Design Space Exploration of Embedded SoC Architectures for Real-Time Optimal Control

Kris Shengjun Dong, Dima Nikiforov, Widyadewi Soedarmadji, Minh Nguyen, Christopher Fletcher, Yakun Sophia Shao

***Abstract*—Empowering resource-limited robots to execute computationally intensive tasks such as locomotion and manipulation is challenging. This project provides a comprehensive design space exploration to determine optimal hardware computation architectures suitable for model-based control algorithms. We profile and optimize representative architectural designs across general-purpose scalar, vector processors, and specialized accelerators. Specifically, we compare scalar CPUs, vector machines, and domain-specialized accelerators using kernel-level benchmarks and end-to-end representative robotic workloads. Our exploration provides a quantitative performance, area, and utilization comparison and analyzes the trade-offs between these representative distinct architectural designs. We demonstrate that architectural modifications, software, and system optimization can alleviate bottlenecks and enhance utilization. Finally, we propose a code generation flow to simplify the engineering work for mapping robotic workloads to specialized architectures.**

I. Introduction

Robots are increasingly being deployed across diverse fields such as package delivery, high-precision surgical assistance, and autonomous industrial robots for warehouse assistance. Robots run a broad spectrum of algorithms and workloads to function effectively, including reasoning, sensing, perception, localization, forward and inverse dynamics, mapping, motion planning, and control. As robots evolve towards more complex rigid dynamics with multi-joint systems, this introduces additional challenges for control algorithms.

Model predictive control (MPC) is a widely used approach for controlling highly dynamic robotic systems subject to complex constraints. However, its computation demand grows cubically with the robot's state space and linearly with the prediction horizon. This challenge becomes insurmountable on embedded SoCs as robots evolve into highly dynamic multi-joint rigid body systems. Additionally, control algorithms must run at a frequency of 100-400Hz to meet real-time deadlines in safety-critical applications. Ensuring low-latency computation is crucial in safety-critical applications such as obstacle avoidance and navigation, where every millisecond of processing time can be critical.

Traditional microcontrollers and embedded SoCs either lack the computational capacity to meet such latency requirements or consume excessive energy thus becoming impractical for battery-powered edge devices. As robots become more complex and miniaturized, onboard resources, such as compute power, energy, memory storage, and bandwidth, become increasingly constrained. Furthermore, the combination of computation-demanding algorithms, real-time requirements, and resource constraints makes it increasingly impractical to rely on a single hardware architecture to meet all demands on an embedded SoC.

Recognizing the heterogeneity of modern SoCs, this report explores hardware architectures suitable for modern robotic systems by profiling MPC algorithms on embedded platforms as an example. Specifically, we conduct a comprehensive design space exploration across scalar, vector, and matrix architectures as depicted in Figure 1. Our exploration provides a quantitative performance, area, and utilization comparison and analyzes the trade-offs between these architectural designs. Finally, we propose a code generation flow to simplify the engineering work for mapping robotic workloads to specialized architectures.

II. Overview of Architectures

CPUs are the most commonly used general-purpose computing platforms. Since CPUs are designed to handle a broad range of kernels, cheap CPUs, especially microcontrollers, were traditionally used for simple rule-based robotic kernels such as proportional-integral-derivative (PID) control solvers. However, microcontrollers fail to meet the computational demand for real-time robotic tasks with scaled-up models and DNN-based algorithms. This is mainly due to microcontrollers being typically slow with a compute frequency of 200 MHz. One solution to maximize a CPU's available compute is by exploiting the out-of-order and superscalar architecture to drive instruction-level parallelism. However, the performance margin comes with the cost of area overhead and power consumption. An out-of-order and superscalar CPU can achieve up to 100 GFLOP/Sec. However, its power efficiency is less than 1 GOP/J, which is usually not compatible with power-constrained embedded devices [24].

Vector extensions allow CPUs with lightweight front-ends to achieve similar performance as out-of-order and superscalar CPUs, while dramatically simplifying implementation by eliminating excess control logic. Vector machines exploit data parallelism by processing multiple data elements with a single instruction without adding too much power and area overhead. Various vector architectures range from traditional long-vector machines, as presented in Cray-style vector machines [19] to Packed SIMD (P-SIMD) machines that utilize VLIW instruction encodings, [12], to contemporary novel short-vector machines [26]. These architectures adeptly support a broad array of operations, such as scalar multiplication, general vector multiplication (GEMV), and matrix-matrix multiplication (GEMM). While vector cores are highly flexible and can handle a wide range of computations, they

may not be as efficient for GEMM as dedicated systolic arrays or other matrix-specific accelerators, especially in handling large, batched operations. This is because vector machines scale inadequately with a large datapath compared to a systolic array-style processor. As datapath length (DLEN) scales up to feed in data sufficiently to the increased number of lanes, the bandwidth of the register file must scale accordingly.

GPUs are designed to drive data-level parallelism (DLP) and thread-level parallelism (TLP) with up to over 100 thousand cores running simultaneously, enabling massive parallelism. Only a few robotic workloads [5], [15] can be benefited from such massive parallelism. However, a typical GPU can reach up to 10 TOPS/Sec of performance. Its high throughput makes it suitable for tasks that involve heavy matrix computations, graphics rendering, and other non-real-time robotic workloads. GPUs mainly exhibit TLP by running thousands of threads simultaneously to handle extensive graphical calculations and data processing tasks efficiently. However, a conventional GPU's die area is typically around 600 mm^2 to over 800 mm^2 [7], which consumes too much area to be placed on micro- or nano-robotics where the system has limited space on a chip and faces weight constraints. In terms of energy consumption, GPUs consume 100W-350W of power [25], which is far beyond the power budget of most resource-limited robotic systems.

In addition to the general-purpose processors, application-specific integrated circuits (ASICs), and field-programmable gate arrays (FPGAs) have recently been proposed as feasible platforms for power-constraint robotic applications since they offer higher energy efficiency than CPUs and GPUs. Both ASICs and FPGAs consume relatively less power and are often integrated into small systems with limited memory. Exemplified by their reconfigurability and programmability, FPGAs adapt well to the involving robotic algorithms compared to ASICs. Partial Reconfiguration (PR) extends this flexibility, allowing dynamic reconfiguration of a portion of the FPGA at runtime [22]. However, the highly specialized programming model, challenges in mapping irregular algorithms with control flow, and high resource overhead from configuration/routing logic make maximizing the utilization of all processing elements (PEs) challenging [23].

With the integration of domain-specific accelerators, ASICs, and FPGAs can parallelize computations while leveraging the specific characteristics of the kernels to eliminate excess logic and simplify implementation [22] and offload specific computational tasks to the accelerators. Research [13], [16], [23] has been devoted to developing hardware-aware algorithms for FPGAs and has been proposed to achieve software-equivalent performance, making ASICs and FPGAs capable of meeting real-time requirements [14], [20], [21].

Particularly, spatial array style domain-specific accelerators have been attributed to their energy efficiency and compute throughput in compute-intensive workloads such as neural networks. A typical spatial array accelerator architecture can achieve an efficiency of approximately 34.4 GOP/J while consuming 278 mW at a frame rate of 35 FPS for the AlexNet model [4]. The benefit of re-purposing deep neural network (DNN) accelerators for robot computing is based on the observation that neural network accelerators are optimized for matrix linear algebra [9], [11] while major robotic algorithms also mainly consist of DNN workloads and small-sized GEMM and GEMV [2], [8]. Most robotics chips already contain an ASIC neural net inference accelerator to speed up DNN-based workloads like perception, vision, and online learning. Suppose it can be shown that the same accelerator is also capable of speeding up robot dynamics computations. In that case, the overall SoC architecture can benefit from significant savings in the chip area, make room for other specialized units, or instead enable a physical design that is more cost-effective for ASIC implementation.

Besides DNN accelerators, research efforts [5], [13]–[17], [20], [21] have been dedicated to design domain-specific robotic accelerators exclusively for robotic algorithms, focusing on a particular kernel defined for optimization problems for specific types of robots. Although domain-specific accelerators can offer improved efficiency over general-purpose processors on specific robotics workloads, the dramatic amount of engineering cost due to its specific programmable interface with its dedicated domain-specific language (DSL); Besides, due to the limitation of its fixed functionality, certain domain-specific accelerators usually fail short on other types of operations as robotic algorithm evolving. Besides, the life span of such accelerators can be relatively short compared to the general-purpose approach since certain accelerators can be obsoleted fairly fast when they fail to keep up with the fast-evolving algorithms.

From a high-level perspective, we observe that each architecture offers distinct advantages in terms of computational efficiency by leveraging various forms of parallelism: data-level, instruction-level, and thread-level parallelism to achieve the compute efficiency required by specific types of robotic tasks with different area and power costs. Moving forward, our focus will be on three principal architectural categories: general-purpose CPUs, vector machines, and domain-specific accelerators. These architectures are strategically selected to meet the diverse computational needs of robotic systems within stringent area and power constraints.

III. Methodology

This initial analysis in Section II allows us to narrow our focus to the following architecture categories: general-purpose CPUs, vector machines, and domain-specific accelerators. To understand the tradeoff of these architectures, we profile representative architecture candidates from each categories1: For general-purpose CPUs, we evaluate RISC-V CPU cores, including simple in-order core Rocket [1], superscalar in-order core Shuttle [18], superscalar out-of-order BOOM core [3], and more performant out-of-order cores such as MediumBOOM, LargeBOOM and MegaBOOM core with different numbers of floating point unit (FPUs) [27]. For vector machines, we assess Saturn, a vector extension on RISC-V

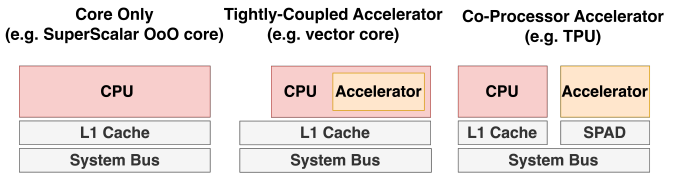

Fig. 1. Categories of Target Hardware Architectures to be Profiled and Evaluated: CPU, tightly integrated accelerators, and decoupled co-processors

CPU with short-vector microarchitecture [26]. For domain-specific accelerators, we profile a state-of-the-art systolic array Gemmini [9].

One distinct attribute of these architectures is their integration styles, ranging from standalone processing units (e.g. CPUs) to tightly integrated systems (e.g. a CPU with vector extensions) to loosely coupled systems (e.g. a CPU with Gemmini). This segment of the study highlights the significant performance enhancements through software optimization and architectural changes. Saturn is a compact design of a vector machine tightly integrated into the CPU. On the other hand, Gemmini integrates with a RISC-V CPU using a decoupled Rocket custom coprocessor (RoCC) interface, which provides the flexibility of allowing CPU and processing units to work concurrently. The reorder buffer in Gemmini decouples the instruction flow latency from the actual execution, allowing for latency hiding.

Furthermore, we assess the impact of the architectural spatial structure among candidate architectures, which is the physical layout of processing units, from scalar and 1D vector cores to complex 2D systolic arrays. we analyze the capabilities of systolic arrays in handling multi-dimensional data structures, which are prevalent in advanced computational tasks such as matrix multiplication. The efficiency of these arrays in leveraging spatial characteristics is weighed against their potential limitations in processing higher-dimensional data.

IV. Workload Characterization

MPC algorithms are computationally intensive. MPC incorporates dense linear algebra kernels like GEMV and GEMM, and domain-specific operations such as Cholesky decomposition and Riccati recursion, which further increase the computational load. Specifically, we profile TinyMPC, a state-of-the-art embedded MPC algorithm as an example, and measure the impact of hardware architectures on kernel performance and end-to-end workloads.

A. Workload Structure

One state-of-the-art approach to solving MPC is the alternating direction method of multipliers (ADMM). ADMM alternates between updating the primal variables, slack variables, and dual variables, ensuring convergence to an optimal solution by gradually reducing constraint violations; with each subroutine denoted as *primal update*, *dual update*, *slack update*. TinyMPC redefines the most computationally expensive step, the *primal update*, as a linear quadratic regulator (LQR)

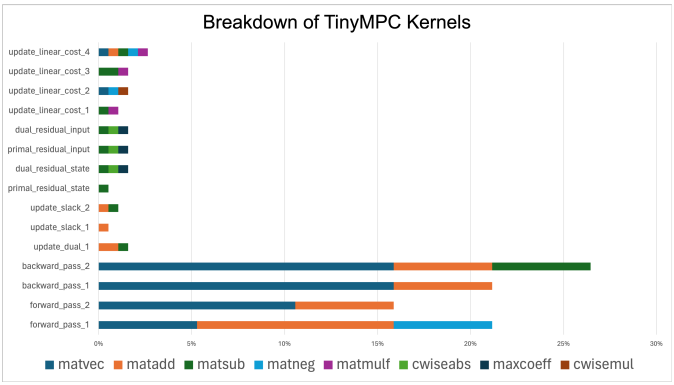

Fig. 2. Kernel Breakdown of TinyMPC

problem. LQR equations are solved using *Riccati recursion*, where the feedback gain matrix K and feedforward term d are computed for each time step. Over sufficiently long horizons, the Riccati recursion converges to an infinite-horizon solution, represented by a single gain matrix K_∞ and cost-to-go matrix P_∞. TinyMPC optimizes memory use by caching these two matrices, avoiding the need to store the entire horizon of intermediate K and P matrices. During each ADMM iteration, only the linear terms of the Riccati equation need updating, further accelerating the computation. Pre-computed matrix inverses and coefficients ensure the algorithm's most intensive operations are limited to matrix-vector products, enabling fast and memory-efficient execution suitable for constrained environments.

The functions of TinyMPC can be broadly classified into three categories: Iterative operations that have data dependencies, strip-mining operations on vectors, and global maximum reductions. These are shown in Algorithms 1, 2, and 3, respectively; with more detailed kernel breakdown in Figure 2.

Algorithm 1 Iterative Operations in TinyMPC

function FORWARD_PASS_1(i)
  $u[i] \leftarrow -Kinf \cdot x[i] - d[i]$
end function
function FORWARD_PASS_2(i)
  $x[i+1] \leftarrow Adyn \cdot x[i] + Bdyn \cdot u[i]$
end function
function BACKWARD_PASS_1(i)
  $d[i] \leftarrow Quu\_inv \cdot (Bdyn^\top \cdot p[i+1] + r[i])$
end function
function BACKWARD_PASS_2(i)
  $p[i] \leftarrow q[i] + AmBKt \cdot p[i+1] - Kinf^\top \cdot r[i]$
end function
function UPDATE_LINEAR_COST_4
  $p[N-1] \leftarrow -(Xref[N-1]^\top \cdot Pinf \cdot (vnew[N-1] - g[N-1]))$
end function

Algorithm 2 Stripmining Operations in TinyMPC

function UPDATE_SLACK_1
  $znew \leftarrow u + y$
  $znew \leftarrow \min(u\_max, \max(u\_min, znew))$
end function
function UPDATE_SLACK_2
  $vnew \leftarrow x + g$
  $vnew \leftarrow \min(x\_max, \max(x\_min, vnew))$
end function
function UPDATE_DUAL_1
  $y \leftarrow y + u - znew$
  $g \leftarrow g + x - vnew$
end function
function UPDATE_LINEAR_COST_1
  $r \leftarrow -rho \cdot (znew - y)$
end function
function UPDATE_LINEAR_COST_2
  $q \leftarrow -(Xref \cdot Q)$
end function
function UPDATE_LINEAR_COST_3
  $q \leftarrow q - rho \cdot (vnew - g)$
end function

Algorithm 3 Global Reduction in TinyMPC

function PRIMAL_RESIDUAL_STATE
  $primal\_residual\_state \leftarrow \max(\text{abs}(x - vnew))$
end function
function DUAL_RESIDUAL_STATE
  $dual\_residual\_state \leftarrow rho \cdot \max(\text{abs}(v - vnew))$
end function
function PRIMAL_RESIDUAL_INPUT
  $primal\_residual\_input \leftarrow \max(\text{abs}(u - znew))$
end function
function DUAL_RESIDUAL_INPUT
  $dual\_residual\_input \leftarrow rho \cdot \max(\text{abs}(z - znew))$
end function

B. Programming Interface and Libraries

To create a unified platform for comparing the performance of differing architectures for Robotic and DSP applications, we create a C library of commonly used operators, `matlib`[1]. `matlib` provides a lightweight interface to various linear algebra operations similar to Eigen [10], a C++ header-only library. `matlib` was used to write reference implementations for each backend; however, to achieve full performance, hand-tuned implementations required optimizations across the abstractions provided by these function definitions.

V. Hardware Software Co-optimization

Developing highly optimized software mapping for each hardware target is necessary. These optimizations ensure a clear comparison between different hardware architectures by isolating architectural characteristics from sub-optimal software implementations. This approach attempts to overcome the impact of other variables such as inefficient software mappings or overheads introduced by inefficiently written code, that might obscure the hardware's true capabilities. This approach ensures that any observed performance differences are directly attributable to the hardware's architectural design rather than extrinsic software factors.

A. Optimizations on Vector Core Saturn

To initially accelerate TinyMPC, we utilize a library-based approach. For every `matlib` function that is used within the solver, we write a vectorized implementation using RISC-V vector extension (RVV) intrinsic support from `gcc`. However, as shown in Figure 3, although the vectorized `matlib` code has meaningful speedup over scalar `matlib` code running on Rocket, highly optimized scalar code using Eigen still outperforms the Saturn implementation, necessitating further optimizations.

1) Software Loop Unrolling: One benefit of the RVV ISA is its ability to perform register grouping using the `LMUL` field. This performs register grouping and unrolls contiguous vector operations in hardware using a vector sequencer, theoretically eliminating the need for software loop unrolling. As depicted in Figure 4, utilizing `LMUL` improved the performance of stripmining operations in `matlib`. However, increasing `LMUL` harmed performance for the iterative components of TinyMPC, `backward_pass` and `forward_pass`. The major component was that these operations must to compute GEMV and vector addition operations serially and could not be mapped to larger vector registers due to dependencies. Furthermore, optimized GEMV kernels on Saturn require using the `vfmacc_vf` instruction, multiplying a scalar element of a vector against a column of a matrix. However, due to TinyMPC using vector dimensions of 4 and 12 for the iterative kernels, these could only be partially mapped to the 512-bit vector registers, further hindering the use of `LMUL`. Aggressive software loop unrolling allowed variation in the scalar elements across instructions, compensating for the absence of hardware unrolling.

Alternatively, we consider using vector summation using reduction operators (`vfred(o|u)sum`) to increase the utilization of vector lanes for wide matrices. However, Saturn currently implements vector reduction serially. Because of this, despite using fewer vector lanes, `vfmacc_vf` is better suited than vector reduction.

2) Operator Fusion: Another downside of using `matlib` based code for optimizing TinyMPC was the fact that the function boundaries prevented fusing operators across registers. Every time a `matlib` function is called, data are explicitly written back to memory using RVV store intrinsics and must be subsequently loaded back to registers in future invocations. This increases the instructions the frontend must serve to the vector units, and adds additional memory latency. Instead, we wrote optimized implementations of the TinyMPC functions, fusing `matlib` operators by keeping temporary values within

[1]https://github.com/ucb-bar/matlib

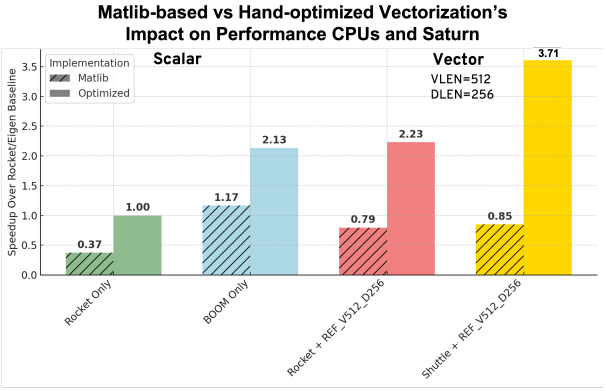

Fig. 3. Matlib-based vs hand-optimized vectorized TinyMPC

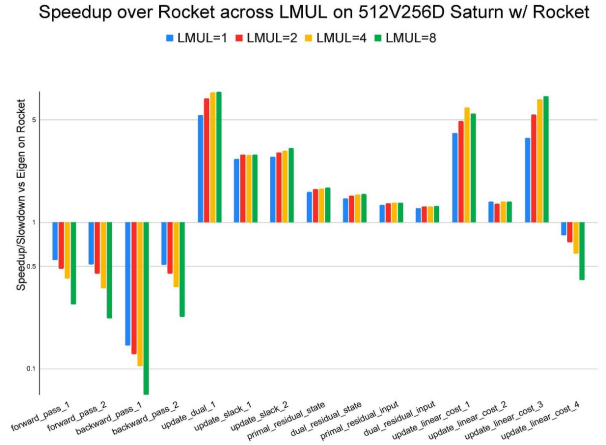

Fig. 4. Accelerating TinyMPC using Saturn with varying sizes of LMUL

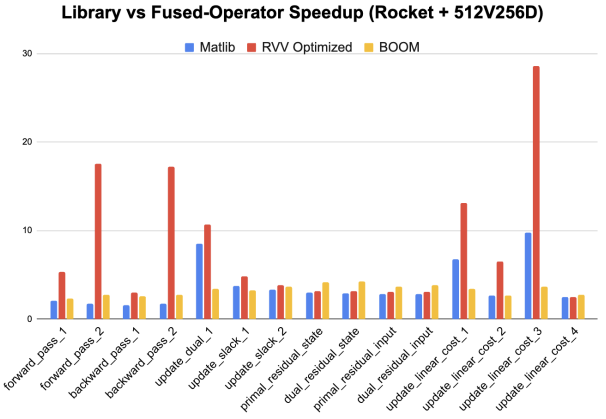

Fig. 5. Library vs Fused-Operator Speedup on Rocket-driven 512V256D Saturn

the vector registers until they need to be written back to the TinyMPC workspace. The impact of these optimizations, along with software unrolling, is depicted in Figure 5.

B. Optimizations on Systolic Array Gemmini

1) Build a Systolic Array that supports GEMV: We architect Gemmini's systolic array to support both GEMM and GEMV operations and effectively address the previously limited utilization of processing elements. This expansion of the systolic array change ensures a fair comparison between Saturn and Gemmini. The rationale is that Saturn, as a general proposed-vector machine, supports GEMV and GEMM well. However, Gemmini is facing the limitation of only 1/DIM utilization in GEMV operations since only one column of the mesh PEs will be utilized. For example, the utilization is approximately 25% for a mesh with a dimension of 4x4. The utilization of the systolic array decreases when the mesh size increases.

The primary difficulty with the matrix-vector operation $AB + D$ is the lack of reuse with the A matrix. No longer can we propagate A rightwards, as each processing element (PE) requires a new element every cycle. Given the desire for full utilization of the mesh, we first added additional scratchpad banks and modified the execute controller to fetch DIM $\times$ DIM elements of A in parallel. By providing mesh DIM^2 elements of A at every compute cycle, one for each of the PE in the systolic array. Next, to improve the reuse of B, we broadcast a singular element of B across all columns. This allows the columns to be independent of one another during GEMV operations. The changes allow all the PEs to be utilized to compute DIM^2 of A $\times$DIM elements of B for GEMV operation in the systolic array in Figure 6.

By re-architecting Gemmini, the modified GEMV Gemmini achieves more than a $4\times$ speedup in GEMV by achieving full utilization on such operation while maintaining its performance in GEMM. Another benefit is removing the delay across columns, as each input can be directly wired to a PE with a multiplexer. The trade-off is that the additional features come at a 2% area cost in a 4×4 mesh compared to Gemmini's original design. As depicted in Table II, an additional area is attributed to the extra scratchpad banks since our feature needs at least DIM + 1 banks to ensure all the necessary data for the mesh being loaded in one cycle. DIM for the elements of input A and 1 for the biases in weight stationary mode and weights (vector) B in output stationary mode.

The heatmap in Figure 8 illustrates the speedup achieved by our design compared to the original Gemmini design over a range of matrix and vector sizes. The x-axis represents the matrix width and the vector length (K), while the y-axis shows the matrix height (I). Each cell shows the speedup achieved for the given matrix and vector dimensions. The result demonstrates that the new GEMV/GEMM systolic array achieves a significant performance improvement over the traditional GEMM systolic array on various sizes of vector and matrix sizes. On average, the new architecture delivers on average ˜a

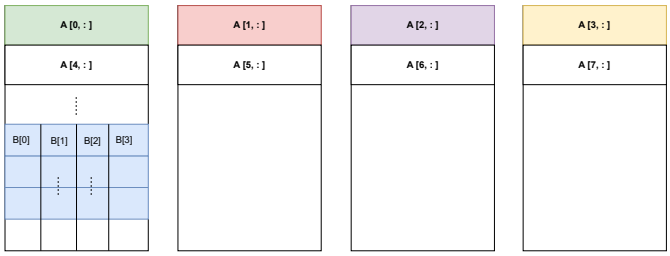

Fig. 6. Example DIM $= 4$ scratchpad setup with GEMV mesh. The rows of A are strided across DIM banks to allow for DIM $\times$ DIM elements to be accessed at once. The weight vector is stored consecutively in memory.

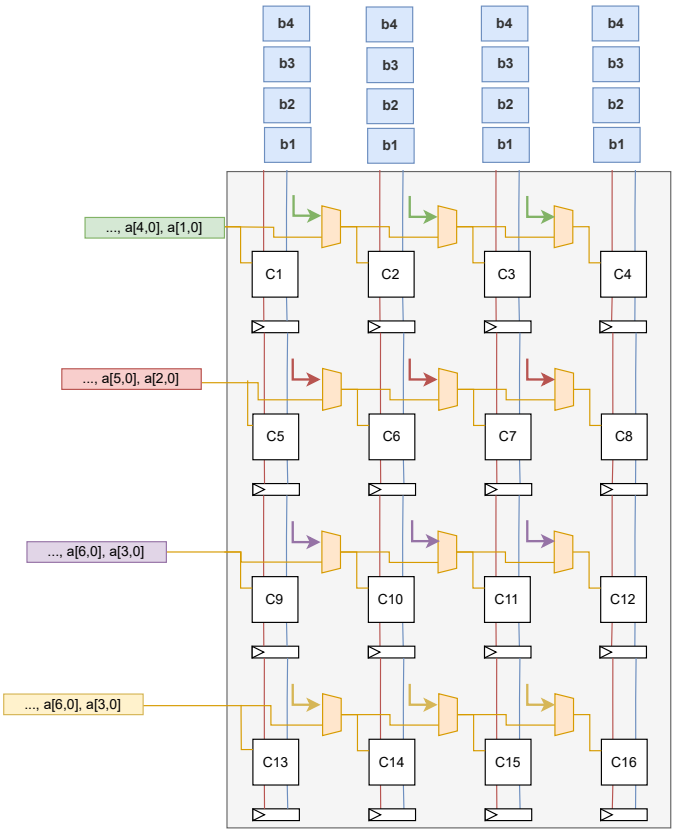

Fig. 7. A 4×4 example of the execute controller wiring the elements of A into the mesh PEs. It also duplicates the weight and bias vectors, depending on the dataflow, across the columns of the mesh.

6$\times$ speedup on GEMV operations while maintaining similar performance for GEMM.

2) Perform Software Optimization on Gemmini:

a) Static Mapping of Kernels: Static mapping of kernels involves pre-calculating and optimizing the allocation of data and tasks to various processing elements within the hardware. In scenarios involving fixed-size operations like those commonly found in MPC, dimensions, tiling, and indexing can be determined without dynamic memory allocation. This allows for the entire computational workflow to be streamlined, as all necessary instructions such as moving data in (i.e. `mvin`) and out (i.e. `mvout`) of scratchpad and computation (i.e. `execute`) can be prepared and optimized beforehand. This approach not only simplifies the coding and execution process but also significantly reduces the computational overhead associated with dynamically calculating these parameters during runtime.

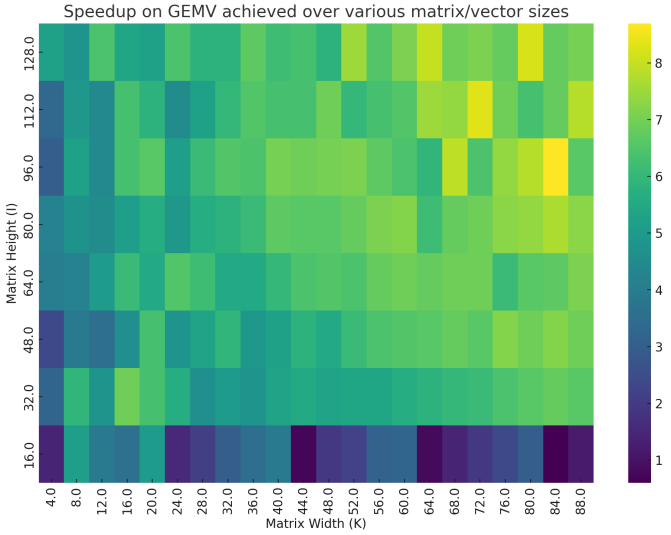

Fig. 8. Speedup on randomly generated GEMV operations achieved by GEMV fine-grained weight stationary workload on Gemmini driven by Rocket

b) Enhanced Instruction Throughput: By statically mapping kernels and unrolling loops, unnecessary CPU operations, which are typically used to calculate indexing and tiling on the fly, are eliminated. This removal of redundant calculations ensures that the accelerator is not left idle, thus avoiding poor utilization and wasted computational resources. Such optimizations enable a direct feed of optimized instructions to the accelerator, enhancing the instruction throughput and ensuring that the computational units are continuously active and productive.

c) Reduction of Redundant Operations: Beyond improving throughput, this method also involves the removal of redundant operations that do not contribute to the result of the computation. This includes unnecessary configuration commands, fencing operations that ensure data integrity but may be excessive, and redundant data movement of data already present in the system. By eliminating these operations, the efficiency of the execution process is further enhanced. Each instruction executed is necessary and directly contributes to the computational goal, minimizing the cycle time and improving the overall speed of computation.

d) Streamlined Execution and Reduced Latency: Static mapping of kernels addresses the overhead introduced by default software interfaces, which often include redundant and unnecessary instructions. We streamlined the computation process by leveraging statically known strides, tiling factors, and matrix/vector sizes, eliminating superfluous operations

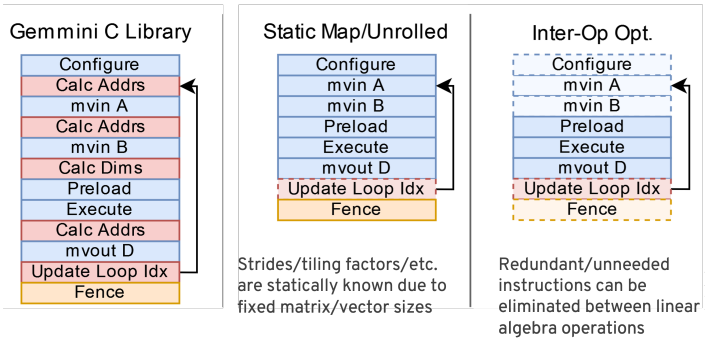

Fig. 9. Optimizing Gemmini software mapping with loop unrolling and static mapping

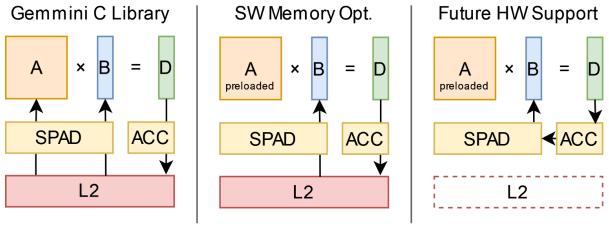

Fig. 10. Optimize on Gemmini's Memory Interface to Achieve Scratchpad-Resident

and significantly enhancing the execution efficiency between linear algebra operations. This method ensures that the kernels are optimally aligned with the hardware capabilities, reducing latency and improving throughput.

3) Use of Gemmini's Fine-Grained ISA: Gemmini's coarse-grained instruction set is necessary for most machine-learning applications to achieve high mesh utilization. However, coarse-grained instructions require multiple RoCC instructions for configuration, spending 5-7 instructions before beginning execution. Additionally, constructing these RoCC instructions can take multiple cycles due to the bit-shifting that needs to be performed by the scalar CPU to construct the RoCC arguments. For the matrix sizes found in TinyMPC, coarse-grained instructions only map to 2-6 fine-grained instructions, resulting in negligible benefit from hardware instruction sequencing. Furthermore, coarse-grained instructions require the input operands to be stored in memory, preventing additional optimizations from re-using data on Gemmini's scratchpad.

However, using the fine-grained ISA requires improved instruction throughput from the CPU, driving the RoCC interface to achieve high utilization of Gemmini. To address this, we aggressively unroll code and perform indexing and address calculation at compile-time using static mapping to minimize the overhead of constructing RoCC instructions.

4) Scratchpad-Resident Linear Algebra Operations: In order to support data-dependent operations on Gemmini, modifying the typical DNN kernel mapping for Gemmini was necessary. Typically, inputs to an operation are loaded into Gemmini's scratchpad before they are used as inputs to tiled operations, accumulating in Gemmini's accumulator memory. These results are then stored back in DRAM before being reused in subsequent operations. However, when performing operations that take several cycles on Gemmini, this overhead severely limits performance. In addition to the latency induced by data movement, explicit fence instructions must be inserted between Gemmini store and load instructions, as Gemmini's ROB does not track data RAW hazards across memory operations. In our experimentation, this can introduce up to 600 cycles of stalling upon a fence instruction within the iterative phase of TinyMPC.

To avoid this issue, we perform several optimizations to our mapping. First, we load all matrices used by TinyMPC onto the first bank of the scratchpad, in addition to several utility matrices, such as identity and negative identity matrices. Secondly, intermediate results computed by Gemmini are directly written to the scratchpad so that they can be immediately reused in subsequent operations. One downside of this approach is that writing to the scratchpad instead of the main memory prevents the use of Gemmini's output scaling pipeline, which can help perform fused scaling/GEMM operations. To compensate for this, we allocate additional utility matrices onto the scratchpad, which are commonly used scalar multiples of the identity matrix.

Although it is theoretically possible to perform most of TinyMPC's kernels fully resident on the scratchpad, we only perform this optimization for the iterative passes. The reason for this is when performing GEMV operations using Gemmini's original hardware, vectors must be stored in a single column of the destination scratchpad, resulting in inefficient loads when performing element-wide operations, as only one element is loaded per cycle. However, future architectural enhancements to Gemmini, such as hardware GEMV support, would allow vectors to be stored and packed across scratchpad rows, addressing this issue.

5) Output Stationary Dataflow and Elimination of Accumulator Memory: Storing the outputs of Gemmini's compute operations in the scratchpad prevents unnecessary data movement to memory. However, this also prevents the use of hardware accumulators to perform tiled operations on the same partial outputs. To address this, this work uses a Gemmini configuration with an output-stationary dataflow. Unlike machine learning workloads, there is not enough significant reuse of weights to benefit from a weight-stationary dataflow, and accumulating within the mesh's PEs eliminates the need for an explicit accumulator memory. Furthermore, although Gemmini does not support using coarse-grained instructions for the output stationary dataflow, since coarse-grained instructions are not used for TinyMPC, there is no drawback to using the output stationary dataflow from a programming interface perspective.

6) Activation Functions and Pooling: Although linear operations such as vector addition and subtraction, scalar multiplication, and GEMV can be performed solely using Gemmini's mesh, other operations such as absolute value and min/max cannot be performed. Absolute value is required to perform the

strip-mining operations in TinyMPC on Gemmini. However, as depicted in Equation 1, the absolute value can be implemented using ReLU, an activation function natively supported by Gemmini. This allows Gemmini to fully compute the strip-mining operations and utilizes Gemmini's ability to fuse activation functions with tiled operations on the mesh.

$$\text{abs}(x) = \text{ReLU}(x) + \text{ReLU}(-x) \tag{1}$$

Additionally, ReLU can also be used to clip a value to an upper or lower bound, which is required for updating slack variables in TinyMPC. However, these operations can be also performed using ReLU using Equations 2 and 3.

$$\text{clip}_{\text{low}}(x, \text{min}) = \text{ReLU}(x - \text{min}) + \text{min} \tag{2}$$

$$\text{clip}_{\text{high}}(x, \text{max}) = -\text{ReLU}(-x + \text{max}) + \text{max} \tag{3}$$

Finally, computing the residuals of TinyMPC requires calculating a global maximum across vectors of approximately 100 elements each. While this could potentially implemented using sequences of ReLU operations, computing a maximum between two arbitrary vectors would require over 5 operations, achieving comparable performance to simply using a scalar CPU. However, to avoid computing the entire maximum on the scalar core, Gemmini's hardware support for max-pooling and be utilized. By using a pool size of 2 when moving out to shared memory, Gemmini can perform a reduction across 4 scratchpad rows. This can potentially be increased by increasing the hardware pooling dimensions. Although this does not enable max reduction within a scratchpad row, this still reduces the max reduction needed to be performed on the CPU by a factor of 4.

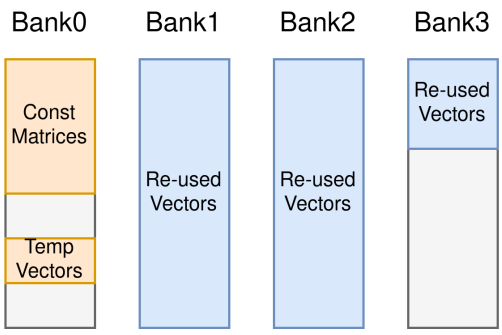

Fig. 11. Mapping of the TinyMPC Solver Workspace to Gemmini Scratchpad

7) Develop Fine-Grained Synchronization Interface : This optimization targeted the reduction of synchronization overheads in memory interfaces, which can significantly hamper the computational efficiency in parallel processing environments. By streamlining the synchronization processes and enhancing the communication protocols between CPU and Gemmini, the fence instruction can be reduced to avoid stall on the CPU side. This reduces the time wasted waiting for memory accesses and synchronization, improving the overall system responsiveness and throughput.

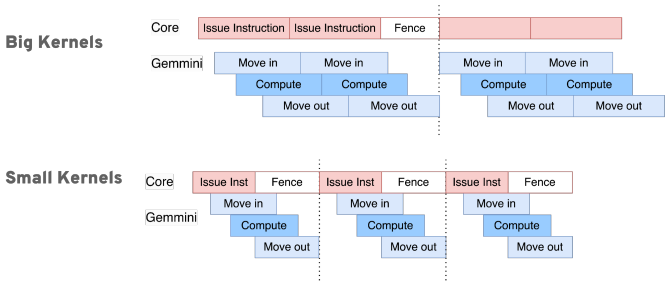

Fig. 12. challenge of Gemmini's Synchronization Interface

By integrating both software and hardware optimizations, we ensure the following evaluation reflects the strengths and limitations of each architectural design. This comprehensive approach also allows us to understand better how different architectures can be optimized and utilized for the complex, diverse demands of modern robotic and machine-learning applications.

VI. Evaluation

We conduct a detailed profiling and synthesis to explore the trade-offs between area, performance, and utilization of each backend. First, we carried out the comparison over standard benchmarks between RISC-V cores, vector core Saturn, and the systolic array, Gemmini, specifically GEMV and GEMM operations. Second, we evaluated each backend in the context of robotics applications with the target workloads. By synthesizing different architectural configurations, we understand the strengths and limitations of each approach under varying computational loads and analyze the trade-off between performance, area, and utilization across all designs.

A. Performance Evaluation

1) Kernel-level Evaluation: The heatmaps in Figure 13 and Figure 15 illustrate the performance patterns of the vector architecture and systolic array across various operations, highlighting each backend's strengths. These differences in performance reflect that each backend has its unique strength over the other with different operations.

Figure 13 demonstrates the speedup achieved by Saturn over the original Gemmini design for GEMV operations across a range of randomly generated matrix and vector sizes. Both backends are powered by a simple in-order RISC-V core, Rocket [1]. The x-axis represents the matrix width and vector length (K), while the y-axis indicates the matrix height (I). Each cell displays the relative speedup achieved for a given matrix-vector size combination. On average, Saturn achieves a speedup of approximately 2.78× for GEMV operations, as the original Gemmini architecture can only utilize a single column of its mesh for this workload.

After adding the GEMV support, Gemmini's performance on GEMV operations improved significantly by over 5× on average, which allows Gemmini's to outperform Saturn by 2.34× in various vector and matrix sizes. However, Saturn must perform matrix-vector operations using vector-scalar

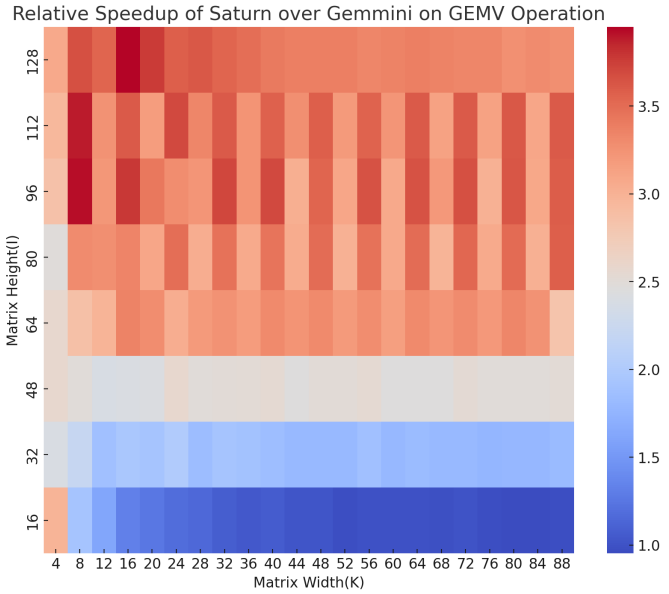

Fig. 13. Speedup Achieved by Saturn over Gemmini on Randomly Generated GEMV Operations

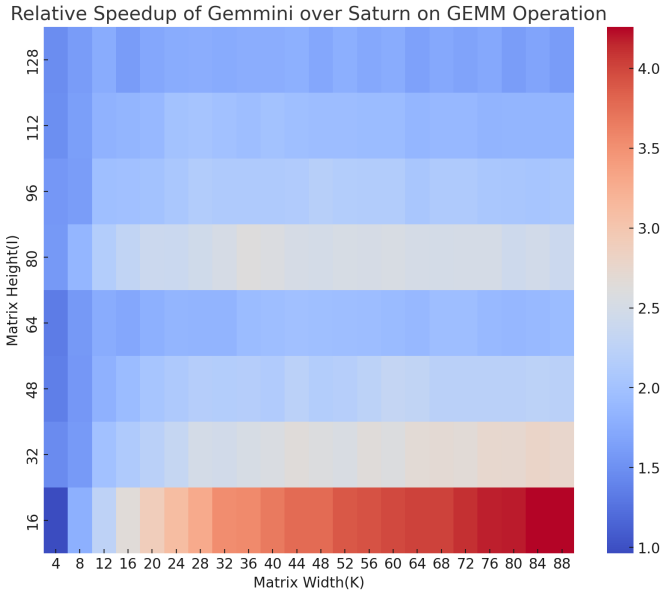

Fig. 15. Speedup Achieved by Saturn over Gemmini on Randomly Generated GEMM Operations

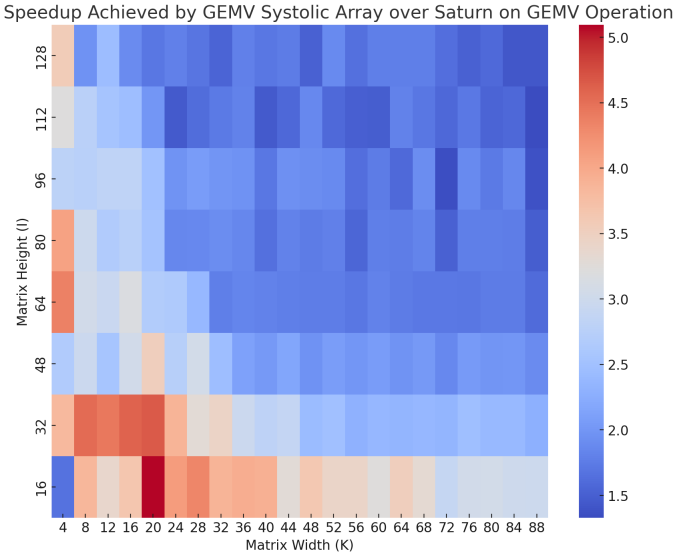

Fig. 14. Speedup Achieved by GEMV Gemmini over Saturn on Randomly Generated GEMM Operations

operations, broadcasting one element of the vector to perform parallel reductions. The GEMV kernel uses register grouping with `LMUL`=8 to reduce instruction throughput requirements from the front end. However, driven by a simple Rocket core, for small matrix heights (I), this kernel cannot make full use of the grouped registers. In this case, Gemmini, which can perform sequenced operations using the K dimension, performs comparably to Saturn.

On the other hand, Figure 15 shows the speedup of Gemmini relative to Saturn for GEMM operations. For large matrix dimensions, both architectures perform comparably, as they can achieve high utilization of their PEs. However, for smaller matrices, Gemmini benefits from having a more flexible instruction sequencer through its FSM, which can automatically generate a sequence of load, compute, and store instructions, while Rocket must explicitly issue vector instructions to Saturn without being able to take advantage of register grouping for short vectors.

The result demonstrates the distinct strength of both types of architecture. In general, the vector core performs better than the original Gemmini for GEMV operations, and the systolic array achieves a significant performance for GEMM since it exploits the 2D spatial attribute of the GEMM operation.

2) End-to-end Workload Evaluation: To compare the performance of hardware backends on real robotic workload, we evaluated all the hardware designs' performance in computing TinyMPC. Figure 20 plots each hardware configuration using its area, as well as the average frequency at which it can execute the ADMM solver used in TinyMPC. Furthermore, this chart highlights the Pareto-optimal frontier for this application. Reflecting the efficiency gains of switching to specialized architecture compared to general-purpose processors, all design points from both Saturn and Gemmini lie above the frontier set by the scalar cores, Rocket, LargeBoom, and MegaBoom.

a) Performance Evaluation of RISCV CPUs: Breaking down the performance within each category, we profiled representative architecture candidates from each category: RISC-V CPU cores, RISC-V CPU with vector extensions Saturn, and the systolic array Gemmini, where RISC-V CPU cores include simple in-order core, superscalar in-order core, out-of-order BOOM core, a more performant out-of-order MegaBOOM core with 2FPUs. When comparing the end-to-end performance of control workloads, we use Rocket running optimized scalar Eigen code as a baseline. However, when evaluating the performance of individual kernels, we use the `matlib` implementation of the kernel; although `matlib` is

not as performant as Eigen, our `matlib` implementation has the same interface as the accelerated implementation of each kernel.

The different BOOM configurations showcase varying specifications designed for specific performance needs. The Small BOOM has a fetch width of 4, a decode width of 1, and three instruction queues (IQ) for memory (mem), integer (int), and floating-point (fp) pipelines. The Medium BOOM also features a fetch width of 4 but increases the decode width to 2, enhancing its IQ configurations with mem having 1 issue and 2 dispatches, int with 2 issues and 2 dispatches, and fp with 1 issue and 2 dispatches. The Large BOOM continues with a fetch width of 4 and a decode width of 1, increasing dispatch capacity with mem, int, and fp pipelines featuring more dispatch capabilities. The Mega BOOM expands further with a fetch width of 8 and a decode width of 4, having each pipeline's IQ equipped for higher issue and dispatch rates.

As the CPU configuration scales up, the performance over the end-to-end workload grows, especially when the CPU exploits out-of-order and superscalar, as seen in BOOM's performance improvements. CPUs also deliver better performance when dedicated instruction queues for each pipeline to drive instruction-level parallelism, which maximizes the CPU's compute availability. However, this increased performance margin comes with the cost of area overhead and power consumption. All the CPU configurations fail to perform efficiently, as shown in the Pareto optimal Figure 20 analysis.

b) Performance Evaluation of Saturn: Saturn shows a strong performance across a variety of operations but seems particularly effective in tasks like primal and dual state updates, residual calculations, and linear operations based on Figure 16. This could suggest efficient utilization of vector operations which can process multiple data points in a single instruction, enhancing throughput and decreasing cycle time for these specific tasks.

In operations where high data parallelism is exploitable, Saturn tends to perform better, which is evident from the consistent high speedup factors, particularly in the last chart, where it achieves a very high speedup in several operations. In these cases, Saturn can fully map operations to vector registers without significant overhead from handling tail cases and can make use of instruction sequencing and register grouping.

c) Performance Evaluation of Gemmini: Gemmini's performance peaks in specific operations like forward passes and linear cost updates, as shown in Figure 18, suggesting that its architecture is highly optimized for scenarios where matrix-vector operations are dominant. The significant variability in Gemmini's performance across different operations may indicate its specialized nature, excelling greatly in its niche (matrix operations) but perhaps not as flexible or efficient for other types of data processing tasks compared to general-purpose vector processors like Saturn.

d) Comparison Setup: Furthermore, we conduct an initial comparison between Gemmini and Saturn to better understand each architecture's relative performance and their unique strengths in certain types of operations. The configurations

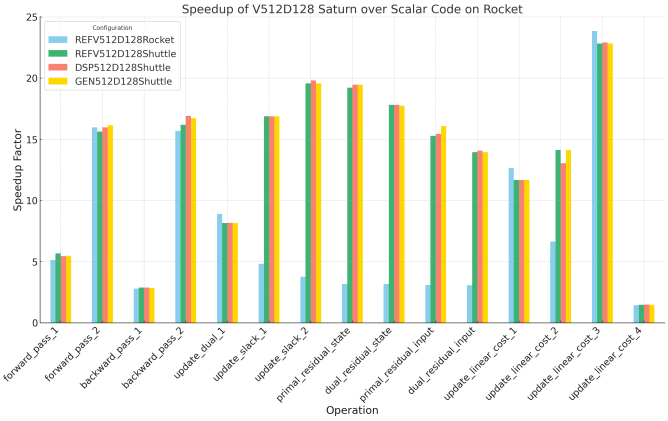

Fig. 16. Performance of Saturn(V512D128) on End-to-End Workload TinyMPC with Kernal Breakdown

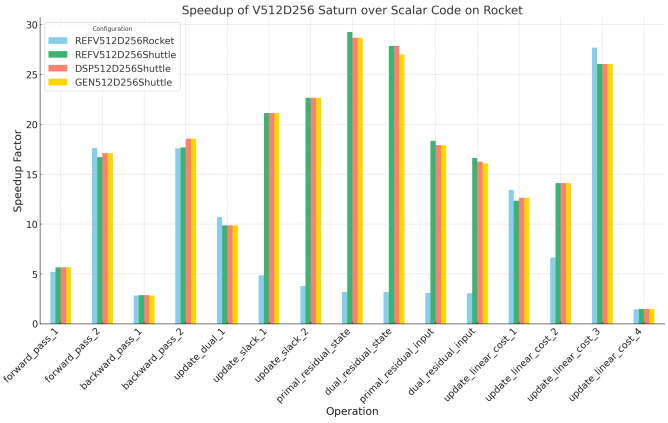

Fig. 17. Performance of Saturn(V512D256) on End-to-End Workload TinyMPC with Kernal Breakdown

are carefully chosen, such as Gemmini with 4×4 FP Mesh and Saturn V512D512, both driven by Rocket. These configurations ensure the backends of each architecture have equal amounts of PEs, thus allowing them to perform theoretically the same FLOPs/cycle. Besides the same computational capacity of the backend of each candidate, choosing the front end to be Rocket for both also allows the instructional throughput to be relatively similar among both candidates.

At a high level, Saturn shows more uniform and often higher speedup across a broad range of operations, suggesting better general-purpose usability with its vector extensions, especially for varied computational tasks. On the other hand, Gemmini excels in specific tasks tailored to its systolic array architecture, achieving exceptional speedup in matrix-related operations but showing less versatility across a broad spectrum of computational tasks.

e) Performance Comparison Across all Architectures: Due to the flexible nature of the RVV vector instruction set as well as the Saturn generator, we were able to evaluate many hardware configurations using the same software mapping. By using RVV intrinsics and dynamically computing `VLMAX`, the same binary can be executed on every hardware configuration evaluated. Since the iterative functions within TinyMPC can

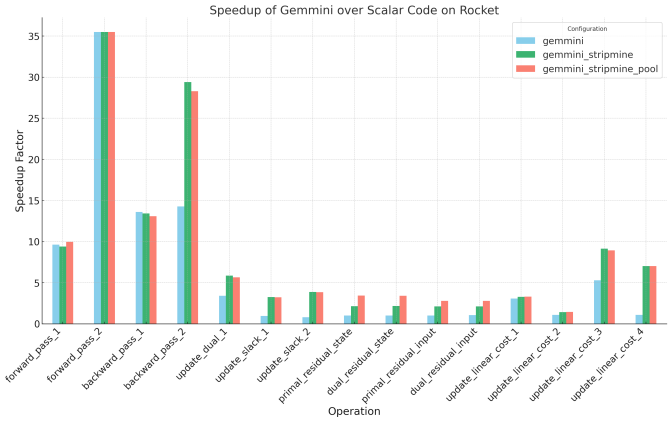

Fig. 18. Performance of Gemmini(4x4 FP Mesh) on end-to-end workloads with TinyMPC with kernel breakdowns.

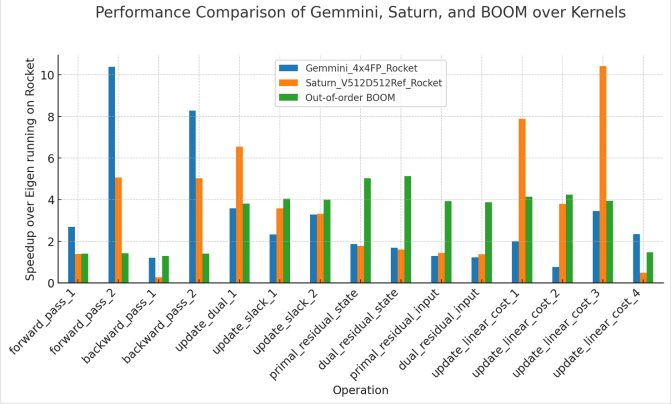

Fig. 19. End-to-End Performance Comparison of Saturn vs Gemmini on End-to-End Workload TinyMPC with Kernal Breakdown

often only utilize 4 vector lanes at most, Saturn configurations with a DLEN of 128 are more efficient compared to their counterparts with a DLEN of 256. Furthermore, achieving the highest optimal performance required both a superscalar Shuttle frontend and a DLEN=256 backend. However, interestingly, the single-issue reference design outperformed the DSP and general-purpose Saturn configurations for a DLEN=256, even though both of those configurations should outperform the single-issue reference design. Further investigations could include evaluating more area-minimal Saturn configurations. Currently, under 1.4mm^2, a Rocket core is the most efficient implementation. However, minimal Saturn configurations could result in improved performance in this domain due to Saturn's instruction sequencing. For higher-performance configurations, hardware support for explicitly operating on small, dense matrices could improve Saturn's performance as the vector and datapath lengths scale beyond what was evaluated in this work.

Evaluating multiple configurations for Gemmini was more challenging, as programming Gemmini using its low-level assembly interface requires manual rewriting across changes to the mesh. Because of this, we primarily focused on creating a single, well-optimized implementation for the OSGemminiRocket configuration, which uses a 4×4 output-stationary FP32 mesh, evaluated with both a 64KB and 32KB scratchpad. Furthermore, we include the area report for an equivalent weight stationary design, which requires the generation of a 1KB accumulator memory for computing intermediate results. However, this implementation has significantly worse performance since the software optimizations aside from software unrolling and static mapping have not been implemented for this design. Under an area window of 1.5mm^2 to 2.3mm^2, Gemmini is the optimal design, even without the use of hardware support for GEMV operations. In this case, the efficiency improvements could be due to the ability to fuse scaling and matrix operations, alongside the fact that the fine-grained instruction sequencing works well with blocks of matrices that are multiples of 4. To achieve a more thorough understanding of Gemmini's end-to-end performance characteristics, future evaluations can consider other mesh dimensions, smaller scratchpad capacities, as well as hardware modifications such as the GEMV support presented in this work, as well as activation and pooling functions suitable for classical control workloads.

B. Area Evaluation

To understand area usage, we use post-synthesis area results generated by ASAP7 toolkit [6]. We found that both Gemmini and Saturn consume relatively smaller areas than an out-of-order MegaBOOM core with 2FPUs [27]. Gemmini has smaller areas overhead (1.5-2mm^2). A breakdown of the area for a 4×4 output stationary Gemmini with a 32KB scratchpad and a V512D256 reference Saturn, both driven by a Rocket core, is depicted in Figure 21. Proportionally, the FPFMAs in Gemmini's mesh combined with the scratchpad contribute to a greater proportion of the design than in Saturn. However, this is because Saturn also supports a vectorized integer pipeline. Additionally, Gemmini's scratchpad has 16× the capacity of the Saturn register file despite only using 35% more area. This is because the scratchpad is synthesized with SRAMs, whereas the Saturn design uses flip-flops. This suggests that Saturn's area could be significantly improved by using a more dense memory technology for the register file and by replacing the integer backend with a simplified scalar design to better align with the computing requirements of robotics workloads.

Additional area report Table II provides a comparison of Gemmini with and without our GEMV architectural change. The additional 2% area increment in a 4x4 mesh is attributed to the extra scratchpad banks since our feature needs at least DIM + 1 banks to ensure all the necessary data for the mesh is loaded in one cycle. In all runs, we configured Gemmini to have DIM + 1 scratchpad banks, rounded up to the nearest power of 2. This is a current requirement of a legal Gemmini configuration; however, we would like to address this constraint in the future. Currently, as DIM scales, the utilization of these extra banks decreases, making a 16x16 mesh where 15 banks are in excess impractical. This can be attributed to the extra logic per scratchpad bank, as traditional Gemmini only utilizes the 2 banks in a single computer. The

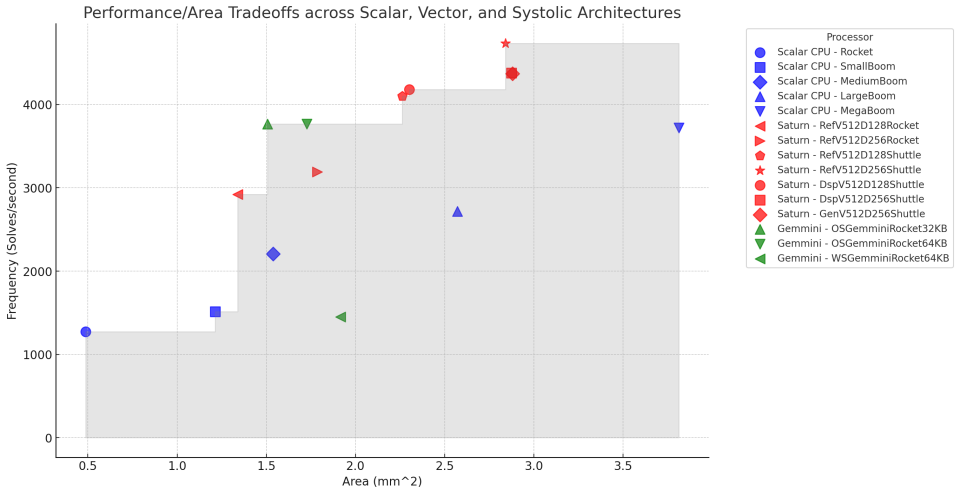

Fig. 20. Saturn vs Gemmini Performance vs Area Tradeoffs

area is largely dominated by scratchpad banks, which would benefit from the changes stated above. Looking further, the mesh area has been insignificantly impacted by our design due to the lack of logic between the mesh columns. On the other hand, ExecuteController grows 9.2% in size between 4×4 GEMM and 4×4 GEMV as it shoulders more responsibility to distribute data from different scratchpad banks to every PE. This relationship appears to be linear as DIM scales, as the area grows by 18% between 8×8 GEMM and 8×8 GEMV.

TABLE I
PERFORMANCE AND AREA METRICS OF SCALAR, VECTOR, AND SYSTOLIC ARCHITECTURES

Categories	Configurations	Area (um^2)	Performance (Cycles/solve)
CPU	TinyRocket	186,963	—
	Rocket	486,287	392,261.0
	Shuttle	826,608	—
	SmallBoom	1,212,513	330,139.0
	MediumBoom	1,537,374	226,510.4
	LargeBoom	2,570,964	183,907.0
	MegaBoom	381,402,3	134,356.7
Saturn	RefV512D128Rocket	1,340,095	171,189.4
	RefV512D256Rocket	1,786,260	156,721.1
	RefV512D128Shuttle	2,262,203	121,870.0
	RefV512D256Shuttle	2,840,849	105,611.2
Gemmini	OSGemminiRocket32KB	1,506,498	132,696.9
	OSGemminiRocket64KB	1,726,167	132,696.9
	WSGemminiRocket64KB	1,916,970	344,682.5
	OSGemminiRocket64KB	2147008	132,696.9

VII. CONCLUSION

Our profiling and optimization across various computing platforms ranging from CPUs and vector machines to domain-specialized accelerators encompass both kernel-level benchmarks and a comprehensive end-to-end robotic workload. We compare integrated accelerators like RISC-V cores with vector extensions to more decoupled systems such as systolic arrays, evaluating their performance, area efficiency, and utilization. This exploration not only quantifies the trade-offs inherent between these architectural paradigms but also underscores how the choice of hardware architecture is contingent upon specific workload characteristics and application demands.

This project conducts a thorough design space exploration for architecture research for classical and optimization-based algorithms for embedded robotics applications. Different architectures are optimal under each area constraint, and well-optimized software mappings are critical to achieve improved performance. To address these challenges and open questions, future work will involve design-space exploration across a broader spectrum of hardware design spaces, as well as robotics algorithms.

A significant challenge for deploying optimized robotics algorithms on specialized hardware is the engineering overhead of hand-optimizing the software mapping. To facilitate rapid evaluation of new hardware designs, we pursue two potential options. First, we explore a library-based approach supporting common operations. Existing interfaces such as Eigen [10] result in poor utilization for small operands, and do not provide clean abstractions for specialized memory units such as software-managed scratchpads.

To address this, we develop the `matlib` library, with ongoing work on applying optimizations such as kernel fusion. Currently `matlib` is implemented using RVV intrinsic, and has planned support for the Gemmini ISA. We are currently developing automated code-generation flows to emit optimized embedded solvers on top of the `matlib` interface, with the end goal of being able to pass in hardware configurations and robot parameters (which impact matrix and vector sizes), generating optimized libraries for the desired targets. A second approach is to express robotic algorithms directly as computation graphs in an intermediate representation such as an MLIR [**?**] dialect and adapt end-to-end code-generation techniques.

Additionally, realistic robotics workloads consist of multiple software nodes executing concurrently on an embedded SoC. Given optimized mappings of multiple algorithms to several hardware backends, future work will consider strategies for dynamically scheduling these systems while addressing physical and computational constraints in robotic systems.

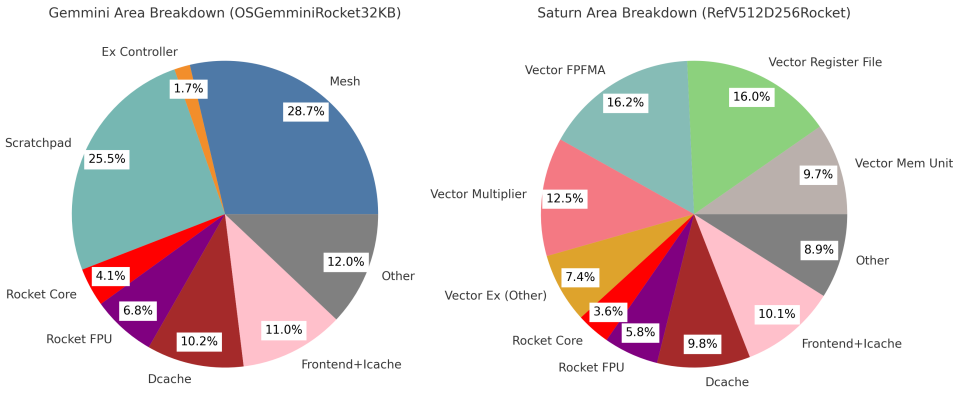

Fig. 21. Gemmini vs Saturn area breakdown

	4x4 GEMM			4x4 GEMV			8x8 GEMM			8x8 GEMV		
	Cell Count	Area	Percentage	Cell Count	Area	Percentage	Cell Count	Area	Percentage	Cell Count	Area	Percentage
RocketTile	355448	2981579.833	100.00%	364171	3431062.477	100.00%	444474	3029321.036	100.00%	509930	3218630.633	100.00%
Gemmini	264805	2488116.71	83.45%	273528	2937598.421	85.62%	353629	2535848.862	83.71%	419434	2725373.477	84.67%
Scratchpad	84384	1998508.846	67.03%	89943	2439544.441	71.10%	116586	1908130.736	62.99%	133421	2054100.891	63.82%
Mesh	17669	43827.618	1.47%	17635	44323.8	1.29%	70045	173683.16	5.73%	69949	176043.669	5.47%
ExecuteController	27391	71909.518	2.41%	29640	78528.903	2.29%	84120	212707.921	7.02%	98092	248889.148	7.73%
ReservationStation	20193	63583.088	2.13%	20447	64351.589	1.88%	19564	61376.576	2.03%	19627	61567.409	1.91%
LoadController	3760	11669.458	0.39%	3746	11723.614	0.34%	3852	11987.269	0.40%	3870	11956.434	0.37%
StoreController	5444	13872.202	0.47%	5454	13884.839	0.40%	5304	13378.469	0.44%	5304	13377.983	0.42%
Other	90643	493463.123	16.55%	106663	285241.235	8.31%	54158	154584.731	5.10%	62357	159437.943	4.95%

TABLE II
COMPARISON OF AREA WITH GEMV SUPPORT ENABLED

REFERENCES

[1] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The Rocket Chip Generator," EECS Department, University of California, Berkeley, Tech. Rep., 2016.

[2] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry *et al.*, "An updated set of basic linear algebra subprograms (blas)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[3] C. P. Celio, *A Highly Productive Implementation of an Out-of-Order Processor Generator*. eScholarship, University of California, 2017.

[4] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, 2016, pp. 262–263.

[5] B. Chrétien, A. Escande, and A. Kheddar, "Gpu robot motion planning using semi-infinite nonlinear programming," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 10, pp. 2926–2939, 2016.

[6] L. Clark, V. Vashishtha, L. Shifren, A. Gujia, S. Sinha, B. Cline, C. Ramamurthya, and G. Yeric, "ASAP7: A 7-nm FinFET Predictive Process Design Kit," *Microelectronics Journal*, 2016.

[7] W. J. Dally, S. W. Keckler, and D. B. Kirk, "Evolution of the graphics processing unit (gpu)," *IEEE Micro*, vol. 41, no. 6, pp. 42–51, 2021.

[8] R. Featherstone, *Rigid body dynamics algorithms*. Springer, 2014.

[9] H. Genc, A. Haj-Ali, V. Iyer, A. Amid, H. Mao, J. Wright, C. Schmidt, J. Zhao, A. Ou, M. Banister, Y. S. Shao, B. Nikolic, I. Stoica, and K. Asanovic, "Gemmini: An Agile Systolic Array Generator Enabling Systematic Evaluations of Deep-Learning Architectures," 2019.

[10] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[11] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[12] C. Kozyrakis and D. Patterson, "Vector vs. superscalar and vliw architectures for embedded multimedia benchmarks," in *35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35), Proceedings*. Istanbul, Turkey: IEEE Computer Society, 2002, pp. 283–293.

[13] R. Li, X. Huang, S. Tian, R. Hu, D. He, and Q. Gu, "Fpga-based design and implementation of real-time robot motion planning," in *2019 9th International Conference on Information Science and Technology (ICIST)*. IEEE, 2019, pp. 216–221.

[14] S. Lian, Y. Han, X. Chen, Y. Wang, and H. Xiao, "Dadu-p: A scalable accelerator for robot motion planning in a dynamic environment," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.

[15] J. Liang, V. Makoviychuk, A. Handa, N. Chentanez, M. Macklin, and D. Fox, "Gpu-accelerated robotic simulation for distributed reinforcement learning," in *Conference on Robot Learning*. PMLR, 2018, pp. 270–282.

[16] S. Murray, W. Floyd-Jones, Y. Qi, G. Konidaris, and D. J. Sorin, "The microarchitecture of a real-time robot motion planning accelerator," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[17] B. Plancher, S. M. Neuman, T. Bourgeat, S. Kuindersma, S. Devadas, and V. J. Reddi, "Accelerating robot dynamics gradients on a cpu, gpu, and fpga," *IEEE Robotics and Automation Letters*, 2021.

[18] U. B. A. Research, "Shuttle: A rocket-based risc-v superscalar in-order core," https://github.com/ucb-bar/shuttle, 2024, accessed: 2024-05-10.

[19] R. M. Russell, "The cray-1 computer system," *Communications of the ACM*, vol. 21, no. 1, pp. 63–72, 1978.

[20] J. Sacks, D. Mahajan, R. C. Lawson, B. Khaleghi, and H. Esmaeilzadeh, "Robox: An end-to-end solution to accelerate autonomous control in robotics," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 479–490.

[21] A. Suleiman, Z. Zhang, L. Carlone, S. Karaman, and V. Sze, "Navion: A 2-mw fully integrated real-time visual-inertial odometry accelerator for autonomous navigation of nano drones," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 4, pp. 1106–1119, 2019.

[22] Z. Wan, B. Yu, T. Y. Li, J. Tang, Y. Zhu, Y. Wang, A. Raychowdhury, and S. Liu, "A survey of fpga-based robotic computing," *IEEE Circuits and Systems Magazine*, vol. 21, pp. 48–74, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:221654980

[23] ——, "A survey of fpga-based robotic computing," 2021.

[24] T. Wang, C. Wang, X. Zhou, and H. Chen, "A survey of fpga based deep learning accelerators: Challenges and opportunities," *arXiv preprint arXiv:1901.04988*, 2018.

[25] Z. You *et al.*, "A macroscopic analysis of gpu power consumption," in *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2023. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/you

[26] J. Zhao, D. Grubb, M. Rusch, T. Wei, B. Nikolic, and K. Asanovic, "Saturn vector unit," https://github.com/ucb-bar/saturn-vectors, 2024, accessed: 2024-10-17.

[27] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "Sonicboom: The 3rd generation berkeley out-of-order machine," in *Fourth Workshop on Computer Architecture Research with RISC-V*, 2020.

APPENDIX

As part of our benchmarking process in this project, we have developed new features and infrastructure in the following repositories:

- Profiling and Acceleration Result: https://github.com/ucb-bar/Accelerated-TinyMPC/tree/rvv_handopt
- `matlib` interface:
  - https://github.com/ucb-bar/matlib
- Gemmini GEMV Hardware: https://github.com/ucb-bar/gemmini/tree/gemv-support
- Gemmini GEMV Software: https://github.com/ucb-bar/gemmini-rocc-tests/tree/gemv-support