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

1st Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

2nd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

3rd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

***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 being handled by the processors, the memory system is becoming the critical component that determines the overall performance and power consumption. Since the memory system and the processing system are strongly interdependent, they must be co-designed. State-of-the-art tool flow for custom processor design-space exploration focus mainly on processor architecture optimization and 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 designing a custom-processor architecture using different memory technologies, such as MRAM and SRAM.**

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 ADDCITATION 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 ADDCITATION.

Computer Aided Design (CAD) tools are available to help with the increasing complexity of hardware design, and increase the productivity of hardware designers. 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 the processor architecture optimization in terms of latency, area usage and power consumption [1]–[5]. The memory system is then added as a final step according to the processor architecture requirements, as illustrated in Figure 1. On the other hand, prior-art that performs co-optimization of processor and memory system (including emerging memories such as MRAM, eDRAM, PCM, RRAM [6]) perform co-simulation of the processor and the memory system and optimizes the cache replacement policies [7]–[14]. Spatial custom processor architecutres [15], [16] are proposed to distribute the instruction or program memory and achieve maximum performance, however, they are not optimized for a limited set of applications.

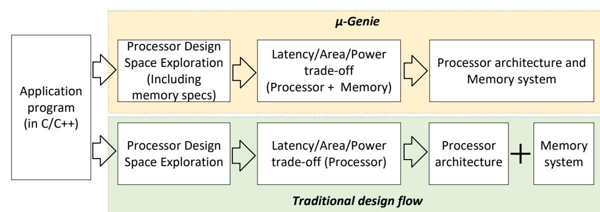

Fig. 1. Difference between state-of-the-art and the proposed μ-Genie design flow custom processor architecture design and implementation.

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 in terms of performance, power consumption and area usage. Since the memory system and the processing system are *interdependent*, they should be *co-designed*. This is especially important while using emerging memory technologies with different access latencies for read and write operations but offers high integration density and often low power than SRAM.

Our main contributions in this paper are:

- μ-Genie: An automated framework for memory-aware custom processor architecture design-space exploration and provide different area/latency tradeoffs, as shown in Figure 1. The framework allows use of different kind of memory technologies and configuration of different parameters such as, levels of memory, clock speed, read-/write latency and data width.
- A novel spatial processor architecture template that can be configured at design-time 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 ..

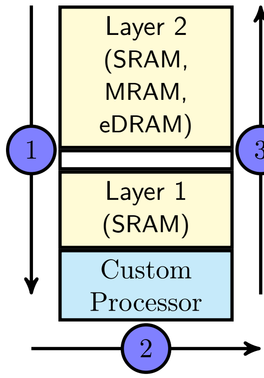

Fig. 2. The system under analysis. Composed by two layers of memory, the memory at layer two can use various technologies while the memory at layer one uses only SRAM technology.

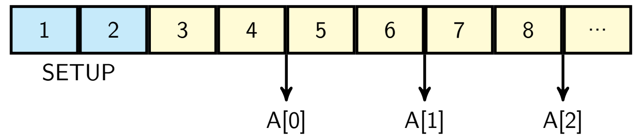

Fig. 3. Example of Layer 2 memory burst access.

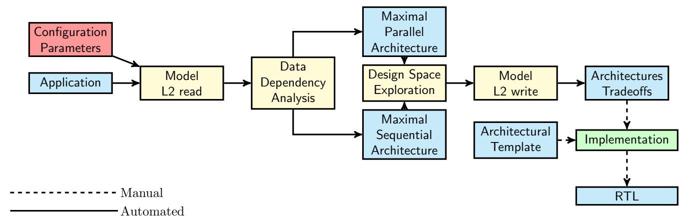

Fig. 4. Diagram of the Framework presented in this work.

II. Background

This section will contain the relevant background. Consider the following items:

- Application specific processor
- Non-Volatile Memories
- Static Data-Dependency Analysis
- System Under Analysis

A. System Under Analysis

The system architecture we use in this work is composed of two layers of memory and a *Custom Processor*. Taking advantage of the possibility to produce stacked chip having layers produced using different manifacturing technologies we are able to explore different memory implementations at the *Layer 2* of the memory architecture. The first layer of memory - *Layer 1* - is instead fabricated on the same chip layer as the Custom Processor and uses SRAMs memories. The diagram in Figure 2 shows a representation of the entire computing system. The computation proceed as described in Figure 2. At the beginning of the computation we assume that all of the required input data are present at the *Layer 2* of the memory hierarchy. In the first stage of the computation the input data are transfered to the *Custom Processor* using the *Layer 1* memory as intermediate storage location. The actual computation is peformed in the second stage where each output element is stored in the *Layer 1* memory. Once the computation is compleate, in the third step, the output data stored in the *Layer 1* memory is transfered to the *Layer 2* memory. At the end of the three stages of the computation the result will be stored in the *Layer 2* memory.

B. Layer 2 Memory Model

We model the *Layer 2* memory as accessible using burst accesses. Figure BLA shows a representation of such access. A read or write access to the *Layer 2* memory is controlled by a Direct Memory Access (DMA) controller.Starting address and size of the burst are given as input to the DMA, which perform the burst memory access. After an initial setup time the accessed elements are transfered in sequence from the starting address to the ending address to the *Layer 1* memory if a read access is being berformed, or to the *Layer 2* memory if a write access is being performed.

III. Framework

The framwork presented in this work allows the user to **customize** the elements to be used in the design of the *Custom Processor* and *Memory System*, **explore** custom architectures automatically generated for a given application and **predict** area, power and latency of such architectures. Figure 4 gives an overview of the modules contained in the framework. The framework takes two inputs, the *Configuration Parameters* - described in III-B - and an *Application* - detailed in III-A. A set of hardware architectures, behaviorally equivalent to the input application, is automatically generated by the framwork. The output is the area, power and latency tradeoff of such architectures. The generated hardware architectures can be used to generate an RTL implementation using the *architectural templates* described in IV. The framework is composed by three modules. The first module - *L2 Model* - models the transfer of the input data between the *Layer 2* memory and the *Layer 1* memory. The *Data Dependency Analysis* performs static analysis on the input application. The last module, *Design Space Exploration*, uses the information extracted by the previous step to procedurally generate hardware architectures. The rest of this section will describe more in detail the inputs and modules of the framework.

A. Application

The framework performs exact data dependency analysis on the 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. Listing 1 shows the application source code that will be used for the rest of this paper. The language that the framework currently supports is C/C++, however the framework uses the LLVM Intermediate Representation CITATION NEEDED, so the support for other languages can be easily added.

```c
void matrix_vec_kernel(int *A,int *B, int *C){
    int sum;
    for(int i=0;i<DIM2;i++){
        sum=0;
        for(int j=0;j<DIM1;j++){
            sum+=A[i*DIM1+j]*B[j];
        }
        C[i]=sum;
    }
}
```

Listing 1. Example of input application, C implementation of a matrix vector multiplication.

B. Configuration Parameters

The second input to the framework is a configuration file. The Configuration file contains a description of the different component to be used in the realization of the hardware architecture, an example is shown in Listing 2. Using this file the user can specify the process technology to be used - e.g. 16nm, 28nm - the clock frequency of Layer 1 memory, Layer 2 memory and Custom Processor. The user can moreover specify the datawidth used by the different operators - e.g. multipliers, adders-, and by the Layer 1 and Layer 2 memory. Detailed information requred to model the Layer 2 burst accesses are as well contained in this file: the setup latency for write/read accesses, the type of Layer 2 memory to be used -e.g. MRAM, SRAM etc..- and the size of the Layer 2 memory. **TODO**: describe the L1 memory model (linear model that extrapolate area/power/energy using the depth as a variable to fill in missing values in the database)

```json
{
   "resource_database": {
  "technology": 16,
  "clock_frequency": 1000,
  "bitwidth_adder": 128,
  "bitwidth_multiplier": 64,
  "bitwidth_register_file": 128,
  "type_l2": "tt1v1v85c",
  "technology_l2": 16,
  "clock_l2": 800,
  "bitwidth_l2": 32,
  "depth_l2":2048,
  "setup_write_latency_l2":2,
  "setup_read_latency_l2":2
   }
}
```

Listing 2. Example of input configuration file

C. Layer 2 Read Model

The first operation performed by the framework is to compute the transfer time of the application input data from the *Layer 2* memory to the *Later 1* memory. An address in Layer 2 memory is given to the 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 *Layer 2* memory (see BACKGROUND). The information required to compute the arrival clock time of each input element to the Layer 1 memory is extracted from teh *Configuration Parameters* and performing static analysis on the input *Application*. Using such information the exact clock at which each input elements arrives in the Layer 1 memory can be computed with:

$$ACLk_i = rSL + L2RL * (L2Add_i + 1) * \frac{L1B}{L2B} * \frac{L1Clk}{L2Clk}$$

Where $ACLk_i$ is the clock at which element i arrives to the Layer 1 memory, rSL is the setup latency of a Layer 2 burst read, $L2RL$ is the Layer 2 read latency - i.e. latency of a single read operation - expressed in number of L2 clock cycles, $L2Add_i$ is the offset of element i from the beginning of the burst access, $L1B$ is the data bitwidth of the Layer 1 memory, $L2B$ is the data bitwidth of the Layer 2 memory, $L1Clk$ is the clock frequency of the Layer 1 memory and $L2Clk$ is the clock speed of the Layer 2 memory.

D. 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 ADD-CITATION. 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 application variables is then possible to produce a Data Dependency Graph like the one shown in Figure ADDIMAGE. We further process the obtained DDG in the aim to reduce the lenght of the path between the input nodes and the output nodes. The reason is that the lenght of such paths is equivalent to the number of successive operations required to obtain the outputs which is connected to the latency of the application. Taking advantage of the commutativity of some operation we can transform a long sequence of operations into an equivalent shorter tree as shown in Figure ADDIMAGE. 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 III-C. This allows us to take into account the transfer time between the Layer 2 memory and Layer 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 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 III-E - will allocate the DDG nodes to FU, using the nodes mobility to minimize the number of FUs of the final hardware architecture.

E. 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 CITE - 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 3 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 3. 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 - e.i. 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.

F. Maximal Parallel Architecture and Maximal Sequential Architecture

We refer to output hardware architecture obtained after the *Data Dependency Analysis* module as **Maximal 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 Maximal 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 **Maximal 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 Maximal Parallel and Maximal Sequential that offer tradeoffs between power, latency and area. Section III-G describes how these intermediate architecures can be procedurally generated.

G. 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 **Maximal Parallel Architecture** and ends when the **Maximal 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 **Maximal 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.

H. Layer 2 Write Model

The schedule produced by the Modified Interval Partitioning algorithm already takes into account the transfer of the input data between the Layer 2 memory and the Layer 1 memory as explained in III-C. However, it does not include the transfer of the output data from the Layer 1 memory to the Layer 2 memory - which represents the end of our 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 Layer 1 memory. The transfer of the outputs to the Layer 2 memory can therefore start right after the the Custom Processor generates the last output. We model the transfer back to Layer 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 Layer 2 Write Back Latency, wSL is the setup latency of a Layer 2 burst write, $L2WL$ is the Layer 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 Layer 2 memory, $L1B$ is the data bitwidth of the Layer 1 memory, $L1Clk$ is the clock frequency of the Layer 1 memory and $L2Clk$ is the clock speed of the Layer 2 memory.

I. Architecture Tradeoffs

IV. Architectural Template

This section will describe the details of the architectural template and how this can be used to implement the system architecture generated by the framework.
Possible subsections:

- System Level Architecture
- Processing Element template Architecture

V. Experiments

In this section we demostrate the capabilities of μ-Genie using four 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. In our experiments we 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* - III-B - 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(III-G), which we use to compute its static energy consumption. Moreover, after applying the Modified Interval Partitioning algorithm (see III-E), 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 experiment - V-A - we show the result of the DSE methodology presented in section III-G. The second experiments compares the use of MRAM memory - modeled according to [17] - at the Level 2 memory against the use of SRAM - TSMC tt0p9v85c for the two input applications. The last experiment compares different matrix dimensions for the Matrix Vector application, 5x5, 10x10 and 15x15.

A. Single SRAM configuration

In this experiment 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 memories at all memory levels, the Level 2 memory is clocked at 350MHz, while the Level 1 and Custom Processor are clocked at 1GHz. Figure 5 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

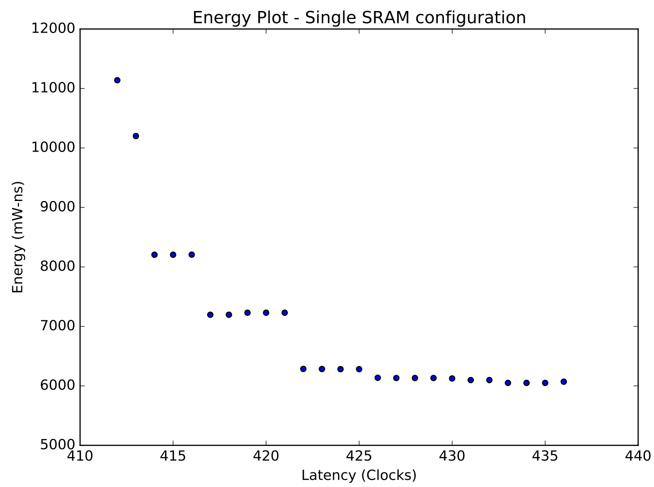

Fig. 5. Energy over Latency in clock cycles generated from a single configuration of a matrix vector multiplication of size 5x5. Each point corresponds to a generated hardware architecture. This configuration uses an SRAM Level 2 memory clocked at 350MHz, while the SRAM Level 1 memory and the Custom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

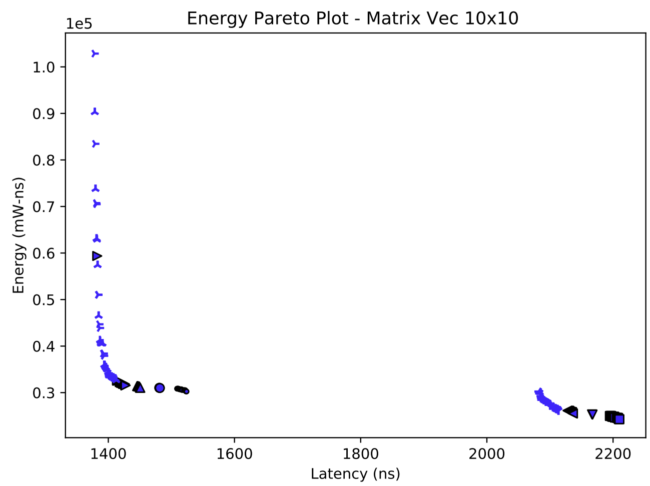

Fig. 6. Energy Pareto optimal hardware architectures generated by μ-Genie for a Matrix Vector multiplication 10x10, each point corresponds to an architecture generated by the framework.

- 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 offering various energy-latency tradeoffs.

B. MRAM vs SRAM layer 2 memory

In this experiment we use μ-Genie to compare two Level 2 MRAM memory against Level 2 SRAM memory. We used two input application a Matrix Vector 10x10, shown in Figure 6 and a matrix multiply 10x10, in Figure 7. 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 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. In the case of the matrix vector application -Figure 6 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 7, 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 3x - 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.

- The most energy efficient designs are obtained using the highest possible clock for the Layer 2 SRAM memory (1GHz).
- The architecture consuming less energy has a Layer 2

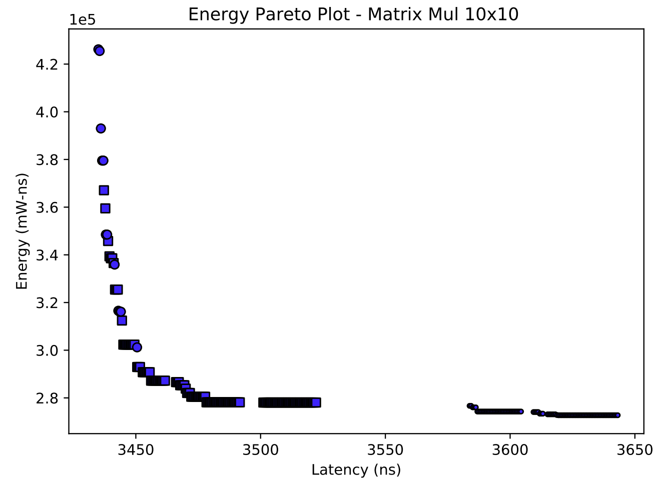

Fig. 7. Energy Pareto optimal hardware architectures generated by μ-Genie for a Matrix Vector multiplication 10x10, each point corresponds to an architecture generated by the framework.

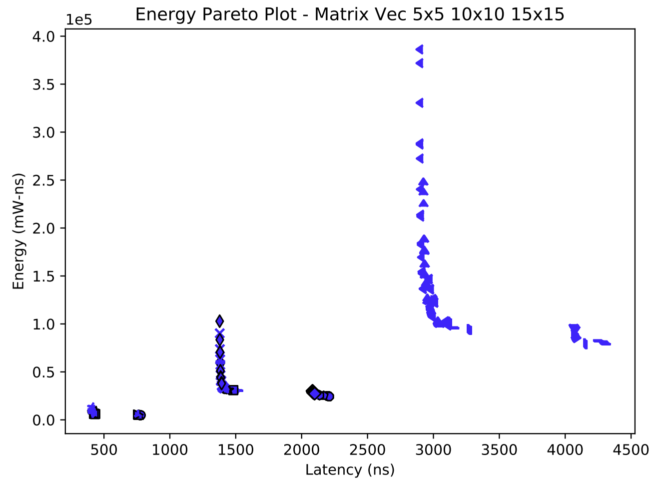

Fig. 8. Energy Pareto optimal hardware architectures generated by μ-Geniefor a Matrix Vector multiplication 10x10, each point corresponds to an architecture generated by the framework.

clock frequency of 1GHz and a Layer 1 clock frequency of 1600 MHz.

C. Different Matrix Vector Dimensions

VI. Conclusion

Here we will draw the conclusion from the framework and the experiment as well as highlight possible future work.

References

[1] 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

[2] R. Jordans, L. Jwiak, and H. Corporaal, “Instruction-set architecture exploration of VLIW ASIPs using a genetic algorithm,” in *2014 3rd Mediterranean Conference on Embedded Computing (MECO)*, June 2014, pp. 32–35.

[3] 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.

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

[5] 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.

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

[7] 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.

[8] M. P. Komalan, C. Tenllado, J. I. G. Prez, F. T. Fernndez, 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.

[9] 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.

[10] G. Prenat, K. Jabeur, P. Vanhauwaert, G. D. Pendina, F. Oboril, R. Bishnoi, M. Ebrahimi, N. Lamard, O. Boulle, K. Garello, J. Langer, B. Ocker, M. Cyrille, P. Gambardella, M. Tahoori, and G. Gaudin, “Ultra-fast and high-reliability sot-mram: From cache replacement to normally-off computing,” *IEEE Transactions on Multi-Scale Computing Systems*, vol. 2, no. 1, pp. 49–60, Jan 2016.

[11] J. Zhang, M. Kwon, C. Park, M. Jung, and S. Kim, “ROSS: A design of read-oriented stt-mram storage for energy-efficient non-uniform cache architecture,” in *4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 16)*. Savannah, GA: USENIX Association, Nov. 2016. [Online]. Available: https://www.usenix.org/conference/inflow16/workshop-program/presentation/zhang

[12] R. Patel, X. Guo, Q. Guo, E. Ipek, and E. G. Friedman, “Reducing switching latency and energy in stt-mram caches with field-assisted writing,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, pp. 129–138, 2016.

[13] M. Komalan, J. I. G. Pérez, C. Tenllado, P. Raghavan, M. Hartmann, and F. Catthoor, “Feasibility exploration of nvm based i-cache through mshr enhancements,” in *Proceedings of the Conference on Design, Automation & Test in Europe*, ser. DATE ’14. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2014, pp. 21:1–21:6. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616606.2616632

[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] 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.