectures generated from each input application. The increase in the matrix size is reflected by an increase in latency; the result obtained by the 5x5 matrix application have latencies below 1000ns, the ones relative to the 10x10 have latencies above 1000ns and below 2500ns, lastly the 15x15 architecture have latencies above 2500ns. The increased number of operations results instead in wider tradeoffs possibilities. The normalized latency gap between the most energy efficent SRAM and MRAM, decreases with the matrix size, with an 82% increase for the 5x5, 42% for the 10x10 and 32% for the 15x15. The reduction in energy consumption between the same pair of results is instead 20% for the 5x5 and 10x10, while for the 15x15 drops to 16%. Therefore, as the size of the matrix grows the benefits in energy consumption of using MRAM technology in the Level 2 memory reduce. This is due to the increase in the amount of write operations that have high energy impact when the MRAM technology is used.

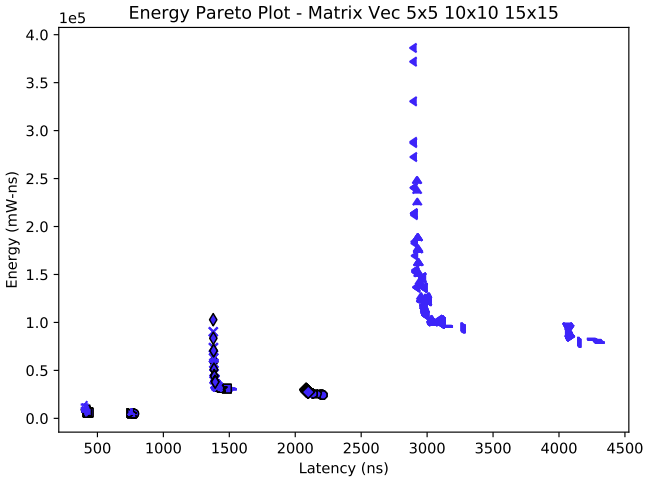


Fig. 10: Energy Pareto optimal architectures generated by μ-Genie for different sizes of Matrix Vector multiplication 5x5 - with latencies ranging from 0 to 1000,10x10 - having latencies between 1000ns and 2500ns , and 15x15- with latencyies above 2500ns. Each point corresponds to an architecture generated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

V. Conclusion and Future Work

We presented μ-Genie a framwork for the design space exploration of memory-aware custom processors. Our framework allows to automatically design an application-specific processor using two levels of memories. The framework enables the design space exploration of different hardware technologies and *co-designs* the memory system and the custom processor. We have empirically demonstrated the implementability of the generated architecture using our functional unit hardware templates. Lastly, we have shown the capabilities of the framework using three case studies, where we where able to obtain quantitative insights over the use of the MRAM technology against SRAM for a use case applcation. We are currently working on automating the genration of the RTL implementation of the architectures generated by μ-Genie using the presented hardware templates. In the near future, we plan to enanche our framework adding the capability of automatically *merging* multiple application-specific architectures, generating *multi-application application-specifc hardwares*.

References

[1] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *2011 38th Annual international symposium on computer architecture (ISCA)*. IEEE, 2011, pp. 365–376.

[2] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 37–47, 2010.

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

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

[5] Cadence Design Systems, Inc., "Tensilica customizable processors," in *https://ip.cadence.com/ipportfolio/tensilica-ip/xtensa-customizable*, 2019.

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

[7] P. Meloni, S. Pomata, G. Tuveri, S. Secchi, L. Raffo, and M. Lindwer, "Enabling Fast ASIP Design Space Exploration: An FPGA-based Runtime Reconfigurable Prototyper," *VLSI Des.*, vol. 2012, pp. 11:11–11:11, Jan. 2012. [Online]. Available: http://dx.doi.org/10.1155/2012/580584

[8] J. Eusse *et al.*, "Pre-architectural performance estimation for ASIP design based on abstract processor models," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*, July 2014, pp. 133–140.

[9] L. Jozwiak *et al.*, "ASAM: Automatic architecture synthesis and application mapping," *Microprocessors and Microsystems*, vol. 37, no. 8 PARTC, pp. 1002–1019, 2013.

[10] K. Karuri, R. Leupers, G. Ascheid, and H. Meyr, "A Generic Design Flow for Application Specific Processor Customization through Instruction-Set Extensions (ISEs)," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, K. Bertels, N. Dimopoulos, C. Silvano, and S. Wong, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 204–214.

[11] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen, "A novel architecture of the 3D stacked MRAM L2 cache for CMPs," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, Feb 2009, pp. 239–249.

[12] M. P. Komalan, C. Tenllado, J. I. G. Pérez, F. T. Fernández, and F. Catthoor, "System level exploration of a stt-mram based level 1 data-cache," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2015, pp. 1311–1316.

[13] S. Lee, J. Jung, and C. Kyung, "Hybrid cache architecture replacing sram cache with future memory technology," in *2012 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2012, pp. 2481–2484.

[14] 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

[15] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "ShiDianNao: Shifting vision processing closer to the sensor," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, June 2015, pp. 92–104.

[16] Y. Chen, T. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, June 2019.

[17] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

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

[19] Q. Dong, Z. Wang, J. Lim, Y. Zhang, Y. Shih, Y. Chih, J. Chang, D. Blaauw, and D. Sylvester, "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 International Solid - State Circuits Conference - (ISSCC)*, Feb 2018, pp. 480–482.