2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA) | 979-8-3503-2658-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/ISCA59077.2024.00047

Tartan: Microarchitecting a Robotic Processor

Mohammad Bakhshalipour
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
bakhshalipour@cmu.edu

Phillip B. Gibbons
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
gibbons@cs.cmu.edu

***Abstract*—This paper presents *Tartan*, a CPU architecture designed for a wide range of robotic applications. *Tartan* provides architectural support for *common* robotic kernels, ensuring its broad utility across different robotic tasks. The architecture effectively addresses both computational and memory bottlenecks, marking a significant advancement over previous works. Key features of *Tartan* include architectural support for *oriented vectorization*, *approximate acceleration with accurate outcome*, *robot-semantic prefetching*, and *intra-application cache partitioning*.**

On the six end-to-end robots in the RoWild Suite, *Tartan* boosts the performance of legacy robotic software by 1.2$\times$ (up to 1.4$\times$), non-approximable software optimized for *Tartan* by 1.61$\times$ (up to 3.54$\times$), and approximable software optimized for *Tartan* by 2.11$\times$ (up to 3.87$\times$).

***Index Terms*—Robotics, Domain-Specific Processors, Approximate Computing, Specialization.**

I. Introduction

Robots are rapidly permeating our society, transforming various aspects of life, from economy [56] and healthcare [64] to agriculture [65] and military [63]. Market reports predict over 20 million robots in operation by 2030, with over $210 billion robotics market capitalization [74], [155]. To reach this potential, robots must seamlessly integrate into real-world environments, demanding autonomous capabilities and real-time execution of complex artificial intelligence tasks [118].

Computer architecture plays a critical role in enabling real-time robotics [86], [129], [130], [132]. The recent surge in research has introduced numerous *hardware accelerators*, developed in both academia [89], [101], [143], [149] and industry [91], [111], [120], aiming to expedite *specific* robotic tasks like motion planning and scene understanding.

Nevertheless, robotics is in a state of constant evolution. With a plethora of algorithms and ideas emerging constantly, the state of the art shifts rapidly, rendering rigid hardware accelerators obsolete. Moreover, these accelerators struggle to adapt beyond their specifically targeted applications. However, the field of robotics encompasses a vast range of applications, spanning from industrial robots to atmospheric robots. It is unlikely that hardware vendors will implement specialized hardware for each of these diverse robot types individually.

To address these challenges, we propose *Tartan*, a CPU architecture tailored for robotics. We undertake a thorough evaluation of robotic workloads to pinpoint software bottlenecks and mismatches between workload demands and architecture's capabilities (§III). Based on the insights gained from this analysis, we develop *Tartan* with a focus on mitigating bottlenecks in *common* robotic kernels (e.g., pathfinding), providing architectural support to enhance their performance. The focus on common bottlenecks ensures that *Tartan* is versatile and more future-proof, capable of supporting various robots and (future) algorithms that utilize these kernels.

A distinctive feature of *Tartan* is its improvements to the memory subsystem. *Tartan* introduces enhancements to the cache hierarchy, targeting and mitigating memory bottlenecks that impede robotic applications [81]. This emphasis on enhancing memory-bound performance distinguishes *Tartan* from previous efforts in "hardware acceleration of robotics," which mainly concentrated on *computational* acceleration.

Key architectural features of *Tartan* include:

- **Oriented Vectorization:** *Tartan* introduces a novel approach for vectorizing non-contiguous memory accesses that display *oriented* patterns, frequently encountered in robotic tasks like ray-casting and collision detection. *Tartan* implements an *explicit*, *in-hardware* address generator to efficiently capture these patterns, departing from the implicitly-contiguous or software-based gather address generation methods used in existing processors.
- **Approximate Execution *Accurate* Results:** *Tartan* offers support for the approximate acceleration of robotics applications. Specifically, *Tartan* introduces a novel computational paradigm termed AXAR: by leveraging guarantees from specific algorithms, it allows for the approximation of certain computations without affecting the final result.
- **Robot-Semantic Prefetching:** *Tartan* aims to leverage semantic features in robotics for its novel hardware data prefetchers. Specifically, *Tartan* introduces an Adaptive Next-Line Prefetcher (ANL) that utilizes the application's semantic information to adapt the prefetching degree. The semantic information used by *Tartan* (e.g., sparse versus dense environmental areas) is relevant across a wide range of robotic applications. This work represents the first effort in the robotics field to leverage application semantic information for hardware prefetching.
- ***Intra*-Application Cache Partitioning:** *Tartan* introduces a cache management scheme to address *intra*-application contentions in complex robotic scenarios (e.g., pathfinding in unpredictable terrains). It seeks to implement a (soft) partitioning of the cache space among semantic units (e.g., paths) to optimize cache performance. This work is the first to explore cache partitioning for a singularly running application to *enhance its performance*. It also represents

Authorized licensed use limited to: Carnegie Mellon University Libraries. Downloaded on August 28,2024 at 22:32:02 UTC from IEEE Xplore. Restrictions apply.

the first exploration into partitioning *private* caches.

- **Engineering Optimizations:** *Tartan* incorporates engineering optimizations tailored for robotics, including adjustments to cacheline size and selective caching policies to enhance the performance of robotic workloads.

In addition to architectural enhancements, we propose software-only techniques that leverage the features of modern processors. Specifically, we introduce an aggressively vectorized implementation of the nearest neighbor search kernel using locality-sensitive hashing.

Our evaluation of *Tartan* using six end-to-end robotic applications from RoWild [81], which are modeled after real-world robots, shows substantial performance enhancements in all tested scenarios. The observed performance gains span from $1.31\times$ to $3.87\times$. *Tartan*'s area overhead is merely 0.001%.

II. Motivation

A. The Need for Efficient Robotics CPUs

Tartan is a *CPU* architecture tailored for robotics. While DSPs, GPUs, FPGAs, and ASICs are increasingly popular in robotics [3], [19], [39], [72], CPUs continue to be an indispensable element of *every* robot manufactured to date. We believe CPUs will *remain* a critical component in robotics for the foreseeable future, owing to the following attributes:

- **General-Purpose Processing:** Robotics encompasses a broad array of algorithms, each exhibiting varied computational behaviors. While programmable accelerators like DSPs, GPUs, and FPGAs excel in specific computation models (e.g., SIMD), not all robotics tasks align with these models. Conversely, CPUs are designed to manage a wide spectrum of computation types, adeptly accommodating the expanding computational diversity of robotics algorithms.
- **Single-Thread Performance:** Due to their aggressive architecture and high clock frequencies, CPUs deliver superior single-thread performance compared to a GPU's single core or an FPGA's individual logic block. This capability is essential for providing real-time *latency* in robots, not only for hard-to-parallelize algorithms, but also for gather-scatter parallel algorithms where a main thread initiates multiple worker threads and aggregates their results–both these computation models are prevalent in robotics [81], [123]. For instance, RoWild [81] demonstrates that *CPUs outperform GPUs* in robotic workloads characterized by high instruction-level but low thread-level parallelism. Along with quick execution of sequential tasks, the CPU's ability for fast context switching renders it crucial for robots requiring real-time decision-making and control.
- **Reliability:** CPUs, as a mature technology, undergo extensive testing at hardware and software levels, resulting in robust error-handling capabilities and well-documented failure modes. The extensive use of ECC memory and parity checks further bolsters CPU fault-tolerance [87], [102]. This reliability is critical for robots operating in inaccessible or demanding settings, like space, where other platforms like FPGAs struggle to offer as high reliability [160]. For instance, Valkyrie, NASA's robot for space exploration, performs all its operations on three Intel Core-i7 CPUs [26], with only error-tolerant sensor interpretation algorithms on an NVIDIA Quadro 1000M GPU [23], and no FPGA [140].
- **Price:** Achieving widespread adoption of robots across various applications hinges on their cost-effectiveness. The inclusion of hardware accelerators like GPUs and FPGAs can substantially elevate production costs; even applications requiring real-time performance may not always justify the heightened price [106]. As such, CPU may be the sole computing platform in some robots, running the entire robotics software, as is the case in real-world robots like [2], [17], [46], [60], [68], [69].

Due to the extensive use of CPUs in various robotic applications and their pivotal role in robot performance, robotics CPUs have undergone significant evolution over the past decade, becoming markedly more powerful (e.g., increased transistor count, higher clock frequencies, deeper pipelines, larger caches). This evolution is illustrated by microcontroller-based robots using Raspberry Pi, transitioning from a single-core ARM11 CPU in the Raspberry Pi 1 [52] to a 4-core ARM Cortex-A76 CPU in the Raspberry Pi 5 [53]; Qualcomm's robotics platform evolving from a Kryo 385 CPU in the RB3 [51] to a Kryo 585 CPU with double the last-level cache size in the RB5 [50]; NVIDIA robotics boards upgrading from a quad-core ARM Cortex-A57 CPU with 2MB of total cache in the Jetson TX1 [35] to a 12-core Arm Cortex-A78AE CPU with 9MB of total cache capacity in the Jetson AGX Orin [34]; and Intel's NUC, which is extensively utilized in a variety of robots such as [30], [38], [48], transitioning from a Celeron 847 processor [25] with a maximum clock frequency of 1.1GHz in its first generation [29] to a Core i9-12900 processor [27] with a maximum clock frequency of 5.1GHz in its twelfth generation [28].

Nevertheless, as we show in this paper, substantial potential for enhancement remains even beyond the capabilities of state-of-the-art processors. We extensively explore the architectural implications for robotics and suggest architectural improvements to a cutting-edge robotics CPU, aiming to boost performance across a diverse range of robotic applications.

B. The Need for Efficient Memory Systems

Robotics are becoming increasingly data-intensive [1], [77], [81], [153], fueled by the need to process large volumes of data produced by more precise sensors (e.g., high-resolution cameras and LiDARs, highly sensitive force sensors) and the large number of model parameters necessary for robots to function accurately in the wild. This trend underscores the role of memory systems in processors for efficiently managing data delivery to the processors [95], [142].

Unlike prior work that focused only on computational acceleration, *Tartan* also provides enhancements to the memory subsystem, specifically improving the cache hierarchy to alleviate memory bottlenecks in robotics [81].

Authorized licensed use limited to: Carnegie Mellon University Libraries. Downloaded on August 28,2024 at 22:32:02 UTC from IEEE Xplore. Restrictions apply.

III. Performance Study

In this section, we perform a performance analysis to identify bottlenecks within robotic workloads, aiming to target these areas for optimization. We begin by outlining our methodology, then proceed to discuss the results of the study.

A. Methodology

System: Our methodology involves initially establishing a baseline processor model, followed by the integration of our proposed architectural enhancements. We model the baseline processor after Intel Core i7-10610U Processor [26], which is integrated into NASA's Valkyrie [41]. The processor features four OoO cores fabricated in 14nm. It includes L1-D, L2, and a shared L3 cache with sizes of 32KB, 256KB, and 8MB, respectively. The latencies for these caches are 4, 14, and 45 clock cycles. The chip includes two DDR4-2666 channels, which offer a bandwidth of up to 45.8GB/s.

Upgraded Baseline: We apply engineering optimizations to benchmark *Tartan* against an upgraded baseline.

- We upgrade the processor's vector ISA and hardware from AVX2 to the current leading standard, AVX-512.
- Recognizing that robotic workloads are adversely affected by excessive unnecessary data movements (UDM) [81], we shrink the cacheline size from 64B to 32B. This adjustment leads to a 1.56× reduction in UDM and yields a slight average performance enhancement.
- Data structures facilitating producer-consumer communications across stages of the robotics software pipeline are allocated to memory regions managed by write-through policies, through manipulation of memory type range registers [24]. This results in a 9%–43% reduction in L3 cache traffic and a 2%–4% improvement in overall performance.

Framework: We use ZSim [146] to evaluate our proposal. We run all applications until their completion and report execution time for performance analysis.

Workloads: We evaluate all six robots from the RoWild suite [81]. The workloads model the end-to-end functionality of real-world robotics applications, containing the computation of all software pipeline stages: *perception* (sensing and interpreting environment), *planning* (decision-making process), and *control* (executing planned actions). Table I details the workloads with the algorithms used in their software and the number of threads used in each stage of the perception→ planning→ control pipeline of the robots.

TABLE I: Application parameters. **Bold** algorithms are time-dominant.

Robot	Resembling	Major Algorithms	Pipeline Threads
DeliBot	Spot [12]	**MCL** [167], Greedy [139]	8→ 1→ 1
PatrolBot	Pioneer 3-DX [47]	**MobileNet** [164], EKF [157], PP [133]	1→ 1→ 1 ‖ 4†
MoveBot	LoCoBot [73]	**RRT** [117], **CCCD** [161], PID [162]	1→ 8→ 1
HomeBot	Roomba i7+ [57]	**Point-Based Fusion** [151], BT [105]	8→ 1→ 1
FlyBot	Pelican [7]	LT [126], **WA*** [139], MPC [100]	1→ 4→ 4
CarriBot	Boxbot [13]	POM [103], **A*** [139], DMP [115]	1→ 4→ 1

† Four threads run network inference in parallel with the robot's software pipeline.

We tune software for the evaluated processor, leading to a slightly different thread count compared to original RoWild's [81] evaluations on ARM Cortex A57 of NVIDIA Nano board [33]. Although threads outnumber the cores, empirical findings indicate these settings as optimal. This can be largely attributed to the uneven distribution of work among threads and the benefits of latency hiding [138].

B. Bottleneck Analysis

We conduct a thorough evaluation of RoWild's robots on our framework to identify bottlenecks and mismatches between the demands of the workloads and the capabilities of the architecture. The insights gained from this analysis are utilized in designing *Tartan*. Fig. 1 summarizes the results.

Baseline presents the execution time breakdown for the upgraded baseline processor, while Tartan illustrates how *Tartan*, employing the techniques explained later, focuses on and accelerates bottleneck operations. Below, we detail the execution statistics of applications on the baseline processor.

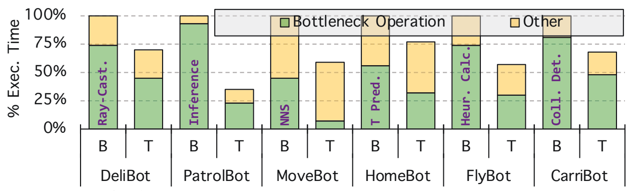


Fig. 1: Execution time breakdown and bottleneck analysis.

DeliBot utilizes MCL for localization [167], heavily relying on "ray-casting" operations that consume 74% of the end-to-end time. Ray-casting matches laser data with the robot's location hypotheses by checking occupancy in the environment map cell-by-cell. Despite proximity of memory checks, vectorization is unfeasible on current processors due to misalignment with vector loads (see §IV).

PatrolBot conducts object detection by feeding captured images into a pre-trained neural network [164] to identify suspicious objects. The neural network inference accounts for 93% of the total processing time.

MoveBot is tasked with moving its arm from one point to another, employing RRT for planning [117]. It uses cuboid-cuboid collision detection (CCCD) [161] to bound obstacles and the robot's body with cubes, checking for intersections during movement planning. CCCD prioritizes speed over accuracy and is parallelized across eight threads, each responsible for assessing collision possibilities with certain obstacles. Without parallelization, CCCD emerges as the primary bottleneck [81]; yet, once parallelized, the major bottleneck shifts to nearest-neighbor search (NNS) operations required by RRT, consuming 45% of execution time. NNS results in irregular memory accesses, challenging existing cache and prefetch techniques in the architecture.

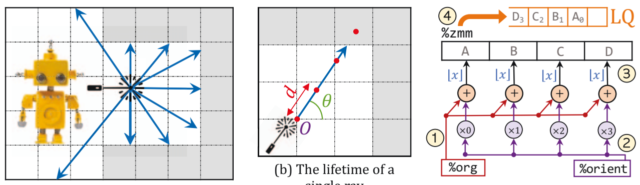
HomeBot uses point-based fusion for 3D reconstruction [151], with 56% of execution time spent on transformation matrix (T) prediction to track the robot's movements. This involves matching point clouds and solving a large linear equation system, including many NNS operations. The irregular memory references from point cloud matching and the heavy floating-point computations for the equations challenge the architecture, stressing memory and processing capabilities.

Authorized licensed use limited to: Carnegie Mellon University Libraries. Downloaded on August 28,2024 at 22:32:02 UTC from IEEE Xplore. Restrictions apply.

FlyBot conducts aerial photography, requiring frequent relocations in a 3D space. It uses the WA* algorithm for path planning [139], utilizing a sophisticated heuristic function to significantly narrow down its search scope. Yet, the computation of this heuristic function, discussed in §V-F, dominates the process, consuming over 74% of the execution time.
CarriBot transports sensitive materials within a factory, employing the A* algorithm [139] with precise collision detection in (x, y, θ) space. This collision detection process is time-intensive, consuming over 81% of the execution time. Similar to ray-casting, it requires verifying the occupancy status of various cells in the environment map along oriented lines.

IV. Oriented Vector Loads

Data accesses during intensive robotic kernels such as collision detection [80] and ray-casting [82] manifest in *oriented* patterns. Fig. 2.a illustrates this using ray-casting as an example. A robot's laser casts rays in different directions to gauge the distance to the nearest obstacle in each direction. Ray-casting is the process of integrating laser-generated distance readings with the robot's pre-existing knowledge (i.e., stored state). During ray-casting, the algorithm scans the map along trajectories that align with the orientation of each emitted ray.



(a) Ray-casting example (b) The lifetime of a single ray (c) Address generation circuit

Fig. 2: Tartan's oriented vectorization.

Fig. 2.b shows the process of ray-casting for a single ray. Let O be the origin of the ray, which is the location of the laser in the environment; let θ be the orientation of the ray with respect to the x-axis; and let d be the step length. The algorithm starts at the origin and iteratively extends the ray's length until it encounters the first obstacle. At step i, the algorithm evaluates the location $(O_x, O_y) + i \cdot (d_x, d_y)$, where $dx = d \cos \theta$ and $dy = d \sin \theta$. All the (x, y) pairs generated during ray-casting are floating-point numbers; these are rounded to integers for mapping to the addressable memory grid. For instance, in a 16×16 environment, stored in `env[256]`, the point $(4.6, 8.5)$ would be flattened to $4.6 \times 16 + 8.5 = 82.1$ and mapped to `env[82]` in memory.

A. The Problem

In Fig. 2.b, the red dots denote the points checked during a single ray-casting operation, with each check determining whether a location is free or occupied. Current CPU vectorization approaches, including post-AVX2 gather instructions (§VIII-A), fail to vectorize such operations: *despite the memory locations checked being nearby and running the same check, the operation cannot be vectorized.*

Consequently, the software needs to sequentially traverse the map on a cell-by-cell basis to perform these checks, leading to excessive processing time. For example, RoWild [81] reports that more than 80% of the end-to-end execution time in two of the six modeled robots are attributed to ray-casting or collision detection–kernels dominated with oriented memory accesses.

B. Vectorization of Oriented Loads

To address the issue, we propose *Oriented Vectorization (OVEC)*. We incorporate an *extra operand* into the vector load instruction: a register containing the traversal orientation. *OVEC* extends the ISA with the following instruction:

```
O_MOVE %zmm, (%org), %orient
```

`zmm` is the destination vector register for data loading. `org` is the memory source operand, which is a register holding the address of the starting point. In ray-casting, `org` initially holds the origin of the ray. These two operands are similar to conventional vector loads. `orient` is the new operand that *OVEC* introduces, which is a *scalar* register encapsulating the traversal orientation; the flattened representation of (d_x, d_y) in 2D or (d_x, d_y, d_z) in 3D in number of bytes. For example, with an $N \times N$ occupancy grid [81], in which every cell stores the occupation probability of an environment location in a `float`, `orient` is $(d_y \times N + d_x) \times$ `sizeof(float)`.

Finally, as in other x86 instructions, the data type is specified in the opcode; e.g., `O_MOVEAPS` and `O_MOVEAPD` for single- and double-precision floating-point, respectively.

C. Implementation Details

Upon execution of the instruction, the needed addresses must be generated and sent to the memory system. Conventional vector load instructions send only the address of the vector's initial lane, stored in `org`. Addresses of the following lanes are *implicitly* sequential (e.g., `org+1`, `org+2`, ...). However, in oriented vector loads, the addresses for each lane within the vector require *explicit generation*.

Fig. 2.c shows the address generation process for a vector comprising four lanes. For each lane i within the vector, the address is computed by adding ① the origin address to ② the product of i and the orientation. ③ The fractional parts of the resulting addresses are omitted, and ④ the integral addresses are enqueued into the load queue (LQ). *Parallel address generation via this hardware circuit*, as opposed to serial checking in software, substantially accelerates operations like ray-casting and collision detection, as we show in §VIII-A.

A challenge in oriented vectorization is the accurate data alignment *within* a vector register. The addresses from an oriented vector load correspond to *different lanes* in the register. Thus, when data are fetched from the memory hierarchy, its designated lane within the vector register is unknown.

To tackle this issue, we adopt Intel's approach for *gather* operations: storing the designated lane for loads within LQ (subscripts in Fig. 2.c). Consequently, when data arrive, it uses the number in LQ to position itself in the intended lane.

Once data are fully loaded into the vector register, it engages the vector ALU similar to conventional vector instructions, implementing operations as directed by the software. The vector ALU and register file remain unchanged.

Authorized licensed use limited to: Carnegie Mellon University Libraries. Downloaded on August 28,2024 at 22:32:02 UTC from IEEE Xplore. Restrictions apply.

D. Related Work

Prior work on augmenting vector unit capabilities encompasses speculative vectorization support for certain operations [135], employing vector units for runahead execution [127], [128], and data streaming [96]. *Tartan* introduces *OVEC*, a novel design markedly distinct from previous initiatives in terms of design, operation, and targeted applications.

V. Approximate-Acceleration

Prior work exploits error-tolerance in tasks like speech recognition, proposing *approximate execution* to trade accuracy for performance. Both software and hardware can implement this approach. For example, NVIDIA's TensorRT [44] uses *quantization*, allowing neural networks to switch from `FP32` to `INT8` representations, which results in faster operations, albeit with some accuracy loss. Similarly, EDEN [114] lowers voltage for DRAM partitions hosting neural network data, sacrificing some accuracy to save energy.

A. Approximate Execution Accurate Results

In this paper, we introduce a new paradigm: *Approximate Execution Accurate Results (AXAR)*. We find that certain computations in robotics can be approximated without altering the *final outcome*, thanks to algorithmic guarantees. We explain and implement *AXAR* in the context of robot path planning. Specifically, we approximate *heuristic cost calculation* in robot path planning without affecting the final path.

Path planning refers to the process of finding an efficient (e.g., short) path from a start to a goal point—a key operation in every autonomous robot. A*, along with its derivatives [83], [108], [119], [139], [168], is widely used in path planning in robotics [81] and beyond (e.g., Google Maps [76]). Central to A* is its heuristic cost function, which makes it an *informed* search algorithm. For a given state S, the heuristic function, $h(S)$, *estimates* the cost to reach the goal from S. In practice, $h(S)$ can be for example the aerial or Manhattan distance of S to the goal. Using the heuristic function drastically narrows the search compared to uninformed algorithms (e.g., Dijkstra).

A*[1] with any *admissible* heuristic outputs an optimal path. An admissible heuristic function h satisfies $h(S) \leq h^\star(S)$ for all S, where $h^\star(S)$ denotes the optimal cost to the goal. This means that *an admissible heuristic never overestimates the cost*. An example heuristic $h(S) = 0$ for all S is technically admissible, but not effective as it provides no insight into the actual cost. Ideally, $h(S)$ should be close to $h^\star(S)$; i.e., $h(S) \lesssim h^\star(S)$—the closer $h(S)$ is to $h^\star(S)$, the more effectively A* narrows the search.

Calculating $h(S)$ can be costly in robotic applications [83], [147], [168]. For instance, it may require solving an optimization problem at each step [83]. We propose *AXAR*-acceleration of such cases by approximating heuristic cost calculation. We argue that *as long as the heuristic admissibility is maintained, the planning outcome will be unaffected.*[2]

[1]More precisely, A* with re-expansions permitted [166].

B. Traditional Approximation

Besides *AXAR*, robotics permits *Traditional Approximation (TRAP)*; trading off some accuracy for performance. For example, while a vacuum robot [81] ideally covers every inch of a floor, occasionally missing a spot does not significantly affect the overall cleaning outcome. This means there is some leniency in the execution of its scene understanding algorithm.

C. Hardware-Accelerated Neural Approximation

Both approximation schemes are important. We find that as much as 74% and 56% of the execution time in our workload suite could potentially benefit from *AXAR* and *TRAP*, respectively (§VIII-B). This underscores the importance for the processor to be compatible with both schemes.

To support both *AXAR* and *TRAP*, *Tartan* includes a *Neural Processing Unit (NPU)* [99], [104] tightly coupled to its pipeline. *NPU* is a spatial array of processing elements (PEs), each with a multiply-accumulate (MAC) unit, a lookup table implementing sigmoid-activation, and dedicated buffers for inputs, weights, and outputs, as shown in Fig. 3.

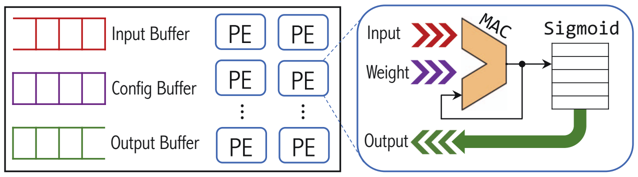


Fig. 3: Tartan's neural processing unit.

As in [99], the programmer marks certain functions as "approximation-safe." Then, a neural model is developed with a focus on *efficiency*, which is indicated by its capability to provide satisfactory accuracy in replicating the outputs of the original function while maintaining a computational footprint that does not far exceed the cost of the original function on CPU. This model replaces the original function at the compile-time, allowing the *NPU* to execute it efficiently during runtime.

At runtime, the CPU initially sends the configuration parameters (e.g., layers and weights) to the *NPU*. Then, the CPU sends inputs to the *NPU* to initiate the inference process for a particular input. Upon completion, the CPU retrieves the inference results from the *NPU* for subsequent operations.

Leveraging the *NPU*'s *highly-parallel architecture*, an efficient neural network can execute significantly faster than the intensive original function on the CPU. Further details, like interrupt handling and CPU-*NPU* communications, being akin

[2]We believe this behavior is not limited to robotics. For example, when finding a graph's minimum spanning tree, edge weights can be approximated; as long as their relative order does not change, the approximation will not affect the output. This concept is innovatively exploited by Tartan, but in fact, it is not new to computer architecture. For example, in caching, in hardware or software, LRU is sometimes approximated by Pseudo-LRU. However, since LRU itself is a heuristic to the optimal policy and caching operates on a best-effort basis, the approximation does not alter execution time significantly nor the results at all. We leave explorations beyond robotics to future research.

Authorized licensed use limited to: Carnegie Mellon University Libraries. Downloaded on August 28,2024 at 22:32:02 UTC from IEEE Xplore. Restrictions apply.

to [99], are not explained further for brevity. However, their quantification and significance are discussed in §VIII-B.

The *NPU*'s configuration, including the number of PEs, can be set based on the available area in the host processor. In §VIII-B, we evaluate various *NPU* configurations.

D. Why "Neural" Approximation?

The inclusion of the *NPU* and the use of neural models, rather than table-based alternatives, serves a dual purpose: *(i)* Versatility: neural models can learn a broad range of functions, surpassing table-based methods in complex applications [121], and *(ii)* Multimodal: as robotics are increasingly reliant on neural network algorithms [81], the *NPU* can expedite both "native" neural networks (i.e., neural models originally employed by the robot software) and "imported" neural networks generated for approximate computing.

Notice *NPU*'s primary purpose is *not* to expedite native neural models; this is rather a secondary benefit. This advantage becomes apparent in scenarios where a robot lacks a GPU or a specific accelerator for neural models, and the *NPU* is available, not being used for approximate acceleration. Thus, it becomes suitable to offload native models to *NPU*. Essentially, for applications that demand large native neural networks, such as advanced perception in autonomous vehicles, the use of GPUs or dedicated accelerators [165] is often necessary, and is complementary to our use of *NPU*.

E. Software Workflow

Model training is offline, utilizing multilayer perceptrons (MLPs) for their balance of performance and cost-efficiency in robotics [158]. Network topology and parameters are tailored to each application, considering the acceptable quality loss. §VIII-B outlines the training data for each application, the selected network topology, and an analysis of quality loss.

F. Training for AXAR

In *AXAR*, the inaccuracy of neural models necessitates a *supervisor* to ensure outputs align with algorithmic requirements, similar to quality controllers in [121], [144]. Unlike these methods that need hardware changes and complex software-hardware co-design, our approach implements supervision in software, integrated *within the algorithmic steps*. Differing from previous methods' emphasis on *predicting erroneous invocations*, our strategy *reduces erroneous predictions* using recent training techniques (see §VIII-B).

We next detail *AXAR* in its application context, applying it to AnyTime A* (ATA*) [119] within FlyBot [81], a drone navigating in 3D. However, it is important to note that the methods and discussions are also relevant to a wide range of other applications.

ATA*: ATA*, widely used in real-time robotics, operates on the principle that *inflating* the heuristic cost with a factor $\varepsilon > 1$ speeds up execution at the expense of generating ε-optimal paths, costing up to ε times the optimal. It starts with a high $\varepsilon = 8$ for a quick initial path, then progressively reduces ε by $step = 1$ to enhance path quality, eventually reaching $\varepsilon = 1$ for the optimal path. Its resilience in unpredictable situations, like delays from unexpected interrupts, makes it popular. In such scenarios, ATA* quickly produces an initial path and continually refines it, or delivers the best path so far, thereby maintaining functionality despite disruptions.

Our method leverages a key aspect of the ATA* algorithm to supervise *AXAR*: each step's path cost does not exceed that of the previous step. The first iteration, with a high ε, runs entirely on the CPU (high ε runs quickly). From the second iteration onward (with lower ε; slower), we offload heuristic cost calculations to the *NPU*. After each iteration's completion (not after every *NPU* invocation), we assess if the current path's exact cost is higher than the previous, indicating *NPU* overestimation. In such cases, the iteration is rerun on the CPU. If not, the process moves to the next iteration. This supervision method introduces minimal overhead, adding a few CPU instructions at the end of each extensive iteration.

This approach ensures that *AXAR* consistently produces outputs within an acceptable range, as the initial iteration runs accurately on the CPU. Thus, *AXAR* maintains the path cost guarantees inherent to the algorithm. Although, in theory, *AXAR* might marginally extend the worst-case execution time by the duration of one NPU-accelerated iteration, the algorithm's design and the fact that the first iteration is CPU-based ensures a key feature: the availability of a viable path even if unexpected events occur post the first iteration. This mirrors the reliability offered by an *AXAR*-less execution.

Training for *AXAR*: FlyBot is a battery-powered drone. It tries to find the shortest path to extend its operation range. It relies on a sophisticated heuristic function to estimate the cost to the goal. The heuristic function calculates the impact of *(i)* aerodynamic drag, *(ii)* altitude change, and *(iii)* wind influence to estimate the cost. Calculating *(ii)* is simple, but computing *(i)* and *(iii)* involves *integrating* over the path, which is computationally expensive.

In our approach, the heuristic function is substituted by a neural model, with an emphasis on *training to minimize CPU rollbacks by minimizing overestimations*. Our training involves an asymmetric, piece-wise loss function which penalizes overestimations more significantly than underestimations:

$$L(y_{\text{true}}, y_{\text{pred}}) = \begin{cases} \alpha \cdot (y_{\text{pred}} - y_{\text{true}})^2 & \text{if } y_{\text{pred}} > y_{\text{true}} \\ (y_{\text{pred}} - y_{\text{true}})^2 & \text{otherwise} \end{cases}$$

Here, $\alpha = 8$ is a constant that determines *how much more* we penalize overestimations. Also, we employ L2 regularization [131] to prevent overfitting by penalizing larger weights. This regularization adds a term $\lambda \sum_i w_i^2$ to the loss function, where $\lambda = 0.01$ is the regularization strength, and w_i denotes the model weights. Lastly, we utilize gradient clipping [124], capping the gradients at $c = 2.5$ during training. This limit is crucial for preventing the model from making excessively large updates, thus aiding in curbing overestimations.

In §VIII-B, we show that, through the employed training techniques, overestimation does not occur during the entire operational period of FlyBot, with more than a million inference operations (see Table II). Finally, it bears repeating that

Authorized licensed use limited to: Carnegie Mellon University Libraries. Downloaded on August 28,2024 at 22:32:02 UTC from IEEE Xplore. Restrictions apply.

the entire process specific to *AXAR* is conducted purely in software. From a hardware standpoint, there is no distinction between *AXAR*, *TRAP*, and native neural network executions.

G. Related Work

Approximation is a widely explored concept in robotics [110] and other fields [99]. The innovations of *Tartan* primarily lie in *AXAR*, proposing that certain computations can be effectively approximated based on algorithmic guarantees to yield reliably accurate results. A secondary contribution of *Tartan* involves examining hardware-based approximation for tasks such as T prediction, detailed further in §VIII-B.

VI. Efficient NNS in High-Dimensional Spaces

Nearest Neighbor Search (NNS) in high-dimensional spaces is essential in key robotic tasks, including motion planning [82], scene reconstruction [169], object detection [81], kinematics [134], and sensor data fusion [152].

Popular libraries like OMPL [62], integrated within ROS [58] and MoveIt [40], implement NNS using k-d trees and octrees. These data structures, however, present several challenges: *(i)* They lead to inefficient memory access due to scattered node locations, especially in deep octree structures. *(ii)* They do no fully utilize application *semantic* information. Sparse areas result in octrees having many underutilized nodes and k-d trees developing long, data-sparse branches, both leading to inefficient traversal. *(iii)* Octrees are not effective beyond three dimensions, posing a challenge for industrial robots with higher degrees-of-freedom (DoF), which often require searches in higher dimensions (typically 6–7, with cases like the 57-dimensional ASIMO robot [8]).

We implement NNS using Locality Sensitive Hashing (LSH), a dimensionality reduction method. While not the first to use LSH for NNS, our approach is unique in how it capitalizes on the architectural capabilities of modern processors.

A. Background on LSH

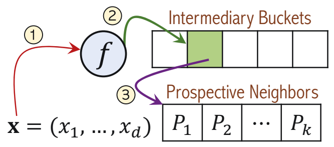
LSH is a technique for reducing dimensionality. It hashes input items so that similar items tend to be mapped to the same "buckets." The fundamental aspect of LSH is its ability to *increase the likelihood of similar items colliding*. In this work, we implement LSH using random projections.

For a point $\mathbf{x} \in \mathbb{R}^d$ (where d is the dimensionality, e.g., 5 for a 5-DoF robot), the hash function is given by $f(\mathbf{x}) = \left\lfloor \frac{\mathbf{x} \cdot \mathbf{r}}{w} \right\rfloor$, where $\mathbf{r}$ is a random d-dimensional vector, each element of which is sampled from a Gaussian distribution $\mathcal{N}(0, 1)$, and w controls the bucket sizes in the hash space.

The likelihood of two points $\mathbf{x}$ and $\mathbf{y}$ hashing to the same value is linked to their Euclidean distance—the closer two points are in Euclidean space, the higher the probability they end up in the same bucket.

B. Approximate NNS using LSH

Fig. 4.a shows how NNS is performed using LSH. For ① a query point $\mathbf{x}$, the hash function, f, ② assigns it to a specific bucket. This bucket ③ contains a set of points $P_1, P_2, \ldots, P_k$, likely to be near $\mathbf{x}$ in Euclidean space. The algorithm *examines* these points and selects those within a predefined distance threshold ϵ (i.e., $\|\mathbf{x} - \mathbf{y}\|_2 \leq \epsilon$) as the nearest neighbors. This selection can include all or a specific number of points, based on the implementation specifics.



(a) Nearest neighbor search using LHS

PC	Region	CD	LD
0x5f	0x4008e1	4	7
0x99	0x40073f	1	1
⋮	⋮	⋮	⋮

(b) Adaptive next-line prefetching

Fig. 4: Nearest-neighbor search with LSH.

Notably, NNS via LSH is *approximate*, as it relies on LSH's probabilistic properties. Nonetheless, in the context of robotic NNS within high-dimensional spaces, where the utilized algorithms (e.g., RRT [82]) inherently accommodate certain levels of error, LSH is an effective approach. For example, RRT aims to find an efficient, rather than optimal, path for planning. Its stochastic nature and reliance on random sampling inherently absorb the imprecision of inexact NNS—the algorithm's success is not predicated on perfect accuracy (optimal path) but on its ability to rapidly explore and connect feasible paths through the space to output an efficient path.

C. Vectorization of NNS

Vector units are becoming increasingly potent, with AVX-512 featuring 512-bit vector registers. However, the full potential of these units is often underutilized, partly due to limitations in compiler optimization capabilities [148].

We identify *untapped* potential for vectorization in LSH-based NNS, a domain where existing implementations, including the widely adopted FLANN [20] (integrated into OpenCV [45]), fall short. We develop a highly-vectorized version of LSH-based NNS, and show that it offers superior performance (§VIII-C). Our approach focuses on vectorizing the projection step (i.e., the dot-product calculation) and aggressively vectorizing the examination process. We call this implementation *Vectorized LHS-Based NNS (VLN)*. *VLN* is a software approach with no hardware modifications.

D. Adaptive Next-Line Prefetching

As Fig. 4.a suggests, access patterns within each bucket are sequential, leading us to employ next-line prefetchers. However, we observed notable variability in the *number* of accesses per bucket, correlating directly with their density. This variance is linked to the *state of the analyzed environment*. For instance, in motion planning, areas densely populated with obstacles result in fewer viable pathfinding points, creating less populated buckets. Conversely, obstacle-free zones yield densely-filled buckets. This trend is observed beyond motion planning, such as in point cloud manipulation for scene

Authorized licensed use limited to: Carnegie Mellon University Libraries. Downloaded on August 28,2024 at 22:32:02 UTC from IEEE Xplore. Restrictions apply.

understanding [82], where we note significant differences in memory access patterns between dense and sparse areas.

To leverage this observation, we introduce an *Adaptive Next-Line Prefetching (ANL)* prefetcher, depicted in Fig. 4.b. *ANL* operates with two counters per `PC+Region` pair: current degree (CD) and last degree (LD). `PC` is the program counter of load instructions, and `Region` is the high-order bits of load addresses. CD *learns* the prefetching degree (i.e., the number of prefetches issued) for each `PC+Region`, while LD stores the past observations to *issue* prefetches.

ANL employs a 16-entry table, tagged by the concatenation of `PC` and `Region` bits. Upon a cache miss, the table is looked up using the `PC+Region` bits of the load. If the table lookup is a hit, we *(i)* prefetch the number of cachelines indicated by LD, *(ii)* increment CD, and *(iii)* reset LD. If it is a miss, we allocate a new entry, possibly evicting an existing one (see below). New entries start with CD and LD set to 0. When a region is *terminated* (a cacheline of it is evicted from the cache), all *ANL* entries tracking that region *(i)* copy their CD to LD and *(ii)* reset CD. This mechanism enables *ANL* to learn distinct access patterns for each `PC+Region`.

Two details are important to the efficient performance of *ANL*. Firstly, *ANL* needs to use small regions to minimize overprediction. *ANL* bases its prefetching decisions solely on the count of used cachelines. Therefore, in medium-density environments, larger region sizes could lead to significant overprediction. In this work, we utilize 1KB regions.[3]

Secondly, when *ANL* needs to evict an entry for a new one, it chooses the entry with the lowest `max(CD, LD)` value. This approach is hardware-implementable with small tables like *ANL*'s (§VIII-C). The rationale behind this policy is to keep entries with higher degrees, as these are responsible for the majority of prefetch requests. This means that *ANL* is less affected by missing prefetch opportunities in sparser regions, whereas missing such opportunities in denser regions would be more detrimental to its performance.

Finally, *ANL* is not designed merely to accelerate NNS. Rather, it is designed as a *general-purpose* prefetcher for robotic applications, adept at learning and adapting to the *density* of references across different regions *during runtime*. In §VIII-C, we evaluate the effectiveness of *ANL* for all six robots. Also, *ANL* can prefetch into any cache level; in this paper, we put the prefetch requests into the private L2 cache.

E. Discussion

Our approach addresses the challenges associated with k-d trees and octrees (§VI). *(i)* Storing points in buckets facilitates cache-friendly, sequential memory accesses. *(ii) ANL* leverages semantic information (e.g., varying densities) within the application. *(iii)* LSH scales well to higher dimensions due to the dimensionality reduction in its projection phase.

[3]Notice, while the explanation of *ANL* draws parallels between LSH "buckets" and prefetcher "regions", and a correlation exists (different buckets correspond to different memory addresses), it is important to differentiate them: "buckets" are conceptual, at the algorithm level, while "regions" pertain to hardware-level physical address granularities.

F. Related Work

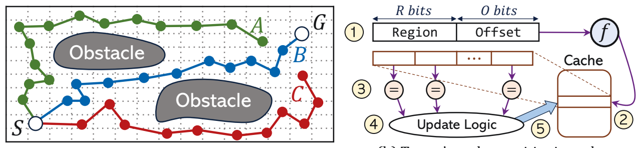
NNS is vital for a broad range of applications and is tackled through a variety of approaches including parallelization [79], [88], compression [97], and application-specific optimizations (e.g., for CNNs [163]). *Tartan* introduces a novel hardware/software strategy that significantly enhances performance (see §VIII-C). While *Tartan*'s NNS solution can stand alone as a simple and effective method, it can also work orthogonally with existing techniques. For instance, compressing LSH data [97] boosts efficiency without compromising the benefits of *VLN*. Alternatively, *Tartan*'s components, such as *ANL*, can be synergistically combined with other methods where its premise, like data heterogeneity, applies.

VII. Intra-Application Cache Partitioning

Graph search is crucial in tasks like pathfinding, motion planning, and decision making. In graph search, the robot seeks a *path* from a start to a goal point. This "path" varies by context: in pathfinding, it is the sequence of locations to the goal; in motion planning, it is the configurations for object grasping; and in decision making, it is the set of actions required to perform a task.

Graph algorithms employed across various robotic applications are varied, yet they share a key feature: *concurrent exploration of multiple paths* to determine an efficient, or the most efficient, path. The definition of efficiency varies by application: for drones, it may be the shortest path; for manipulator robots, the smoothest; and for self-driving cars, the path that optimizes fuel efficiency.

Fig. 5.a illustrates a mobile robot's concurrent exploration of multiple paths in pathfinding, moving from start point S to goal G. Due to two large obstacles, the route *forks* into three paths: A, B, and C. The algorithm concurrently expands these paths, ultimately choosing one as the final route. This illustration omits two key aspects for clarity: *(i)* the actual number of paths can exceed three, depending on the number and proximity of obstacles; *(ii)* each graph node typically involves extensive neighbor explorations, implying numerous memory accesses to adjacent locations.



(a) Pathfinding example (b) Tartan's cache partitioning scheme

Fig. 5: Tartan's FCP with an example application.

A. The Problem

When multiple paths are explored concurrently, they compete for resources, notably hardware caches. This competition leads to paths *evicting each other's data* from the cache. This issue arises even in private caches and is particularly prevalent in widely-used algorithms like A* and RRT. The frequent eviction of data from caches negatively impacts the hit ratio, degrading performance.

Authorized licensed use limited to: Carnegie Mellon University Libraries. Downloaded on August 28,2024 at 22:32:02 UTC from IEEE Xplore. Restrictions apply.

Importantly, the concurrent exploration of multiple paths is *not* a rare event in many applications. In the A* algorithm and its variants, each iteration involves selecting the node with the highest potential for the optimal path, leading to frequent time-wise switches between paths in $A \rightarrow B \rightarrow C \rightarrow A \rightarrow B \rightarrow C \rightarrow \cdots$ order. Therefore, each time $C \rightarrow A$ occurs, there is a possibility that some of A's data has been evicted from the cache, resulting in a slowdown. Similarly, in the RRT algorithm, path choices are based on random sampling, which leads to random switches between paths and the same caching challenge.

B. Cache Partitioning

Partitioning the cache space among different paths can alleviate this issue. However, current cache partitioning solutions like Intel's CAT [32] are unsuitable for several reasons. Firstly, they focus on partitioning *across cores*, which is not applicable to single-thread executions. Secondly, they partition cache by physical ways, an approach not feasible when dealing with potentially tens of paths ($>$ number of ways). Thirdly, these methods often compromise performance to ensure fairness or quality of service, which holds no value in this context. Research proposals in this area [93], [94], [98], [145], [156], [159] similarly encounter one or more of these issues.

In this work, we propose *Fuzzy Intra-Application Cache Partitioning (FCP)*. The key idea of *FCP* is the partitioning of cache by *manipulating replacement metadata*. Specifically, it prioritizes evicting cachelines associated with paths that have excessively used cache capacity. Below, we detail the operations of *FCP*. Unlike CAT [32], *FCP* does not enforce strict cache partitioning but instead implements a "fuzzy" partitioning approach, functioning on a best-effort basis.

FCP is predicated on the understanding that *inter-path* cache contention becomes problematic when paths diverge significantly, each exploring a *distinct, distant memory region*. In other words, when paths traverse spatially close regions, inter-path cache contention is not only unproblematic but also beneficial for spatial locality. For instance, in Fig. 5.a, the $A \rightarrow B \rightarrow C \rightarrow A$ traversals adversely affect A if both B and C access memory regions far from A. Conversely, if either is spatially proximate to A, the inter-path contention from temporally-interleaved $B \rightarrow C \rightarrow A$ traversals can be advantageous. This is because it might (inadvertently) prefetch data for A if that data resides in the cachelines of B or C.

To address the issue, *FCP* first aims to *map some of the data from individual regions to the same cache sets*. It achieves this by altering the cache's indexing scheme. Considering regions of 2^O cacheline size, the incoming addresses (byte offset excluded) comprise R bits for the region and O bits for the offset within the region, as shown in ① in Fig. 5.b. With 2^S sets, the standard practice without *FCP* is to use the lower S bits for indexing. When $O < S$, which is almost always the case, *cachelines from a region never map to the same set*. *FCP* seeks to modify this, increasing the likelihood that *some* cachelines within a region map to the same set.

A naive solution could be to index the cache solely with the R bits of the region, causing cachelines of a region to map to the same set. However, this method is detrimental to regions with good spatial locality. The cache's limited associativity prevents storing all or most cachelines from such regions simultaneously, thus failing to exploit the spatial locality.

To strike a balance between locality and partitioning, our approach involves XORing the low-order l bits of the region with the high-order l bits of the offset when ② indexing the cache. This technique introduces some uniform entropy into the indexing process, aiding in achieving a balance between spatial locality and the objectives of *FCP*. Additionally, to ensure compatibility with *Tartan*'s *ANL* prefetcher (§VI), we exclude the low-order bits of the offset from the XOR operation to prevent cache hotspots induced by the prefetcher. In Section §VIII-D, we will present an experimental analysis to determine the optimal values for l and the region size, thereby choosing the most effective indexing scheme.[4]

The second component of *FCP* involves manipulating replacement metadata. When a cache fill occurs, either due to a demand miss or prefetching of cacheline X, *concurrently* with tag-checking, ③ cachelines that share the same region bits with X are identified for manipulation. These selected cachelines are then processed through an ④ update logic, which modifies their LRU recency. The update unit executes the function $m(x)$ on each recency number, followed by ⑤ writing the updated metadata back into the cache.

It is important to note that these operations occur concurrently with the cache's baseline functions and are fully-implementable in hardware with minor modifications to existing circuitry. However, the function $m(x)$, which manipulates the recency counters, needs additional circuitry. It alters x to *expedite its eviction from the set*, thus preventing the region from occupying excessive cache capacity.

Applying the $m(x)$ function to recency counters within a set modifies the eviction probability of block x_i to $P_{evict}(x_i) \approx 1 - F(m(x_i))$, where $F(m(x_i))$ represents the cumulative distribution function of the transformed recency counters.[5] As such, given the non-uniform access patterns typical in graph processing [125], a non-linear function (e.g., quadratic), can enhance performance by more distinctly differentiating the eviction priorities of frequently versus infrequently accessed blocks. We explore various manipulation functions in §VIII-D. Finally, *FCP* is adaptable to any cache level; for this paper, we focus on its implementation in the private L2 cache.

C. Related Work

To our knowledge, *FCP* is the first effort to partition cache capacity within *one* application to *enhance* its performance. Previous studies on hardware cache partitioning [98], [145], [156], [159] have focused on partitioning cache space among *multiple* applications, aiming to boost aspects such as fairness or quality of service, but often *degrade* the performance.

[4]It is important to note that a bit-wise XOR with one input known is one-to-one, meaning this indexing method does not alter the number of tag bits.

[5]We assume the baseline replacement policy priorities the eviction of lines with *larger* recency counters.

Authorized licensed use limited to: Carnegie Mellon University Libraries. Downloaded on August 28,2024 at 22:32:02 UTC from IEEE Xplore. Restrictions apply.

VIII. Evaluation

We evaluate *Tartan* using the methodology in §III-A, assessing each component individually and then in combination.

A. *Tartan Accelerates Ray-Casting and Collision Detection*

Fig.6 shows the execution time (bars) and dynamic instruction count (dots) across different methods, normalized to the Baseline processor. We evaluate OVEC, Gather, and RACOD [80]. Results are shown only for robots affected by *OVEC*.

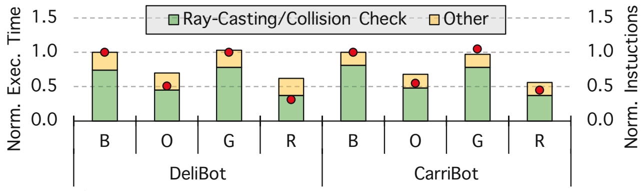


Fig. 6: Oriented access patterns and different vectorization methods.

OVEC, a component of *Tartan* (§IV), is estimated to have a latency of 5 cycles for address generation, based on numbers published in [78], [154]. This estimation accounts for the latency of one floating point addition and multiplication, with the latter simplified in hardware due to *(i)* one constant integer input and *(ii)* no need for the output's fractional part.

Gather, a software implementation of *OVEC* using Intel's `VGATHERDPS` [71], serves as a reference. It involves calculating $\lfloor i \times orient \rfloor$ for each lane i (§IV-C) and arranging them in a vector register *in software*, with `VGATHERDPS` fetching data based on this index vector [24].

RACOD [80], designed for collision detection in mobile robots, parallelizes address generation in hardware. RACOD is not readily applicable to ray-casting. However, to project the speedup that a RACOD-like accelerator can achieve for ray-casting, we model a design that performs both address generation and obstacle-checking in hardware.

Results indicate that *OVEC* substantially boosts ray-casting and collision detection, with speedups of 1.64× and 1.69×, respectively. This is because *OVEC* vectorizes memory fetch operations and exploits the underutilized vector ALU. This not only parallelizes the operations but also reduces the number of executed instructions by an average factor of 1.8×, by transferring the address calculation tasks to hardware, that would otherwise be run by software.

Gather is less effective, as the added instructions for index calculations outweigh the vectorization benefits, resulting in a negligible average speedup of less than 1%. The inclusion of instructions for index calculation (i.e., $\lfloor i \times orient \rfloor$ for different lanes) leads to an increase in the total number of dynamic instructions executed. This surpasses the baseline instruction count, offsetting the advantages of vectorization by increasing the processor's workload. RACOD outperforms both due to eliminating CPU back-and-forths; it fetches addresses and only interacts with CPU for final outcomes. However, RACOD requires *integrating two separate ASIC units* for ray-casting and collision detection. In contrast, *OVEC* achieves 89%/82% of RACOD's benefits in ray-casting/collision detection, with minimal overheads (§VIII-E).

Finally, Intel has fabricated a 10nm ray-casting accelerator [112], which performs in-hardware *trilinear interpolation*, a component of some ray-casting algorithms [137]. This accelerator also includes specialized *local voxel storage (LVS)* to exploit the locality of nearby 3D voxels during ray-casting. However, it lacks any mechanism for vectorizing memory accesses, a feature central to *OVEC*. Consequently, Intel's accelerator is fully orthogonal to our proposal.

To evaluate the effectiveness of this accelerator, we add interpolation into the ray-casting implementation from RoWild. Note that interpolation is not a component of every ray-casting implementation; it is used when a very high level of accuracy is needed. This modification introduces a new bottleneck in ray-casting, which is what Intel's accelerator addresses. Given that the specific details of Intel's accelerator are not available to us, we simulate an optimistic implementation, assuming zero-cycle latency for interpolation operations. Furthermore, we assume an unlimited LVS, where memory references incur cache latency only once before the data are stored in the LVS. Fig. 7 shows the impact on ray-casting time, comparing Baseline (with interpolation), OVEC, Intel, and the combination of the latter two.

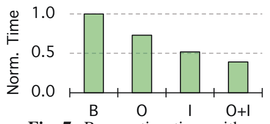


Fig. 7: Ray-casting time with different techniques.

The speedup from *OVEC* decreases from 1.64× to 1.36× due to increased time in what it does not target, i.e., interpolation. Intel's accelerator, by speeding up interpolation and reducing memory references, achieves a 1.92× speedup. When *OVEC* is combined with Intel's accelerator, a cumulative speedup of 2.56× over the baseline and 1.33× over Intel's accelerator alone is observed, reinforcing their orthogonal functionality.

B. *Tartan Significantly Accelerates Approximable Robotics*

Table II details the functions we select for approximate acceleration and their neural network replacements. Note that these functions do not represent the entire spectrum of approximable tasks in robotics. There exist additional tasks (e.g., controlling velocity and acceleration) where exact computation is not strictly necessary. However, for neural acceleration to be beneficial, the tasks must meet two key criteria: *(i)* they should be learnable by an *efficient* neural network, and *(ii)* they must be computationally intensive enough to justify the CPU-*NPU* communication overheads (see below).

TABLE II: The neural network workloads evaluated.

Type	Robot	Function	Topology	Error
AXAR	FlyBot	Heuristic Cost	6/16/16/1	0%
TRAP	HomeBot	T Prediction	192/32/32/6	6.8%
Native	PatrolBot	Classification	50/1024/512/1	1.3%

As discussed in §V-F, for FlyBot, we replace the costly heuristic function with a neural model. The model features

Authorized licensed use limited to: Carnegie Mellon University Libraries. Downloaded on August 28,2024 at 22:32:02 UTC from IEEE Xplore. Restrictions apply.

a topology of 6/16/16/1, which means the network takes 6 inputs (x, y, z coordinates of start and goal), produces 1 output (estimated cost), and has two hidden layers with 16 neurons each. For training, we use a portion of the Freiburg map [21], distinct from FlyBot's operational area [81], and measure the error by the increased size of the final path. We use the custom loss function described in §V-F.

HomeBot employs a neural model for predicting transformations, contrasting baseline's ICP algorithm [81]. The model's training [107] and test [141] data are separate. The error is the geometric mean of rotation and translation errors. The loss function is MSE [116].

PatrolBot's object detection uses a convolutional neural network (CNN), but for the *NPU*, a multi-layer perceptron (MLP) accelerator is used. Despite MLP's limitations in image classification due to input data flattening, their broad learning spectrum led to their choice for *NPU*, aiming for a *general-purpose* approximate-accelerator for various robotic tasks. To showcase *NPU*'s effectiveness, PatrolBot's object detection task is implemented with an MLP. We employ principal component analysis (PCA) [113] with $k = 50$ components for dimensionality reduction, training the model with the same dataset as the original CNN [15]. The loss function is BCE [116]. This MLP model on *NPU* proves to be sufficiently accurate and offers reduced execution time. compared to the original CNN running on the CPU.

As detailed in §V-C, *Tartan* integrates *NPU* directly into the CPU's pipeline, ensuring close interaction. An alternative design involves treating *NPU* as a distinct *co-processor*, similar to Tesla's FSD chip [22], where two independent neural processing units operate outside the CPU die. In such a setup, every time CPU invokes *NPU*, it needs to manage communication by sending messages off-die, launching the *NPU*'s kernel, and collecting the results upon completion.

Fig. 8 compares execution times and dynamic instruction counts across methods, normalized to the Baseline processor. Results are shown only for robots affected by *NPU*. We evaluate Hardware-accelerated and Software-executed neural models, with the former running on a 4-PE *NPU* and the latter implemented using [67] on the baseline processor. In software-executed neural models, the target function is replaced with a neural network executed on software. We assume a CPU-*NPU* communication latency of 4 clock cycles and 8 clock cycles for MAC operations. We also evaluate *NPU* configured as a Co-processor. Optimistically, we project the CPU-*NPU* communication delay to be 104 cycles, drawing from insights into FSD's architecture [22]. Also, we assume zero-cycle inference latency for this arrangement, considering that standalone, off-die *NPU*s might achieve more aggressive performance compared to those integrated within the CPU.

The results indicate substantial target function speedups with hardware-accelerated neural executions for PatrolBot, HomeBot, and FlyBot (3.85×, 1.52×, and 2.7×, respectively, communication time included). These speedups are achieved while maintaining acceptable accuracy, as shown in Table II. On the other hand, software-executed neural models suffer

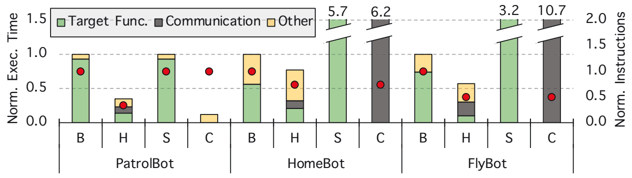


Fig. 8: Neural acceleration of robotics. 'Target Func.' refers to the function designated for neural acceleration. 'Communication' denotes the time spent in CPU-NPU communications. 'Other' signifies the execution time of the remaining program components, i.e., those not selected for neural acceleration.

from significant slowdowns due to increased dynamic instruction counts over the original code, software-based MAC operations that require calculating neuron weight addresses and loading them, and overhead from library function calls.

Utilizing large *NPU*s as co-processors, akin to Tesla's FSD, proves highly advantageous for "native" neural network tasks like object detection in PatrolBot, where *infrequent* CPU-*NPU* interactions (milliseconds-scale) suffice. However, in scenarios like the approximate computations in HomeBot and FlyBot, where only segments of the code execute on the NPU and results must be frequently relayed back to the CPU, the benefits are negated by the high CPU-*NPU* communication overhead in a co-processor architecture, leading to significant performance degradation. This aligns with findings from the original *NPU* study [99], emphasizing that CPU-*NPU* communication latency should be minimal (e.g., 1–4 cycles) to achieve meaningful performance gains. Contrary to FSD's approach, *Tartan* necessitates integrating *NPU* directly into the CPU pipeline to effectively harness approximate acceleration.

Table III explores how varying the number of PEs affects speedup. More PEs allow for increased parallelism of operations, leading to further speed improvements. Given these outcomes, we select a 4-PE design for the *Tartan*'s *NPU*. Although increasing PEs to 8 enhances speedup, the primary benefit accrues to PatrolBot, with minimal gains for other robots.

TABLE III: Different NPU configurations.

PEs	Memory	GMean Speedup	Area [μm^2]
2	10.5KB	1.25×	920
4	18.8KB	1.58×	1661
8	35.3KB	1.68×	3144

A 4-PE *NPU* utilizes 18.8KB of SRAM, with 16.5KB dedicated to PEs and 2.3KB for their interconnect [99]. Most of the per-PE area is allocated for storing weights (2KB) and the Sigmoid LUT (512×32 bits), while a smaller portion is used for input/output buffers (64B). The interconnect comprises a bus scheduler (1.25KB), input/output buffers (1KB), and a configuration FIFO (32B), as illustrated in Fig. 3. The logic area, required for a 32-bit MAC per PE, occupies an insubstantial part of the silicon area (§VIII-E).

Finally, integrating the *NPU* into every core is not necessary. In this paper, we consider its integration into just one core. This approach is similar to heterogeneous core architectures in processors such as CELL-BE [90] and ARM Big.LITTLE [5], where cores possess varying capabilities. The *NPU* is incor-

Authorized licensed use limited to: Carnegie Mellon University Libraries. Downloaded on August 28,2024 at 22:32:02 UTC from IEEE Xplore. Restrictions apply.

porated into a select core, with the runtime or programmer directing tasks for *NPU* execution to that specific cores.

C. Tartan Accelerates Nearest Neighbor Search and Beyond

1) Hardware/Software Nearest Neighbor Search

We evaluate *Tartan*'s techniques for NNS in MoveBot and HomeBot, both heavily reliant on NNS. Fig. 9 shows execution time and L2 misses for various methods. We evaluate Brute-force, VLN, FLANN, and K-d tree, with methods marked with a '+' using *ANL* in hardware. Results are normalized against brute-force search without *ANL*.

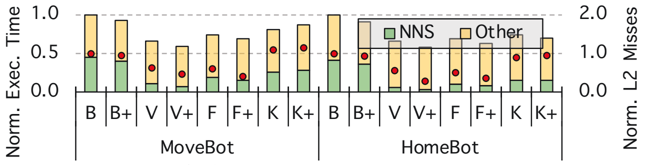


Fig. 9: NNS with different approaches.

The brute-force search, serving as the baseline [81], iterates over all points to identify those close to the query point. Our *VLN* employs LSH and vectorization for NNS (§VI-C). FLANN [20] also uses LSH, but without aggressive vectorization. We tune the bucket sizes (w) for each method to ensure robotic operation accuracy within 1% of the brute-force method (§VI-A). The k-d tree method uses [36].

Our results show that *VLN*, our software-only technique, not only surpasses brute-force and k-d tree but also significantly outperforms FLANN. The NNS performance gains of *VLN* over brute-force, FLANN, and k-d tree are 5.29×, 1.7×, and 2.43×, respectively. The NNS speedup of *VLN* with *ANL* enabled over brute-force rises to 9.37×. The brute-force approach is exhaustive, searching all nodes, while k-d tree, though an improvement, suffers from costly cache misses. Its misses are often *dependent*, causing full stalls [109].

The advantage of *VLN* over FLANN lies in its effective use of processor vectorization capabilities. Compilers like GCC, Clang, and ICC currently struggle to efficiently vectorize LSH-based NNS computation patterns, as seen in FLANN, due to conditional branches in each iteration (§VI-B) [75].

2) Adaptive Next-Line Prefetcher

Fig.10 evaluates *ANL* across all six robots. For context, we also examine Next-Line and Bingo [84]. NL is *not* adaptive and serves to evaluate the significance of adaptiveness. Bingo, a state-of-the-art spatial prefetcher, like *ANL* learns per-page history, albeit with a different algorithm and structure. 'Coverage' is the fraction of L2 cache misses covered by the prefetcher, and 'Accuracy' is the fraction of prefetch requests used by the application.

ANL offers high coverage and accuracy across all workloads, showing its versatility in robotics. It effectively handles the sparse-dense environmental heterogeneity in robots, prefetching memory requests efficiently (§VI-D). In contrast, Next-Line fails to provide high miss coverage due to the untimeliness of its requests (one prefetch per invoke).

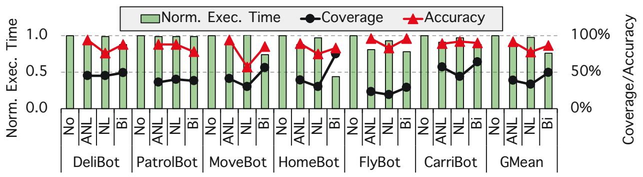


Fig. 10: Different prefetching approaches.

Although Bingo shows higher performance due to its sophisticated pattern learning capabilities, it incurs a significant per-core area overhead of over 100KB for history pattern storage. Conversely, *ANL* matches 85% of Bingo's performance improvement on average with 1000× less area overhead.

In some robots like PatrolBot, prefetchers' high miss coverage does not lead to significant end-to-end speedups. This is typical in compute-bound robots [81], where the absolute number of cache misses is low, rendering even a high coverage of these misses insufficient for substantial performance gains.

Finally, *ANL*'s metadata table tracks 16 entries, with each entry comprising 12 low-order bits from the program counter plus 38 bits from region addresses for tagging, and 10 bits per entry for recording the current and last degrees. This results in a 120B per core overhead. Also, the logic for implementing the table's replacement policy incurs minimal overhead, requiring only a few integer comparators (§VIII-E).

D. Tartan Effectively Mitigates Inter-Path Cache Contention

Fig. 11 assesses *FCP* across various manipulation functions and $R-l$ configurations, where R is the region size and l is the number of bits used for XOR (see §VII-B). The results are normalized to a baseline without *FCP*. The L2 cache is 8-way set-associative in the evaluated processor.

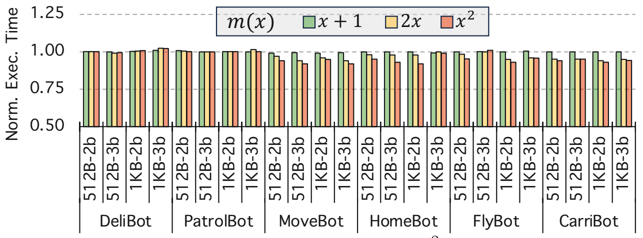


Fig. 11: FCP with different parameters. $x+1$, $2x$, and x^2 are different manipulation functions ($m(x)$) applied to the replacement counters with *FCP* (§VII-B).

The evaluation shows differing behaviors based on the chosen parameters. For example, $l = 3$ is effective in graph-search-intensive robots like MoveBot but incurs slowdowns in certain scenarios where the underlying assumptions do not apply. We select $l = 2$ bits, setting the region size to 1KB.

More, results shows the critical role of the manipulation function, $m(x)$. As expected, $m(x) = x^2$ enhances performance by creating more distinct eviction priorities, as discussed in §VII-B. The function $m(x) = 2x$ also demonstrates competitive performance, trailing x^2 by only 2.9%. We opt for $m(x) = x^2$ in *FCP* due to its superior performance. Notably, the full x^2 logic need not be implemented in hardware; given the known input range, a small lookup table can efficiently

Authorized licensed use limited to: Carnegie Mellon University Libraries. Downloaded on August 28,2024 at 22:32:02 UTC from IEEE Xplore. Restrictions apply.

realize this function. *FCP* achieves up to 8% in performance improvement and a 18% reduction in L2 misses.

Some applications exhibit only modest gains, mainly because most L2 misses are serviced by the larger 8MB L3 cache, whose latency is somewhat tolerable by the employed aggressive OoO cores. Our analysis indicates that *FCP* is not warranted for L3 in our workloads. However, its general-purpose nature suggests potential for more significant improvements in other scenarios. For instance, in less aggressive robotic CPUs [43], private cache misses are less tolerable, or in other graph-intensive applications with high L3 miss rates [85], implementing *FCP* for L3 is more justifiable.

E. Putting It Altogether

So far, we evaluated components individually on *modified* software. It is crucial that the components operate in harmonious synergy. Fig. 12 shows the end-to-end speedup of *Tartan* with all components enabled over the baseline processor.

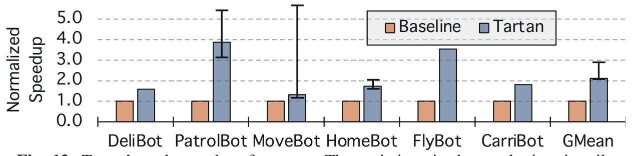


Fig. 12: Tartan's end-to-end performance. The variations in the results is primarily attributed to the applications' extensive use of random number generation.

The results confirm the seamless integration of *Tartan*'s components, preserving their individual performance benefits. An exception is noted in the combination of *NPU* and *ANL* with the optimized software. Here, the integration of *NPU* leads to substituting the T estimation in the ICP algorithm with a neural approach, thereby removing the NNS operations necessary in the baseline algorithm from the approximate version. This change results in a diminished impact of *ANL*, as the opportunities for speedup are reduced.

Fig. 12 shows the results for approximable, optimized-for-*Tartan* software. The results show that *Tartan* achieves an average speedup of 2.11× across all workloads. *Tartan* achieves this speedup by providing architectural supports for accelerating the key bottlenecks in robots (see §III-B).

When approximation is not allowed, *Tartan* offers a speedup of 1.61×. The reduction in speedup comes from not utilizing *NPU*. Also, *Tartan* enhances the performance of legacy software (i.e., not optimized for *Tartan*) by 1.2×. The hardware-only techniques, *ANL* and *FCP*, contribute to performance improvements for both legacy and optimized software.

Finally, Table IV shows the estimated overheads for each component, using data from [78], [154]. Assuming a mobile die area of 133mm^2 in 14nm [31], the overall overhead for *Tartan* is merely 0.001%.

TABLE IV: Overhead breakdown.

Component	Memory	Area [μm^2]
4 × OVEC	—	258
1 × NPU	18.8KB	1661
4 × ANL	480B	30
4 × FCP	12B	1
Total	19.3KB	1949

The overhead in *OVEC* arises from the logic used for address generation. For *NPU*, the overhead is attributed to its PEs and their interconnections. The main source of overhead in *ANL* is its metadata table. Finally, the overhead of *FCP* mainly stems from its 8-entry lookup table per L2 cache, which facilitates the implementation of the manipulation function.

IX. Other Related Works

Tartan belongs to the category of *Domain-Specific Processors*—processors whose architectures are specifically optimized for certain types of workloads. Notable examples include ARM's Neoverse [136] and Cavium's ThunderX [66] for cloud computing; D. E. Shaw's Anton [150] for molecular dynamics simulation; Oracle's SPARC M8 [61] for databases; Cisco's Silicon One [14] and Marvell's ARMADA [6] for networking; Bitmain Antminer [10] for cryptocurrency mining; and, SandForce's SF [59] for SSD management. Each of these processors is tailored to excel in their respective domains.

Robotic developer kits, such as Arduino [4], Raspberry Pi [54], and others [9], [18], [37], [42], [55], [70], are designed to provide a user-friendly platform for robotics development. These kits are furnished with integrated peripherals, enabling interaction with a wide array of sensors and actuators. However, unlike *Tartan*, which offers robot-specific microarchitectural optimizations, these kits come with *general-purpose processors*, such as the ARM Cortex series.

Custom hardware accelerators are developed to optimize different robotic tasks. Qualcomm's QCS610 SoC [49] features dedicated SLAM hardware, a key robotic function. Texas Instruments' DRV8305 [16] includes hardware support for motion control, and Bosch's BMI085 [11] offers hardware-accelerated robotic sensory data fusion. The scholarly realm also sees a surge in related proposals [80], [89], [101], [122], [129], [143], [149] and artifacts [91], [92], [111], [112], [120]. These accelerators optimize *specific* operations, and their application range remains narrow. In contrast, *Tartan* is optimized for a broad spectrum of robotic workloads. Moreover, *Tartan* introduces methods to overcome the "memory wall," an issue often overlooked by hardware accelerators.

X. Conclusion

This paper introduces *Tartan*, a CPU architecture specifically designed for robotics. *Tartan* aims to rectify the limitations of current processors by integrating targeted architectural advancements for more efficient robotic task execution. While currently focusing on performance enhancement, future iterations of *Tartan* could extend its capabilities, improving aspects like cyber-security, user privacy, and error resilience, thereby advancing the development of real-time robotic systems.

Acknowledgment

This work was supported in part by National Science Foundation grant CNS-2211882 and by the Parallel Data Lab (PDL) Consortium (Amazon, Google, Hitachi, Honda, IBM, Intel, Jane Street, Meta, Microsoft, Oracle, Pure Storage, Salesforce, Samsung, Two Sigma, and Western Digital). Mohammad Bakhshalipour was supported by the James Sprague Presidential Fellowship from Carnegie Mellon University.

Authorized licensed use limited to: Carnegie Mellon University Libraries. Downloaded on August 28,2024 at 22:32:02 UTC from IEEE Xplore. Restrictions apply.

APPENDIX

A. Abstract

This artifact comprises the implementation of all software and hardware components of *Tartan*, as well as scripts for replicating the final, end-to-end performance results (Fig. 12). The software components within *Tartan* utilize x86 assembly instructions, built upon the RoWild [81] benchmark suite's six end-to-end robotic applications. On the hardware side, *Tartan*'s components are implemented in the ZSim [146] simulator. This artifact is designed to facilitate the reproduction of our results derived from hardware-software co-design, and to share our implementations of various hardware and software components with the research community for further exploration.

Two methods are provided for reproducing the results: *(i)* Native Execution (NE) and *(ii)* Docker Execution (DE). NE involves installing all necessary packages on the host machine and running the experiments natively, which is considerably faster than DE, where the experiments are run inside a Docker container. Both methods produce equivalent results.

B. Artifact check-list (meta-information)

- **Program:** End-to-end robotic applications implemented in C++ and augmented by x86 assembly instructions for simulation purposes. Plus the hardware processor model implemented in the ZSim simulator.
- **Compilation:** GCC 11.1.0 or above.
- **Data set:** The environments for robots to function, all prepared using the supplied scripts.
- **Metrics:** End-to-end speedup.
- **Output:** Plot with end-to-end speedup data.
- **Experiments:** Generate experiments using supplied scripts.
- **How much disk space required (approximately)?:** 5GB.
- **How much time is needed to prepare workflow (approximately)?:** Less than an hour.
- **How much time is needed to complete experiments (approximately)?:** 2 hours for NE or 6 hours for DE.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT License.
- **Archived (provide DOI)?:** `10.5281/zenodo.10981770`

C. Description

1) How to access

The artifact can be cloned from GitHub at `https://github.com/cmu-roboarch/tartan.git` or downloaded as a `.zip` file from `https://zenodo.org/doi/10.5281/zenodo.10981770`.

2) Hardware dependencies

The artifact runs on any general-purpose CPU with at least 5 GB of free disk space. For optimal performance, a machine with 16 or more cores is recommended.

3) Software dependencies

For NE, dependencies include OpenCV, Intel PIN, and various Linux and Python packages, installed via the provided `setup.sh` script, assuming a Debian-based OS.

For DE, all necessary software packages are pre-installed in the Docker image.

4) Data sets

Utilizes the RoWild [81] benchmark suite data sets for robotic modeling, with preparation handled by included scripts.

D. Installation

1) Native Execution

To install, clone the repository and run the setup script:

```
$ git clone https://github.com/cmu-roboarch/tartan.
    ↪ git
$ ./setup.sh
$ source ${HOME}/.bashrc
```

2) Docker Execution

For Docker installation and setup:

```
$ apt-get install docker.io
$ systemctl start docker
$ service docker status
```

E. Experiment workflow

1) Native Execution

Run the `replicate.py` script to execute all experiments:

```
$ ./replicate.py
```

2) Docker Execution

Set up a `results` directory and run the Docker container:

```
$ mkdir -p results
$ docker run --net=host -it --privileged --name
    ↪ my_interactive_tartan -v "$(pwd)/results:/
    ↪ tartan/results" kasraa/tartan:latest
```

F. Evaluation and expected results

In both NE and DE, execution of the scripts will generate a `results/` directory containing a `.csv` file and a `.png` file depicting the performance results.

REFERENCES

[1] "3 New Chips to Help Robots Find Their Way Around," https://spectrum.ieee.org/3-new-chips-to-help-robots-find-their-way-around.
[2] "ABB IRB 1200," https://new.abb.com/products/robotics.
[3] "An Unprecedented Edge AI and Robotics Platform," https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/.
[4] "Arduino Robot Kit," https://www.arduino.cc.
[5] "ARM Big.LITTLE Architecture," https://www.arm.com/technologies/big-little.
[6] "ARMADA - Marvell," https://en.wikichip.org/wiki/marvell/armada.
[7] "AscTec Pelican," https://www.aeroexpo.online/prod/ascending-technologies/product-181442-24426.html.
[8] "ASIMO Specifications," https://asimo.honda.com/asimo-specs/.
[9] "BeagleBone Blue," https://www.beagleboard.org/boards/beaglebone-blue.
[10] "Bitmain Antminer S19 Pro," https://www.asicminervalue.com/miners/bitmain/antminer-s19-pro-110th.
[11] "Bosch BMI085 IMU," https://www.bosch-sensortec.com/products/motion-sensors/imus/bmi085/.
[12] "Boston Dynamics' Spot Robot," https://www.bostondynamics.com/products/spot.
[13] "Boxbot Launches Last-Mile, Self-Driving Parcel Delivery System," https://www.roboticsbusinessreview.com/supply-chain/boxbot-launches-last-mile-self-driving-parcel-delivery-system/.

Authorized licensed use limited to: Carnegie Mellon University Libraries. Downloaded on August 28,2024 at 22:32:02 UTC from IEEE Xplore. Restrictions apply.

[14] "Cisco Silicon One Q100 and Q100L Processors Data Sheet," https://www.cisco.com/c/en/us/solutions/collateral/silicon-one/datasheet-c78-744214.html.
[15] "COCO Dataset," https://cocodataset.org/#explore.
[16] "DRV8305 Three Phase Motor Driver," https://www.ti.com/product/DRV8305.
[17] "Dual-Arm YuMi - IRB 14000," https://new.abb.com/products/robotics/robots/collaborative-robots/yumi/dual-arm.
[18] "EZ-Robot Developer Kit," https://www.ez-robot.com.
[19] "FANUC Robotics Products," https://www.fanucamerica.com/products/robots/.
[20] "FLANN - Fast Library for Approximate Nearest Neighbors," https://github.com/flann-lib/flann.
[21] "Freiburg Campus 360 Degree 3D Scans," http://ais.informatik.uni-freiburg.de/projects/datasets/fr360/.
[22] "FSD Chip - Tesla," https://en.wikichip.org/wiki/tesla_(car_company)/fsd_chip.
[23] "Https: //www.techpowerup.com/gpu-Specs/quadro-1000m.c1431," NVIDIAQuadro1000M.
[24] "Intel 64 and IA-32 Architectures Software Developer Manuals," https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.
[25] "Intel Celeron Processor 847," https://ark.intel.com/content/www/us/en/ark/products/56056/intel-celeron-processor-847-2m-cache-1-10-ghz.html.
[26] "Intel Core I7-10610U Processor," https://www.intel.com/content/www/us/en/products/sku/201896/intel-core-i710610u-processor-8m-cache-up-to-4-90-ghz/specifications.html.
[27] "Intel Core I9-12900 Processor," https://www.intel.com/content/www/us/en/products/sku/134597/intel-core-i912900-processor-30m-cache-up-to-5-10-ghz/specifications.html.
[28] "Intel NUC 12 Extreme / Pro X," https://www.intel.com/content/dam/support/us/en/documents/intel-nuc/NUC12DCM_NUC12EDB_TechProdSpec.pdf.
[29] "Intel NUC Board DCP847SKE," https://ark.intel.com/content/www/us/en/ark/products/71620/intel-nuc-board-dcp847ske.html.
[30] "Intel NUC10i5 (Fully Configured)," https://roverrobotics.com/products/intel-nuc.
[31] "Intel's Broadwell-U Arrives Aboard 15W, 28W Mobile Processors," https://techreport.com/news/intels-broadwell-u-arrives-aboard-15w-28w-mobile-processors/.
[32] "Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 V4 Family," https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html.
[33] "Jetson Nano Developer Kit," https://developer.nvidia.com/embedded/jetson-nano-developer-kit.
[34] "Jetson Orin Technical Specifications," https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/.
[35] "Jetson TX1 Module," https://developer.nvidia.com/embedded/jetson-tx1.
[36] "KDTree," https://github.com/crvs/KDTree.git.
[37] "LEGO Mindstorms EV3," https://www.lego.com/en-us/themes/mindstorms.
[38] "LoCoBot," https://www.trossenrobotics.com/locobot-base.aspx.
[39] "Modern Hardware Platforms Used in Robotics," https://evergreen.team/articles/how-to-create-robots.html.
[40] "MoveIt," https://moveit.ros.org.
[41] "NASA Humanoid Robot to Be Tested As Remote Oil Rig Attendant," https://www.theregister.com/2023/07/10/nasa_to_test_humanoid_robot.
[42] "NVIDIA Jetson Nano," https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/.
[43] "NVIDIA Jetson Nano System-On-Module," https://developer.nvidia.com/downloads/embedded/dlc/jetson-nano-system-module-datasheet.
[44] "NVIDIA TensorRT," https://developer.nvidia.com/tensorrt.
[45] "OpenCV: Open Source Computer Vision," https://docs.opencv.org/4.x/index.html.
[46] "PicoGo," https://www.waveshare.com/wiki/PicoGo.
[47] "Pioneer 3-DX," https://www.generationrobots.com/media/Pioneer3DX-P3DX-RevA.pdf.
[48] "Powered by Intel Technology, Robot Helps Retail Customers Find the Right Computer," https://www.intel.com/content/www/us/en/newsroom/news/with-intel-tech-robot-assists-retail-customers.html#gs.5h69r1.
[49] "Qualcomm QCS610/615," https://www.qualcomm.com/products/technology/processors/application-processors/qcs610.
[50] "Qualcomm QRB5165 SoC for IoT," https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/qrb5165-soc-product-brief_87-28730-1-b.pdf.
[51] "Qualcomm Robotics RB3 Platform (SDA/SDM845)," https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/robotics-rb3-platform-product-brief.pdf.
[52] "Raspberry Pi 1," https://www.pololu.com/product/2760.
[53] "Raspberry Pi 5," https://www.raspberrypi.com/products/raspberry-pi-5/.
[54] "Raspberry Pi Robot Kit," https://www.raspberrypi.org.
[55] "Robotis Bioloid Kit," https://www.worthpoint.com/worthopedia/robotis-programmable-humanoid-bioloid-468854112.
[56] "Robots and International Economic Development," https://itif.org/publications/2021/01/25/robots-and-international-economic-development/.
[57] "Roomba I7+ Self-Emptying Robot Vacuum," https://www.irobot.com/en_US/roomba-vacuuming/robot-vacuum-irobot-roomba-i7-plus/I755020.html.
[58] "ROS - Robot Operating System," https://www.ros.org/.
[59] "SandForce SF2600 and SF2500 Enterprise," https://www.seagate.com/www-content/product-content/lsi-fam/enterprise-flash-controller/en-us/docs/enterprise-fsp-sf-2500-ds1828-1-1409us.pdf.
[60] "SCARA Robot," http://www.innovativerobotics.com/Downloads/SCARA%20robot%20vs%20r-theta.pdf.
[61] "SPARC M8 Processor," https://www.oracle.com/us/products/servers-storage/sparc-m8-processor-ds-3864282.pdf.
[62] "The Open Motion Planning Library," http://ompl.kavrakilab.org/.
[63] "The Rise of Robots in Defence," https://www.rowse.co.uk/blog/post/the-rise-of-robots-in-defence.
[64] "The (robotic) Doctor Will See You Now," https://www.weforum.org/agenda/2021/03/why-robots-can-be-beneficial-in-healthcare/.
[65] "The Role of Robotics in Agriculture," https://www.challenge.org/knowledgeitems/the-role-of-robotics-in-agriculture/.
[66] "ThunderX2 - Cavium," https://en.wikichip.org/wiki/cavium/thunderx2.
[67] "Tiny-Dnn: Header Only, Dependency-Free Deep Learning Framework in C++14," https://tiny-dnn.readthedocs.io/en/latest/.
[68] "TurtleBot3," https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/.
[69] "UArm Swift & UArm Swift Pro Specifications," http://download.ufactory.cc/docs/en/uArm-Swift-Specifications-171012.pdf.
[70] "VEX Robotics Kit," https://www.vexrobotics.com.
[71] "VGATHERDPS/VGATHERDPD - Gather Packed Single, Packed Double with Signed Dword Indices," https://www.felixcloutier.com/x86/vgatherdps:vgatherdpd.
[72] "Yaskawa: Intel FPGA in Robot Controllers," https://www.intel.com/content/www/us/en/customer-spotlight/stories/yaskawa-customer-story.html.
[73] "LoCoBot: An Open Source Low Cost Robot," http://www.locobot.org/, 2012.
[74] "How Robots Change the World," https://resources.oxfordeconomics.com/how-robots-change-the-world/, 2019.
[75] "C++ Vector Class Library Version 2," https://www.agner.org/optimize/vcl_manual.pdf, 2022.
[76] "How Does Google Map Works?" https://www.geeksforgeeks.org/how-does-google-map-works/, 2022.
[77] M. Afrin, J. Jin, A. Rahman, A. Rahman, J. Wan, and E. Hossain, "Resource Allocation and Service Provisioning in Multi-Agent Cloud Robotics: A Comprehensive Survey," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 2, pp. 842–870, 2021.
[78] M. Anders, H. Kaul, S. Mathew, V. Suresh, S. Satpathy, A. Agarwal, S. Hsu, and R. Krishnamurthy, "2.9 TOPS/W Reconfigurable Dense/sparse Matrix-Multiply Accelerator with Unified INT8/INTI6/FP16 Datapath in 14nm Tri-Gate CMOS," in *2018 IEEE Symposium on VLSI Circuits*. IEEE, 2018, pp. 39–40.
[79] F. André, A.-M. Kermarrec, and N. Le Scouarnec, "Accelerated Nearest Neighbor Search with Quick Adc," in *International Conference on Multimedia Retrieval (ICMR)*, 2017, pp. 159–166.
[80] M. Bakhshalipour, S. B. Ehsani, M. Qadri, D. Guri, M. Likhachev, and P. B. Gibbons, "RACOD: Algorithm/Hardware Co-Design for Mobile Robot Path Planning," in *International Symposium*

Authorized licensed use limited to: Carnegie Mellon University Libraries. Downloaded on August 28,2024 at 22:32:02 UTC from IEEE Xplore. Restrictions apply.

in Computer Architecture (ISCA). IEEE/ACM, 2022. [Online]. Available: https://doi.org/10.1145/3470496.3527383

[81] M. Bakhshalipour and P. B. Gibbons, "Agents of Autonomy: A Systematic Study of Robotics on Modern Hardware," *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)*, vol. 7, no. 3, dec 2023, https://cmu-roboarch.github.io/rowild. [Online]. Available: https://doi.org/10.1145/3626774

[82] M. Bakhshalipour, M. Likhachev, and P. B. Gibbons, "RTRBench: A Benchmark Suite for Real-Time Robotics," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2022, https://cmu-roboarch.github.io/rtrbench/. [Online]. Available: https://doi.org/10.1109/ISPASS55109.2022.00024

[83] M. Bakhshalipour, M. Qadri, D. Guri, S. B. Ehsani, M. Likhachev, and P. Gibbons, "Runahead A*: Speculative Parallelism for A* with Slow Expansions," in *International Conference on Automated Planning and Scheduling (ICAPS)*, 2023.

[84] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo Spatial Data Prefetcher," in *International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 399–411.

[85] S. Beamer, K. Asanović, and D. Patterson, "The GAP Benchmark Suite," *arXiv:1508.03619v4*, 2017.

[86] B. Boroujerdian, H. Genc, S. Krishnan, B. P. Duisterhof, B. Plancher, K. Mansoorshahi, M. Almeida, W. Cui, A. Faust, and V. J. Reddi, "The Role of Compute in Autonomous Micro Aerial Vehicles: Optimizing for Mission Time and Energy Efficiency," *ACM Trans. Comput. Syst.*, vol. 39, no. 1–4, jul 2022. [Online]. Available: https://doi.org/10.1145/3511210

[87] R. Canal, C. Hernandez, R. Tornero, A. Cilardo, G. Massari, F. Reghenzani, W. Fornaciari, M. Zapater, D. Atienza, A. Oleksiak *et al.*, "Predictive Reliability and Fault Management in Exascale Systems: State of the Art and Perspectives," *ACM Computing Surveys (CSUR)*, vol. 53, no. 5, pp. 1–32, 2020.

[88] L. Cayton, "Accelerating Nearest Neighbor Search on Manycore Systems," in *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2012, pp. 402–413.

[89] F. Chen, R. Ying, J. Xue, F. Wen, and P. Liu, "ParallelNN: A Parallel Octree-Based Nearest Neighbor Search Accelerator for 3D Point Clouds," in *International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 403–414.

[90] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, "Cell Broadband Engine Architecture and Its First Implementation—a Performance View," *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 559–572, 2007.

[91] C. Chung and C.-H. Yang, "A Distributed Autonomous and Collaborative Multi-Robot System Featuring a Low-Power Robot SoC in 22nm CMOS for Integrated Battery-Powered Minibots," in *International Solid-State Circuits Conference (ISSCC)*. IEEE, 2019, pp. 48–50. [Online]. Available: https://doi.org/10.1109/ISSCC.2019.8662463

[92] C. Chung and C.-H. Yang, "A 1.5-μJ/Task Path-Planning Processor for 2-D/3-D Autonomous Navigation of Microrobots," *IEEE Journal of Solid-State Circuits (JSSC)*, vol. 56, no. 1, pp. 112–122, 2020. [Online]. Available: https://doi.org/10.1109/JSSC.2020.3037138

[93] S. Darabi, N. Mahani, H. Baxishi, E. Yousefzadeh-Asl-Miandoab, M. Sadrosadati, and H. Sarbazi-Azad, "NURA: A Framework for Supporting Non-Uniform Resource Accesses in GPUs," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 1, pp. 1–27, 2022.

[94] S. Darabi, M. Sadrosadati, N. Akbarzadeh, J. Lindegger, M. Hosseini, J. Park, J. Gómez-Luna, O. Mutlu, and H. Sarbazi-Azad, "Morpheus: Extending the Last Level Cache Capacity in GPU Systems Using Idle GPU Core Resources," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 228–244.

[95] S. Darabi, E. Yousefzadeh-Asl-Miandoab, N. Akbarzadeh, H. Falahati, P. Lotfi-Kamran, M. Sadrosadati, and H. Sarbazi-Azad, "OSM: Off-Chip Shared Memory for GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 3415–3429, 2022.

[96] J. M. Domingos, N. Neves, N. Roma, and P. Tomás, "Unlimited Vector Extension with Data Streaming Support," in *International Symposium in Computer Architecture (ISCA)*. IEEE, 2021, pp. 209–222.

[97] P. H. E. Becker, J.-M. Arnau, and A. González, "KD Bonsai: ISA-Extensions to Compress KD Trees for Autonomous Driving Tasks," in *International Symposium in Computer Architecture (ISCA)*, 2023, pp. 1–13.

[98] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores," in *International Symposium on High-Performance Computer Architecture (HPCA)*, February 2018.

[99] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural Acceleration for General-Purpose Approximate Programs," in *International Symposium on Microarchitecture (MICRO)*. IEEE, 2012, pp. 449–460.

[100] F. Farshidian, E. Jelavic, A. Satapathy, M. Giftthaler, and J. Buchli, "Real-Time Motion Planning of Legged Robots: A Model Predictive Control Approach," in *IEEE-RAS International Conference on Humanoid Robotics (Humanoids)*. IEEE, 2017, pp. 577–584. [Online]. Available: https://doi.org/10.1109/HUMANOIDS.2017.8246930

[101] Y. Feng, B. Tian, T. Xu, P. Whatmough, and Y. Zhu, "Mesorasi: Architecture Support for Point Cloud Analytics Via Delayed-Aggregation," in *International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 1037–1050. [Online]. Available: https://doi.org/10.1109/MICRO50266.2020.00087

[102] W. Fornaciari, G. Agosta, D. Atienza, C. Brandolese, L. Cammoun, L. Cremona, A. Cilardo, A. Farres, J. Flich, C. Hernandez *et al.*, "Reliable Power and Time-Constraints-Aware Predictive Management of Heterogeneous Exascale Systems," in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2018, pp. 187–194.

[103] C. S. Gadde, M. S. Gadde, N. Mohanty, and S. Sundaram, "Fast Obstacle Avoidance Motion in Small Quadcopter Operation in a Cluttered Environment," in *2021 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*. IEEE, 2021, pp. 1–6.

[104] S. Ghodrati, S. Kinzer, H. Xu, R. Mahapatra, Y. Kim, B. H. Ahn, D. K. Wang, L. Karthikeyan, A. Yazdanbakhsh, J. Park, N. S. Kim, and H. Esmaeilzadeh, "Tandem Processor: Grappling with Emerging Operators in Neural Networks," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1165–1182. [Online]. Available: https://doi.org/10.1145/3620665.3640365

[105] R. Ghzouli, S. Dragule, T. Berger, E. B. Johnsen, and A. Wasowski, "Behavior Trees and State Machines in Robotics Applications," *arXiv preprint arXiv:2208.04211*, 2022.

[106] G. Gobieski, B. Lucia, and N. Beckmann, "Intelligence Beyond the Edge: Inference on Intermittent Embedded Systems," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 199–213.

[107] A. Handa, T. Whelan, J. McDonald, and A. J. Davison, "A Benchmark for RGB-D Visual Odometry, 3D Reconstruction and SLAM," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2014, pp. 1524–1531. [Online]. Available: https://doi.org/10.1109/ICRA.2014.6907054

[108] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968. [Online]. Available: https://doi.org/10.1109/TSSC.1968.300136

[109] M. Hashemi, Khubaib, E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Accelerating Dependent Cache Misses with an Enhanced Memory Controller," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 444–455, 2016.

[110] Y.-S. Hsiao, S. K. S. Hari, B. Sundaralingam, J. Yik, T. Tambe, C. Sakr, S. W. Keckler, and V. J. Reddi, "VaPr: Variable-Precision Tensors to Accelerate Robot Motion Planning," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2023, pp. 6304–6309.

[111] D. Im, G. Park, Z. Li, J. Ryu, S. Kang, D. Han, J. Lee, and H.-J. Yoo, "DSPU: A 281.6 MW Real-Time Depth Signal Processing Unit for Deep Learning-Based Dense RGB-D Data Acquisition with Depth Fusion and 3D Bounding Box Extraction in Mobile Platforms," in *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 65. IEEE, 2022, pp. 510–512.

[112] M. Kar, A. Agarwal, S. Hsu, D. Moloney, G. Chen, R. Kumar, H. Sumbul, P. Knag, M. Anders, H. Kaul, J. Byrne, L. Sarti, R. Krishnamurthy, and V. De, "A Ray-Casting Accelerator in 10nm CMOS for Efficient 3D Scene Reconstruction in Edge Robotics and Augmented Reality Applications," in *IEEE Symposium on*

Authorized licensed use limited to: Carnegie Mellon University Libraries. Downloaded on August 28,2024 at 22:32:02 UTC from IEEE Xplore. Restrictions apply.

VLSI Circuits (VLSIC). IEEE, 2020, pp. 1–2. [Online]. Available: https://doi.org/10.1109/VLSICircuits18222.2020.9163067

[113] F. Kherif and A. Latypova, "Principal Component Analysis," in *Machine Learning*. Elsevier, 2020, pp. 209–225.

[114] S. Koppula, L. Orosa, A. G. Yağlıkçı, R. Azizi, T. Shahroodi, K. Kanellopoulos, and O. Mutlu, "EDEN: Enabling Energy-Efficient, High-Performance Deep Neural Network Inference Using Approximate DRAM," in *International Symposium on Microarchitecture (MICRO)*, 2019, pp. 166–181.

[115] L. Koutras and Z. Doulgeri, "Dynamic Movement Primitives for Moving Goals with Temporal Scaling Adaptation," in *International Conference on Robotics and Automation (ICRA)*. IEEE, 2020, pp. 144–150.

[116] H. Koyuncu, "Loss Function Selection in NN Based Classifiers: Try-Outs with a Novel Method," in *2020 12th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*. IEEE, 2020, pp. 1–6.

[117] S. M. LaValle, "Rapidly-Exploring Random Trees: A New Tool for Path Planning," 1998.

[118] J. J. Leonard, D. A. Mindell, and E. L. Stayton, "Autonomous Vehicles, Mobility, and Employment Policy: The Roads Ahead," *Massachusetts Institute of Technology, Cambridge, MA, Rep. RB02-2020*, 2020.

[119] M. Likhachev, D. I. Ferguson, G. J. Gordon, A. Stentz, and S. Thrun, "Anytime Dynamic A*: An Anytime, Replanning Algorithm." in *International Conference on Automated Planning and Scheduling (ICAPS)*, vol. 5, 2005, pp. 262–271.

[120] I.-T. Lin, Z.-S. Fu, W.-C. Chen, L.-Y. Lin, N.-S. Chang, C.-P. Lin, C.-S. Chen, and C.-H. Yang, "2.5 A 28nm 142mW Motion-Control SoC for Autonomous Mobile Robots," in *2023 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 2023, pp. 1–3.

[121] D. Mahajan, A. Yazdanbakhsh, J. Park, B. Thwaites, and H. Esmaeilzadeh, "Towards Statistical Guarantees in Controlling Quality Tradeoffs for Approximate Acceleration," in *International Symposium in Computer Architecture (ISCA)*, ser. ISCA '16, 2016, p. 66–77. [Online]. Available: https://doi.org/10.1109/ISCA.2016.16

[122] V. Mayoral-Vilches, S. M. Neuman, B. Plancher, and V. J. Reddi, "Robotcore: An Open Architecture for Hardware Acceleration in ROS 2," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2022, pp. 9692–9699.

[123] V. Mayoral-Vilches, J. Jabbour, Y.-S. Hsiao, Z. Wan, A. Martínez-Fariña, M. Crespo-Álvarez, M. Stewart, J. M. Reina-Muñoz, P. Nagras, G. Vikhe, M. Bakhshalipour, M. Pinzger, S. Rass, S. Panigrahi, G. Corradi, N. Roy, P. B. Gibbons, S. M. Neuman, B. Plancher, and V. J. Reddi, "RobotPerf: An Open-Source, Vendor-Agnostic, Benchmarking Suite for Evaluating Robotics Computing System Performance," in *Proceedings of International Conference on Robotics and Automation (ICRA)*, 2024.

[124] A. K. Menon, A. S. Rawat, S. J. Reddi, and S. Kumar, "Can Gradient Clipping Mitigate Label Noise?" in *International Conference on Learning Representations (ICLR)*, 2019.

[125] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting Locality in Graph Analytics Through Hardware-Accelerated Traversal Scheduling," in *International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 1–14.

[126] M. P. Muresan, I. Giosan, and S. Nedevschi, "Stabilization and Validation of 3D Object Position Using Multimodal Sensor Fusion and Semantic Segmentation," *Sensors*, vol. 20, no. 4, p. 1110, 2020.

[127] A. Naithani, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Vector Runahead," in *International Symposium in Computer Architecture (ISCA)*. IEEE, 2021, pp. 195–208. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00024

[128] A. Naithani, J. Roelandts, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Decoupled Vector Runahead," in *International Symposium on Microarchitecture (MICRO)*, 2023, pp. 17–31.

[129] S. M. Neuman, R. Ghosal, T. Bourgeat, B. Plancher, and V. J. Reddi, "RoboShape: Using Topology Patterns to Scalably and Flexibly Deploy Accelerators Across Robots," in *International Symposium in Computer Architecture (ISCA)*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589104

[130] S. M. Neuman, B. Plancher, T. Bourgeat, T. Tambe, S. Devadas, and V. J. Reddi, "Robomorphic Computing: A Design Methodology for Domain-Specific Accelerators Parameterized by Robot Morphology," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 674–686. [Online]. Available: https://doi.org/10.1145/3445814.3446746

[131] X. Ni, L. Fang, and H. Huttunen, "Adaptive L2 Regularization in Person Re-Identification," in *2020 25th International Conference on Pattern Recognition (ICPR)*. IEEE, 2021, pp. 9601–9607.

[132] D. Nikiforov, S. C. Dong, C. L. Zhang, S. Kim, B. Nikolic, and Y. S. Shao, "RoSÉ: A Hardware-Software Co-Simulation Infrastructure Enabling Pre-Silicon Full-Stack Robotics SoC Evaluation," in *International Symposium in Computer Architecture (ISCA)*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589099

[133] H. Ohta, N. Akai, E. Takeuchi, S. Kato, and M. Edahiro, "Pure Pursuit Revisited: Field Testing of Autonomous Vehicles in Urban Areas," in *International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*. IEEE, 2016, pp. 7–12.

[134] O. A. Osman, M. Hajij, P. R. Bakhit, and S. Ishak, "Prediction of Near-Crashes from Observed Vehicle Kinematics Using Machine Learning," *Transportation Research Record*, vol. 2673, no. 12, pp. 463–473, 2019.

[135] A. Pajuelo, A. González, and M. Valero, "Speculative Dynamic Vectorization," *ACM SIGARCH Computer Architecture News*, vol. 30, no. 2, pp. 271–280, 2002.

[136] A. Pellegrini, N. Stephens, M. Bruce, Y. Ishii, J. Pusdesris, A. Raja, C. Abernathy, J. Koppanalil, T. Ringe, A. Tummala *et al.*, "The Arm Neoverse N1 Platform: Building Blocks for the Next-Gen Cloud-To-Edge Infrastructure Soc," *IEEE Micro*, vol. 40, no. 2, pp. 53–62, 2020.

[137] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler, "The Volumepro Real-Time Ray-Casting System," in *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, 1999, pp. 251–260.

[138] G. Pinto, F. Castor, and Y. D. Liu, "Understanding Energy Behaviors of Thread Management Constructs," in *International Conference on Object Oriented Programming Systems Languages & Applications*, 2014, pp. 345–360.

[139] I. Pohl, "Heuristic Search Viewed As Path Finding in a Graph," *Artificial intelligence*, vol. 1, no. 3-4, pp. 193–204, 1970. [Online]. Available: https://doi.org/10.1016/0004-3702(70)90007-X

[140] N. A. Radford, P. Strawser, K. Hambuchen, J. S. Mehling, W. K. Verdeyen, A. S. Donnan, J. Holley, J. Sanchez, V. Nguyen, L. Bridgwater *et al.*, "Valkyrie: Nasa's First Bipedal Humanoid Robot," *Journal of Field Robotics*, vol. 32, no. 3, pp. 397–419, 2015.

[141] M. Roberts, J. Ramapuram, A. Ranjan, A. Kumar, M. A. Bautista, N. Paczan, R. Webb, and J. M. Susskind, "Hypersim: A Photorealistic Synthetic Dataset for Holistic Indoor Scene Understanding," in *International Conference on Computer Vision (ICCV)*, 2021. [Online]. Available: https://doi.org/10.48550/arXiv.2011.02523

[142] N. Rohbani, S. Darabi, and H. Sarbazi-Azad, "Pf-Dram: A Precharge-Free Dram Structure," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 126–138.

[143] J. Sacks, D. Mahajan, R. C. Lawson, and H. Esmaeilzadeh, "RoboX: An End-To-End Solution to Accelerate Autonomous Control in Robotics," in *International Symposium in Computer Architecture (ISCA)*. IEEE, 2018, pp. 479–490. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00047

[144] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-Based Approximation for Data Parallel Applications," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, 2014, pp. 35–50.

[145] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and Efficient Fine-Grain Cache Partitioning," in *International Symposium in Computer Architecture (ISCA)*, June 2011.

[146] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in *International Symposium in Computer Architecture (ISCA)*, June 2013. [Online]. Available: https://doi.org/10.1145/2508148.2485963

[147] G. Sartoretti, J. Kerr, Y. Shi, G. Wagner, T. S. Kumar, S. Koenig, and H. Choset, "PRIMAL: Pathfinding Via Reinforcement and Imitation Multi-Agent Learning," *IEEE Robotics and Automation Letters*, vol. 4, no. 3, pp. 2378–2385, 2019.

[148] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey, "Can Traditional Programming Bridge the Ninja Performance Gap for Parallel Computing Applications?"

Authorized licensed use limited to: Carnegie Mellon University Libraries. Downloaded on August 28,2024 at 22:32:02 UTC from IEEE Xplore. Restrictions apply.

ACM SIGARCH Computer Architecture News, vol. 40, no. 3, pp. 440–451, 2012.

[149] D. Shah, N. Yang, and T. M. Aamodt, "Energy-Efficient Realtime Motion Planning," in *International Symposium in Computer Architecture (ISCA)*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589092

[150] D. E. Shaw, P. J. Adams, A. Azaria, J. A. Bank, B. Batson, A. Bell, M. Bergdorf, J. Bhatt, J. A. Butts, T. Correia *et al.*, "Anton 3: Twenty Microseconds of Molecular Dynamics Simulation Before Lunch," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021, pp. 1–11.

[151] S. Shen, Y. Cai, J. Qiu, and G. Li, "Dynamic Dense RGB-D SLAM Using Learning-Based Visual Odometry," *arXiv preprint arXiv:2205.05916*, 2022.

[152] Y. Shi, W. Zhang, Z. Yao, M. Li, Z. Liang, Z. Cao, H. Zhang, and Q. Huang, "Design of a Hybrid Indoor Location System Based on Multi-Sensor Fusion for Robot Navigation," *Sensors*, vol. 18, no. 10, p. 3581, 2018.

[153] J. Siderska, "Robotic Process Automation—a Driver of Digital Transformation?" *Engineering Management in Production and Services*, vol. 12, no. 2, pp. 21–31, 2020.

[154] T. Song, W. Rim, J. Jung, G. Yang, J. Park, S. Park, Y. Kim, K.-H. Baek, S. Baek, S.-K. Oh *et al.*, "A 14 nm FinFET 128 Mb SRAM With $V_M IN$ Enhancement Techniques for Low-Power Applications," *IEEE Journal of Solid-State Circuits*, vol. 50, no. 1, pp. 158–169, 2014.

[155] Statista Research Department, "Global Robotics Market Revenue 2018–2025," https://www.statista.com/statistics/760190/worldwide-robotics-market-revenue/, 2021.

[156] K. T. Sundararajan, V. Porpodas, T. M. Jones, N. P. Topham, and B. Franke, "Cooperative Partitioning: Energy-Efficient Cache Partitioning for High-Performance CMPs," in *IEEE International Symposium on High-Performance Comp Architecture*. IEEE, 2012, pp. 1–12.

[157] I. Ullah, X. Su, X. Zhang, and D. Choi, "Simultaneous Localization and Mapping Based on Kalman Filter and Extended Kalman Filter," *Wireless Communications and Mobile Computing*, vol. 2020, pp. 2 138 643:1–2 138 643:12, 2020. [Online]. Available: https://doi.org/10.1109/SIU.2009.5136492

[158] R. Vang-Mata, *Multilayer Perceptrons: Theory and Applications*. Nova Science Publishers, 2020.

[159] K. Varadarajan, S. K. Nandy, V. Sharda, A. Bharadwaj, R. Iyer, S. Makineni, and D. Newell, "Molecular Caches: A Caching Structure for Dynamic Creation of Application-Specific Heterogeneous Cache Regions," in *International Symposium on Microarchitecture (MICRO)*. IEEE, 2006, pp. 433–442.

[160] 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," *arXiv preprint arXiv:2009.06034*, 2020. [Online]. Available: https://doi.org/10.48550/arXiv.2009.06034

[161] Y. Wang, L. Zhang, and G. Chen, "Optimal Sensor Placement for Obstacle Detection of Manipulator Based on Relative Entropy," in *2019 14th IEEE Conference on Industrial Electronics and Applications (ICIEA)*. IEEE, 2019, pp. 702–707.

[162] J. T. Wen and S. H. Murphy, "PID Control for Robot Manipulators," 1990.

[163] Z. Ying, S. Bhuyan, Y. Kang, Y. Zhang, M. T. Kandemir, and C. R. Das, "EdgePC: Efficient Deep Learning Analytics for Point Clouds on Edge Devices," in *International Symposium in Computer Architecture (ISCA)*, 2023, pp. 1–14.

[164] A. Younis, L. Shixin, S. Jn, and Z. Hai, "Real-Time Object Detection Using Pre-Trained Deep Learning Models MobileNet-SSD," in *International Conference on Computing and Data Engineering (ICCDE)*, 2020, pp. 44–48. [Online]. Available: https://doi.org/10.1145/3379247.3379264

[165] B. Yu, W. Hu, L. Xu, J. Tang, S. Liu, and Y. Zhu, "Building the Computing System for Autonomous Micromobility Vehicles: Design Constraints and Architectural Optimizations," in *International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 1067–1081. [Online]. Available: 10.1109/MICRO50266.2020.00089

[166] U. Zahavi, A. Felner, J. Schaeffer, and N. Sturtevant, "Inconsistent Heuristics," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 7, 2007, pp. 1211–1216.

[167] Q.-b. Zhang, P. Wang, and Z.-h. Chen, "An Improved Particle Filter for Mobile Robot Localization Based on Particle Swarm Optimization," *Expert Systems with Applications*, vol. 135, pp. 181–193, 2019. [Online]. Available: https://doi.org/10.1016/j.eswa.2019.06.006

[168] Y. Zhou and J. Zeng, "Massively Parallel A* Search on a GPU," in *Proceedings of the AAAI Conference on Artificial Intelligence*, ser. AAAI'15. AAAI Press, 2015, p. 1248–1254. [Online]. Available: https://doi.org/10.1609/aaai.v29i1.9367

[169] Y. Zhu, "RTNN: Accelerating Neighbor Search Using Hardware Ray Tracing," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2022, pp. 76–89.

Authorized licensed use limited to: Carnegie Mellon University Libraries. Downloaded on August 28,2024 at 22:32:02 UTC from IEEE Xplore. Restrictions apply.