Analyzing the Stability of Relative Performance Differences Between Cloud and Embedded Environments

Rumen Rumenov Kolev
TTTech Auto AG, Wien, Austria
TU Wien, Austria

Christopher Helpa
TTTech Auto AG, Wien, Austria

Abstract

There has been a shift towards the software-defined vehicle in the automotive industry in recent years. In order to enable the correct behaviour of critical as well as non-critical software functions, like those found in Autonomous Driving/Driver Assistance subsystems, extensive software testing needs to be performed. The usage of embedded hardware for these tests is either very expensive or takes a prohibitively long time in relation to the fast development cycles in the industry. To reduce development bottlenecks, test frameworks executed in cloud environments that leverage the scalability of the cloud are an essential part of the development process. However, relying on more performant cloud hardware for the majority of tests means that performance problems will only become apparent in later development phases when software is deployed to the real target. However, if the performance relation between executing in the cloud and on the embedded target can be approximated with sufficient precision, the expressiveness of the executed tests can be improved. Moreover, as a fully integrated system consists of a large number of software components that, at any given time, exhibit an unknown mix of best-/average-/worst-case behaviour, it is critical to know whether the performance relation differs depending on the inputs. In this paper, we examine the relative performance differences between a physical ARM-based chipset and a cloud-based ARM-based virtual machine, using a generic benchmark and 2 algorithms representative of typical automotive workloads, modified to generate best-/average-/worst-case behaviour in a reproducible and controlled way and assess the performance differences. We determine that the performance difference factor is between 1.8 and 3.6 for synthetic benchmarks and around 2.0-2.8 for more representative benchmarks. These results indicate that it may be possible to relate cloud to embedded performance with acceptable precision, especially when workload characterization is taken into account.

2012 ACM Subject Classification Software and its engineering → Software development techniques

Keywords and phrases Performance Benchmarking, Performance Factor Stability, Software Development, Cloud Computing, WCET

Digital Object Identifier 10.4230/OASIcs.WCET.2023.8

1 Introduction

In recent years, the task complexity of automotive software has increased, and the quantity and quality of software has grown proportionally. To put this software complexity into perspective, a premium car today has more than 100 million object code instructions [6]. The development of this volume of embedded software runs into its own unique challenges. One such challenge is the need for embedded hardware to be available to the developers. Since the requirements for the software are formulated according to the embedded chipset, most performance and functional benchmarks and tests must be executed on the specific embedded hardware to determine the behaviour of safety-critical real-time systems. Even for non-safety-critical real-time applications, it is necessary to assess the overall load/latency of integrated components. Modern automotive software systems are distributed, with many software components contributing to the execution time of an algorithm at the same time. Thus, the overall WCET depends on the performance of multiple different components, yet it is not necessarily composed of the sum of the WCETs of the separate components. Instead, execution times are context-dependent, as discussed in [21]. Benchmarks, such as the ones introduced in this paper, use this context dependency to their advantage by utilizing real-life use cases to measure and compare performance.

Due to hardware and licensing costs, as well as legal and logistical problems, uninterrupted access to target hardware is rarely available to every developer. This is a critical issue since the inability to perform quick verification and tests of the code leads to a decrease in efficiency and quality. Furthermore, safety requirements require extensive testing of autonomous driving applications. It is not feasible to execute all those tests on dedicated physical hardware, and thus, cloud technology could provide a sensible solution to this issue due to its vertical (scale up/down where changes occur in the resources of a VM/container) and horizontal (scale in/out that adds/removes VMs/containers) scalability [2]. This is sufficient for functional tests but not for assessing correctness regarding temporal behaviour. The temporal behaviour can lead to different functional behaviour, leading to tests not representative of real behaviour [11].

Autonomous driving applications have been tested in the cloud, using simulations, yet the behaviour has been shown to differ from the track testing, in part due to timing issues [8]. Hence, testing entirely in the cloud has not been fully providing the required test coverage. Recently, ARM hardware became available in the cloud [14], bringing it closer to embedded hardware, which is often ARM-based as well, due to ARM's energy efficiency and high performance. This allows more similar timing. However, differences in the chip implementation, like the micro-architecture and memory subsystem layout, will lead to performance differences depending on the nature of the workloads running on the machines [19], [12].

The goal of this paper is to determine first how the performance of the cloud and the embedded hardware can be related and, secondly, whether a simple factor would be expressive enough to represent this relation.

To treat performance in the cloud as a reliable indicator of performance on the embedded hardware, using a performance difference factor$(= Perf_{embedded}/Perf_{VM})$, two conditions need to be met. First, the factor jitter must be relatively low for repeating test executions and must not be overly affected by timing outliers. Second, the factor must not differ significantly based on different inputs to the algorithms (e.g., a factor is valid for all cases from the best- to the worst-case behaviour). Thus, we investigate the stability of relative performance differences of benchmarks over the entire input range and for different workloads, executed on a cloud VM (running on dedicated hardware) and on embedded hardware and perform a root cause analysis of any observed outliers. The environments differ not only in the chip used but also in the software infrastructure, e.g., the hypervisor present in the cloud and potential differences in the operating systems. We expect that certain timing outliers exist, which would need to be taken into account in any future framework that might try to relate the cloud to embedded performance.

Should a stable factor be identified, two questions can be answered early on in the development process by analyzing the cloud performance exclusively: Would the set of all applications that should be hosted together fit into the embedded CPU, or can it already be foreseen that they are too compute-intensive? How realistic is it that the real-time applications would meet their end-to-end timing requirements on the embedded hardware?

The rest of the paper is structured as follows: Section 2 provides the relevant background information and related work, Section 3 discusses the three benchmarks and the most important implementation details and input variation of the two representative benchmarks, Section 4 presents an overview of the benchmarking environment, as well as of the different benchmark executions and an interpretation of the results, Section 5 provides a final overview of the results and a summary of the open points.

2 Background and Related work

Modern general-purpose microprocessors on the market differ in features like clock frequency, cache sizes and structure, core count, memory bandwidth, power consumption, and, in general, their architecture and micro-architecture. However, these specifications are not enough to reliably compare performance differences, as CPU performance is highly dependent on the specific workloads being executed, e.g., a higher clock frequency does not imply a better performance of the memory subsystem. Additionally, discrepancies in the performance factor might be caused by differences in the branch prediction algorithms, timing anomalies in the instruction pipeline, etc.

While considerable research has been done in recent years in relation to predicting new workloads or CPU/GPU performance (c.f. [20], [1]), it has focused on the average case. Thus, this paper aims to extend the state of the art by analyzing the performance difference factors of the best- and worst-case performance. One particular problem is how to reliably stimulate representative best-, average- and worst-case behaviour of the applications.

Most applications hosted on automotive ECUs are best-case, soft real-time (infotainment), or firm real-time systems (autonomous driving), and the deployed software is typically a combination of both. No static WCET analysis is performed due to the overly pessimistic results, especially on modern multi-core SoCs and the extensive efforts required [18]. The uncertainty of measurement results over static WCET approaches can be tolerated, as the non-hard real-time context does not require strict guarantees [21]. A low number of deadline misses can typically be tolerated in autonomous driving functions (e.g., cruise control). In this sense, autonomous driving applications are an example of firm real-time systems. Measurement-based analysis has much lower analysis efforts and leads to less hardware overprovisioning need, which would lead to prohibitive hardware costs. Therefore, this type of analysis is useful for the industry use case, where it can speed up development and reduce costs. However, some low-level causes for timing anomalies, which could lead to unexpected and hard-to-predict execution time outliers that could make it infeasible to predict the performance difference, are known and discussed by Lundqvist and Stenstrom in [12], e.g., out of order instruction execution causing domino effects. These types of anomalies were not identified in the performed experiments and are therefore not discussed in further detail. Other, higher-level anomalies are possible, e.g., preemptions due to real-time throttling caused by a FIFO scheduler. This is touched upon in Section 4.

3 Benchmark Details

General overview. While selecting benchmarks for the analysis, we focused on software relevant to the automotive field. Besides that, the main requirement is the ability to control the best-/average-/worst-case performance by varying the input.

Two algorithms were selected - a pathfinding algorithm (the A* search algorithm) and a contour detection algorithm. The A* algorithm, or variations of it, is used for path planning in mapping software. Contour detection algorithms are a vital part of the object extraction process, e.g., used for object recognition in autonomous driving.

A third, more generic benchmark was selected - EEMBC CoreMark-PRO. It utilizes a combination of integer and floating-point workloads and large datasets to stress the entire processor [7]. Due to these features, it provides a range of performance difference factors for workloads with isolated processor stress points and hence gives upper/lower bounds for factors to expect. The two representative algorithmic benchmarks exhibit a more realistic combination of different stress points.

A* algorithm. The A* algorithm was implemented according to its definition [9].

Our implementation of the algorithm works using Manhattan-style grids, represented as 2D C++ vectors, where each "cell" of the grid corresponds to a vector element. Grids are generated in the benchmark using a C++ standard library random number generator in order to block some cells to reduce possible paths and make the search environment more realistic. Maximum grid size and the seed of the RNG function are configured using input parameters. Grid generation is not timed; exclusively, the algorithm execution is.

The heuristic functions used to modify the performance of the algorithm are depicted in Table 1. Worth noting is that inconsistent, yet admissible, heuristics have been shown to expand arbitrarily more cells than an alternative A*-like algorithm in [4].

Table 1 Heuristic functions used in the A* benchmark and their effects on its performance.

heuristic function	**best-case**	**Euclidean distance**	**constant**	**inconsistent**
algorithm behaviour	uses precalculated values for the actual distance to the goal cell	calculates the Euclidean distance to the goal cell	always estimates distance 0 to the goal	randomly estimates the distance between 0 and Euclidean distance to the goal (thus, admissible)
performance impact	best case	average case, good estimate, thus close to best-case	average case, bad estimate	worst-case

Contour detection. The contour detection benchmark was implemented following the OpenCV documentation [16]. In the timed section, the input image is read, converted to grayscale, binary thresholding is applied, contours are found, and contours are drawn.

Three classes of 5120x2880px input images were selected to trigger the best-/average-/worst-case performance of the algorithm, depicted in Table 2. An additional fourth set of worst-case type images was added, with 8k resolution, to analyze the behaviour when increasing image resolution. Fifteen images of each class are used. The contour approximation mode and the contour retrieval mode also cause differences in performance [17], [15].

4 Benchmark Executions and Results

4.1 General

Environment. The code for both representative benchmarks is written in C++17 and compiled using g++ 9.4.0 for Ubuntu. For the contour detection benchmark, OpenCV 4.6.0 is used. It is important to note that the shared object files for the C++ STD and OpenCV and the binaries for each benchmark are shared between the two platforms to avoid any performance differences caused by different compiler versions/settings.

Table 2 Inputs for the contour detection benchmark and their effects on its performance.

image classes			
image type	**solid-color images**	**dashcam pictures of street traffic**	**multicolored white noise**
algorithm behaviour	only image borders are considered contours	average real-life use case	very large number of neighbouring pixels with different colours => highest possible number of contours
performance impact	best case	average case	worst case

contour approximation method		
method	**CHAIN_APPROX_SIMPLE**	**CHAIN_APPROX_NONE**
performance impact	removes redundant points: less memory intensive	all the boundary points are stored: very memory intensive

contour retrieval mode				
mode	**EXTERNAL**	**LIST**	**CCOMP**	**TREE**
performance impact	least compute-/memory-intensive	no hierarchy: less memory-intensive	2-level hierarchy: slightly more memory-intensive	all hierarchy levels: most memory-/compute-intensive

For the cloud virtual machine, a standard Microsoft Azure type D2pds v5 was chosen, which provides an entire physical core for each vCPU. The embedded hardware consists of a Janicto TDA4VMx&DRA829Vx processor connected to a J721EXCP01EVM common processor board. The specifications [3], [10] of both are listed in Table 3.

These platforms closely represent a real-world development scenario in the automotive industry. The embedded CPU is "ARMv8-A"-ISA-based, while the VM utilizes a newer "ARMv8.2+"-ISA. The cloud CPU also supports a higher clock frequency and a larger cache, including a 32MB system-level cache, which is not present on the embedded CPU. Thus, it is to be expected that both the computationally intensive (due to clock frequency) and the memory-intensive (due to memory and caching model [13]) performance of the VM would be significantly better.

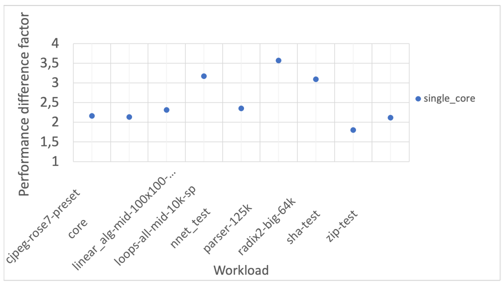
Due to the fact that the virtual machine utilizes a hypervisor and the different Linux distributions in use, we are, in reality, not analyzing the performance differences between the two different ARM chips but the whole stack (OS, hypervisor, hardware). This is acceptable and even desirable, as it is the environment that would be present in the industry-relevant use case.

Preliminary analysis. The A* benchmark was configured to execute 10 times for every heuristic and grid size, and the contour detection benchmark was configured to execute 20 times for every image and contour retrieval/approximation mode. The entire benchmarks were executed multiple times with little to no variance in the results.

A preliminary analysis was performed to evaluate how stable the execution environment is. Some potential temporal anomaly sources were identified, like scheduling and background loads. Therefore, for both the A* and contour detection benchmark executions, it was investigated whether the CPU affinity or the scheduling algorithm has any effect on the performance factor. The taskset command was used to assign the CPU affinity to a single core. The FIFO scheduling algorithm was enabled, once with maximum and once with above-average priority for the benchmark process. No significant difference in the performance factor was observed, except for some outliers caused by the FIFO scheduler throttling the OS and being forced to reschedule due to the priority of the benchmark being maximized. As these outliers do not occur with the default scheduler and the root cause is known, plots for these runs are omitted. This indicates that the measurements are highly reproducible, and there are no apparent external effects that introduce timing anomalies into the applications.

4.2 EEMBC CoreMark®-PRO

A run of CoreMark-PRO was performed to observe the behaviour of the performance difference factor when the CPU is stressed in different ways. The plot in Figure 1 depicts the results. A short description of the stress point of each workload is presented in the table in Figure 1. [7] provides more detailed descriptions of each workload. The average performance factor for



(a) CoreMark-PRO benchmark run performance factor

Workload	Stress point
cjpeg-rose7-preset	integer, compute-intensive
core	integer, compute-/memory intensive
linear_alg-mid-100x100-sp	floating-point, compute-intensive
lloops-all-mid-10k-sp	floating-point, compute-intensive
nnet_test	floating-point, compute-intensive
parser-125k	integer, memory intensive
radix2-big-64k	floating-point, compute-intensive
sha-test	integer, compute-intensive
zip-test	integer, compute-/memory intensive

(b) CoreMark-PRO workloads

Figure 1 CoreMark-PRO benchmark execution.

CoreMark-PRO workloads varies between 1.8 and 3.6, averaging around 2.52. Of interest are the 3 outliers as seen in the plot in Figure 1.

The first increase to above a factor of 3 is the workload "lloops-all-mid-10k-sp". This workload is part of the floating point suite. It is computationally intensive, as it carries out different mathematical kernels. The second outlier occurs with the integer-based workload "parser-125k", which is created in such a way that the focus of the benchmark is data structure creation and traversal [7]. Thus, it is memory intensive. The third increase occurs within the "radix2-big-64k" workload, which is also part of the floating point suite and is computationally intensive. These results show that different performance relations exist based on the workload (integer vs floating point and compute- vs memory-intensive).

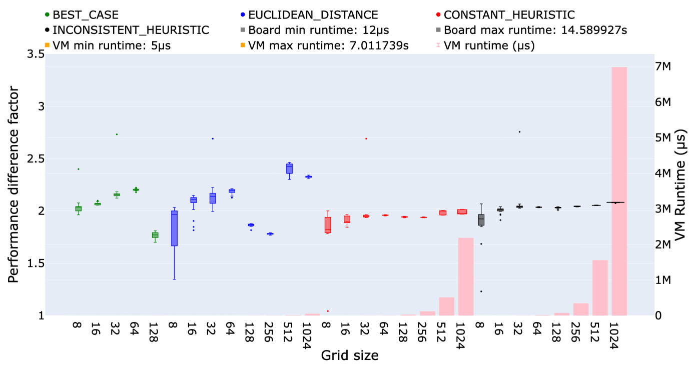
Since neither the A*, nor the contour detection benchmark is heavy on floating point computations, that is left as a possible area to explore in the future, using different benchmarks. On the other hand, both of those create, traverse and delete data structures, which leads to larger than average for the benchmarks performance difference factors for larger data structure sizes, similar to the outliers observed in the XML parser benchmark in the CoreMark suite.

4.3 A* algorithm

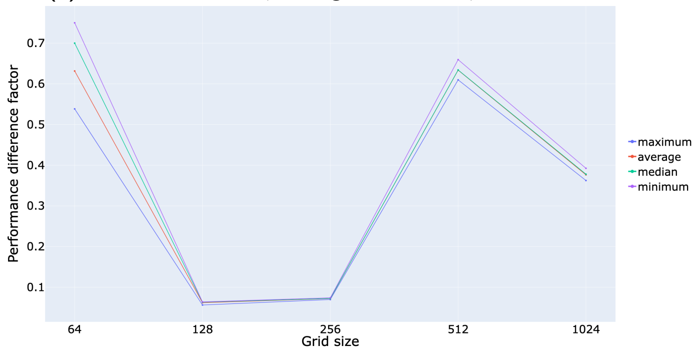
The A* algorithm benchmark was executed with 4 different seeds, with every heuristic. For every seed and grid size, 20 different grids were generated and tested. The same sequence of grids is generated when using the same seed, so executions using different heuristics use identical grids.

Only the plots for one of the 4 different seeds are shown, as the performance (factor) for the other 3 is analogous. Runs for grid sizes above 128 using the best-case heuristic were omitted, as these take the longest amount of time in the preparation phase, and the Euclidean distance heuristic behaves analogously due to their very similar runtimes.

Default benchmark executions. As depicted in plot (a) of Figure 2, the performance difference factor varies between 1.8 and 2.4 for the median of the runs for every combination



(a) Default execution, max grid size 1024, seed 1422785160



(b) Vector destruction

Figure 2 A* benchmark execution and vector destruction measurements.

of heuristic, seed, and grid size. Two interesting points: First, some outliers with factor 2.7 for grid size 32 can be observed. Second, there is a decrease in the median performance factor to 1.9 and 1.8, respectively, for grid sizes 128 and 256, for Euclidean distance and best-case heuristics. Besides those outliers, the performance difference factor appears to be fairly stable.

Average performance factor outliers for grid sizes 128 and 256. The root cause of the outlier lies in the overhead of the return of the function. At the start of the function, 3 data structures are allocated. Relevant for this discussion is a 2D standard library vector of elements of type *cell*, which is a *struct* containing 2 integers and 3 doubles. This vector is of size $gridSize * gridSize$ and is initialized with empty *cells*.

The cause for the decrease in performance was determined to be the destruction of this vector by using a complementary benchmark, which calls a function, which allocates a 2D *cells* vector of variable size and then returns. Measurements were performed from the point after the allocation was done to after the function had returned.

In plot (b) of Figure 2, it can be observed that the VM performs consistently worse than the embedded CPU for vector destruction, with a significant decrease in performance for grid sizes 128 and 256. We are not able to identify the root cause of this outlying behaviour. This slowdown of the VM explains the decrease in the average performance for these grid sizes for the Euclidean distance and best-case heuristics, as these are the fastest heuristics, where vector destruction takes a larger percentage of the overall execution time of the benchmark (24.8% and 30% respectively for grid sizes 128 and 256 on the VM). On the other hand, the constant and inconsistent heuristics take a longer time to execute, and the vector destruction takes a smaller percentage of the overall execution time of the benchmark (1% and 0.5% respectively for constant and inconsistent heuristics, grid size 128 on VM). On the embedded hardware, the vector deletion takes less than 1.5% of the overall execution time for any heuristic. This means that there is a 23% to 29% extra execution time for these grid sizes and heuristics on the VM, which corresponds to the 23% to 29% decrease in the performance factor.

Performance factor increases for minimum case. To analyse the root cause of the outliers for grid size 32, it was determined via debugging that these occur during minimum execution times caused by a specific edge case in the algorithm. In each iteration, the A* algorithm attempts to determine the neighbouring cell with the shortest path so far and the heuristically determined shortest remaining path to the goal cell. An edge case exists, where all cells neighbouring the start cell are blocked. Therefore, the goal is immediately unreachable and no more iterations can be done, so the algorithm returns, leading to the minimal execution time. These were determined to be the cases where the performance difference factor increases.

The pattern repeats because every run represented on a single plot uses the same seed to generate grids, and the heuristic functions are never actually called in these minimum cases, as all surrounding cells are blocked.

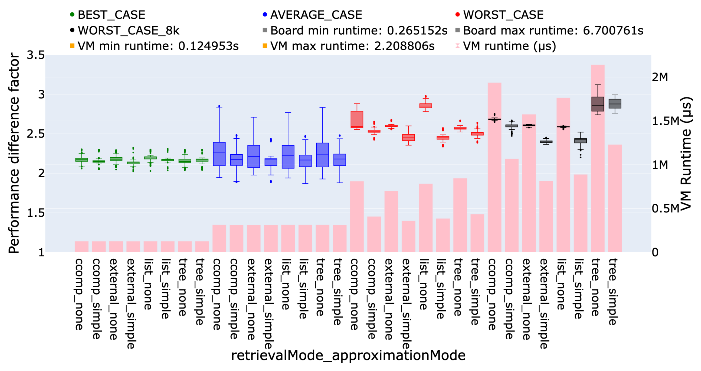
To analyze this behaviour, additional measurements of only the “surrounded” case were performed, inside and outside the function call, as well as of only the data structure initializations. Listing 1 depicts the three pairs of measurement points.

As the performance difference factor remains on average 2.2 in all of these measurements, this points towards caching differences. As the VM has a larger cache and an updated memory model [13], some differences in caching/memory operations performance are expected. This is a very probable cause for the increased factor in the minimum case of the default execution. I.e., in the default runs of the benchmark, many memory operations, which use the cache, are performed in iterations before the “surrounded” case. This leads to cache misses in the “surrounded” case. These misses do not occur in the isolated edge case measurements, as the cache does not get overwritten since every separate iteration only does the vector initialization with empty data. Thus, a higher performance factor in the “surrounded” cases during default runs is expected compared to the targeted “surrounded” case runs. The higher factor is in line with the results from other memory-intensive measurements, as discussed in e.g., Subsection 4.2 or Subsection 4.4.

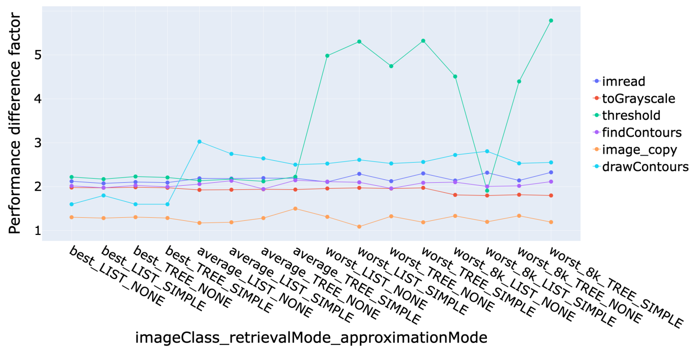
4.4 Contour Detection

Default benchmark executions. Plot (a) in Figure 3 shows that the contour detection benchmark displays a median performance factor of 2.1 to 2.2 for the best- and average-case inputs. A jitter of about 0.1 is present and can be explained by the differences between the different pairs of contour retrieval and approximation methods. However, an increase up to a factor of about 2.6 can be observed for the worst-case input image class. This corresponds to a 15%-20% increase in the performance factor.

To investigate the increase in performance factor between the average- and worst-case results, two approaches were used. First, a further set of white noise images with a larger (7680 × 4320) resolution was introduced to analyze whether the performance factor continues to increase with increasing image resolution due to e.g., differences in caching/memory subsystem or the branch prediction algorithm. Second, the separate sections of the contour detection algorithm were timed to determine specific areas that might explain the difference.



(a) Default execution with additional 8k worst-case images



(b) Median performance factor of separate sections of contour detection

Figure 3 Contour detection benchmark executions.

8k images run. It is evident from plot (a) of Figure 3 that the average performance factor does not continue to increase with increasing image resolution. It stays in the same range of 2.4 to 2.8, as it does for the 5k worst-case images.

A single outlier run using the 8k resolution worst-case images was observed, where the average performance factor decreased to 2.2 for those images, caused by a slowdown in the performance on the VM. The slowdown was not reproducible, and therefore, the outlier is not discussed further.

Timing of separate sections of the algorithm. As can be read from plot (b) of Figure 3, the binary thresholding section of the algorithm experiences an increase in performance factor from a stable 2.2 for the best- and average-case input images to 4.5-5 for the worst-case type images. The factor increases by about 250%. The other slight increase in performance factor is seen in the image read function, which has an increase in factor of about 0.2, which corresponds to a 9%-10% increase.

The thresholding function corresponds to about 2% of the complete execution time of the algorithm, while the reading of the image accounts for about 30% of the overall runtime. Thus, the increase in performance factor caused by the reading and thresholding comes out to about 10% total increase in runtime. We attribute these increases to memory model and caching differences. It should be noted that a 5K image has a size of 14.7MB when loaded into memory. Respectively, an 8K image has a size of 33.1MB. Both of these do not fit in the L1 and L2 DCache of either CPU. However, the VM utilizes a so-called system-level cache, SLC, of size 32MB. This additional caching level likely contributes to the better performance of the VM for the worst-case image class, as cache misses that would go directly to the main memory on the embedded CPU would, in most cases, go to the faster SLC of the cloud CPU. The worst-case image type induces more memory operations, especially in the binary thresholding section, where less optimization (e.g., through branch prediction) is possible due to the random nature of white noise. Thus, the better memory/caching model of the VM leads to better performance for these image types. The cachegrind tool [5] of the valgrind suite was used to simulate the different caching behaviour with or without the 32MB SLC of the VM by setting the last level cache as either 32MB or 1MB. The tool shows a 10% and 25% lower miss rate with the SLC, respectively, for the worst-case and worst-case-8k image classes. Cachegrind provides only a basic simulation of the caching behaviour, but the results appear to support the hypothesis that the 15% to 20% increase in the performance factor for the worst-case image classes is caused by caching differences. A profiler like perf that accesses hardware counters is a better choice for more accurate measurements, but the hypervisor of the cloud VM does not allow perf to read the counters.

5 Conclusion and Outlook

In conclusion, the A* benchmark shows an average factor of 2.045, with a standard deviation of 0.15. The contour detection benchmark shows an average factor of 2.31, with a standard deviation of 0.2. Lastly, CoreMark-PRO shows an average factor of 2.52, with a standard deviation of 0.56. The minimum and maximum factors for each of the benchmarks tend to move in the same range with some outliers of around 10%.

Overall, values between 1.8 and 3.6 were observed for the performance factor. For every workload, however, a separate range within this overall range can be observed, meaning that the factor range depends on the specific workload. The factor is consistent for every specific input type, i.e., no significant jitter occurs due to, e.g., timing anomalies. The difference between the average factors of the two representative benchmarks (A* and contour detection), which exhibit a combination of (integer only) compute- and memory-intensive operations, is around 13% with no vast outliers. These results indicate that it may be possible to use a single factor to relate the performance between the cloud VM and the embedded CPU with 10%-15% accuracy for this type of workload. Characterizing the workload regarding whether it is integer- or floating-point-intensive and computationally or memory-intensive could improve the accuracy.

Some deviation of the performance difference factor for a single algorithm is possible due to different input types causing the algorithm to transition from being compute-bound to memory-bound, thus changing the processor feature under stress. An example of this is the difference between the best-/average- and worst-case image type for the contour detection benchmark, where the worst-case type is more memory-intensive, as described in Section 4.4.

Some interesting topics for further research have also been identified in Section 4. There is always the possibility to add more benchmarks to increase the coverage of different types of workloads, stressing different aspects of the CPU. For example, a comprehensive floating point benchmark and a complex data structure benchmark would be of particular interest.

Another approach would be to investigate whether a similarly stable performance difference factor is present for GPUs or between x86-based CPUs and ARM-based embedded CPUs.

A topic of interest would be to analyze the stability of the performance of the cloud VM, as motivated by the single outlier with images of 8K resolution, mentioned in Subsection 4.4.

References

1 Newsha Ardalani, Clint Lestourgeon, Karthikeyan Sankaralingam, and Xiaojin Zhu. Cross-architecture performance prediction (XAPP) using CPU code to predict GPU performance. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 725–737, 2015.

2 Tao Chen, Rami Bahsoon, and Xin Yao. A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems. *ACM Computing Surveys (CSUR)*, 51(3):1–40, 2018.

3 Ampere Computing. Ampere® Altra® 64-bit multi-core processor features. `https://d1o0i0v5q5lp8h.cloudfront.net/ampere/live/assets/documents/Altra_Rev_A1_DS_v1.30_20220728.pdf`. Accessed: 2023-04-21.

4 Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of A*. *J. ACM*, 32(3):505–536, July 1985. `doi:10.1145/3828.3830`.

5 Valgrind Developers. Cachegrind: a high-precision tracing profiler. `https://valgrind.org/docs/manual/cg-manual.html/`. Accessed: 2023-05-07.

6 Christof Ebert and Capers Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, 2009. `doi:10.1109/MC.2009.118`.

7 EEMBC. CoreMark-PRO. `https://github.com/eembc/coremark-pro`. Accessed: 2023-03-07.

8 Daniel J Fremont, Edward Kim, Yash Vardhan Pant, Sanjit A Seshia, Atul Acharya, Xantha Bruso, Paul Wells, Steve Lemke, Qiang Lu, and Shalin Mehta. Formal scenario-based testing of autonomous vehicles: From simulation to the real world. In *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, pages 1–8. IEEE, 2020.

9 Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. `doi:10.1109/TSSC.1968.300136`.

10 Texas Instruments. DRA829 Jacinto. `https://www.ti.com/lit/ds/symlink/dra829v.pdf?ts=1673940212016&ref_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FDRA829V`. Accessed: 2023-04-21.

11 ISO 21448:2022 Road vehicles – Safety of the intended functionality, April 2022.

12 Thomas Lundqvist and Per Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No. 99CB37054)*, pages 12–21. IEEE, 1999.

13 Berenice Mann. Arm Architecture – Armv8.2-A evolution and delivery. `https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/arm-architecture-armv8-2-a-evolution-and-delivery`. Accessed: 2023-06-18.

14 Paul Nash. Azure virtual machines with Ampere Altra ARM-based processors – generally available. URL: `https://azure.microsoft.com/en-us/blog/azure-virtual-machines-with-ampere-altra-arm-based-processors-generally-available/`.

15 OpenCV. ContourApproximationModes. `https://docs.opencv.org/4.x/d3/dc0/group__imgproc__shape.html#ga4303f45752694956374734a03c54d5ff`. Accessed: 2023-01-16.

16 OpenCV. OpenCV contour detection. `https://docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html`. Accessed: 2023-04-25.

17 OpenCV. RetrievalModes. `https://docs.opencv.org/4.x/d3/dc0/group__imgproc__shape.html#ga819779b9857cc2f8601e6526a3a5bc71`. Accessed: 2023-01-16.

18 Stefan Schaefer, Bernhard Scholz, Stefan M Petters, and Gernot Heiser. Static analysis support for measurement-based WCET analysis. *Editors: Timothy Bourke and Stefan M. Petters Work-in-Progress-Chair: Liu Xiang, Peking University, China*, page 25, 2006.

19 Yu Wang, Victor Lee, Gu-Yeon Wei, and David Brooks. Predicting new workload or CPU performance by analyzing public datasets. *ACM Trans. Archit. Code Optim.*, 15(4), January 2019. `doi:10.1145/3284127`.

20 Yu Wang, Victor Lee, Gu-Yeon Wei, and David Brooks. Predicting new workload or CPU performance by analyzing public datasets. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(4):1–21, 2019.

21 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

A Tables and Listings

Table 3 Benchmarking environment specifications.

	Cloud VM	**Embedded CPU**
ARM CPU	Ampere Altra 64-Bit Multi-Core Processor (ARM v8.2+)	64-bit Dual-core Arm Cortex-A72 (ARMv8-A)
Number of cores	2	2
max. clock frequency	3300MHz	2000MHz
L1 cache	64KB DCache, 64KB ICache per core	32KB DCache, 48KB ICache per core
L2 cache	1MB per core	1MB shared per dual-core cluster
System-Level Cache (SLC)	32MB	-
RAM	8GiB	3.8GiB(4GiB), 512KB on-chip SRAM in MAIN domain
Operating System	Ubuntu 20.04.5 LTS, Kernel: 5.15.0-1034-azure	Arago 2021.09 (based on Yocto Linux), Kernel: 5.10.65-gdcc6bedb2c

Listing 1 Minimum case measurements pseudo-code

```
a_star_benchmark():
   benchmark_preparation
   time_measurement_start_OUTER
   a_star_algorithm_func():
       time_measurement_start_INNER
       time_measurement_start_INITS
       data structure inits
       time_measurement_end_INITS
       calculations
       time_measurement_end_INNER
       return result
   time_measurement_end_OUTER
```