Exploring the Effects of Code Optimizations on CPU Frequency Margins

Konstantinos Parasyris, Nikolaos Bellas, Christos D. Antonopoulos, and Spyros Lalis

University of Thessaly
Glavani 37, 38221 Volos
{koparasy, nbellas, cda, lalis}@uth.gr

Abstract. Chip manufactures introduce redundancy at various levels of CPU design to guarantee correct operation even for worst-case combinations of non-idealities in process variation and system operation conditions. This redundancyis implemented partly in the form of voltage/frequency margins. However, for a wide range of real-world execution scwienarios, these margins are excessive and translate to increased power and energy consumption. Among the various factors that affect the degree to which these margins are actually needed to avoid errors during program execution, the impact of compiler and source code optimizations has not been explored yet. In this work, we study the effect of such optimizations on the frequency margins and the energy efficiency of applications in the ARM Cortex-A53 processor.

Keywords: energy efficiency · compiler optimizations · frequency margins

1 Introduction

As predicted by Moore's Law, the scalability of semiconductor manufacturing process has been the driving force behind the increase in the capabilities of computer systems. However, scaling into lower nanometer geometries has led to variability of transistor characteristics, resulting into increased failure rates in modern CPUs. Conventional techniques for providing reliable execution include extra provisioning in logic and memory circuits in the form of increased voltage margins and reduced operating frequencies (so-called guardbands), as well as special error correction circuitry. But all these techniques consume more power, thus are not very attractive in light of the ambitious goal to reach exascale performance with constrained power budgets [4]. More specifically, guardbanding may increase power dissipation in the order of 35% [6]. Yet most of the time these guardbands are excessive and translate to unnecessary overhead, as the worst-case combinations that were considered at design time may appear only rarely or even not at all during the life cycle of a given CPU part.

Many factors affect the CPU margins during application execution, including the application's characteristics, the libraries used by it, the CPU microarchitecture and the environmental conditions (e.g., temperature) [10, 9]. Among these

factors, compiler and source code optimizations have not been investigated so far. It is important to analyze the impact of such optimizations though, given that these are common practice when trying to improve application performance.

The contributions of this work are the following: (i) we study the effect of common compiler optimizations on the energy efficiency and the frequency margins of four ARM Cortex-A53 processor parts; (ii) we examine the effect of memory access pattern optimizations on the energy efficiency and the frequency margins; (iii) we inspect the interaction of *SIMD* instructions on the energy efficiency and the frequency margins.

We perform bare metal executions to isolate the effect of the optimizations from the system software stack (OS). Our results show that the ARM Cortex A-53 has on average frequency margins equal to 14% of the nominal frequency. The maximum energy gain due to these frequency margins is 12%. Regarding the effect of compiler optimizations, the least optimized versions of the application typically exhibit wider margins. On the other hand, source code optimizations can increase the frequency margins, up to 4% of the nominal frequency.

The remainder of this paper is organized as follows. Section 2 presents the hardware setup and our methodology. Section 3 shows our experimental results when studying the effect of compiler optimizations. Section 4 presents the effect of source code optimizations. Section 5 presents the related work. Section 6 concludes our study.

2 Methodology

We perform our experiments on four raspberry PI 3b platforms. Each PI has a 4 core ARM Cortex A-53 in-order processor running at a nomimal frequency of $1200Mhz$, with a nominal supply voltage of $1.2V$. Cores feature a 64KB L1 data cache and a 64KB L1 instruction cache, a 512KB L2 Cache and 1GB LPDDR2 RAM (900Mhz). The power consumption of the entire platform is measured using an external data acquisition USB device [8]. To accurately capture the behavior of the different applications and optimizations without the interference of the system software stack (which can introduce significant non-determinism), we use a bare metal environment, called Circle [11].

To quantify the frequency margins of each CPU part, we identify the maximum frequency that can be reached while still achieving correct execution (f_{max}). This is done, for each application benchmark, using a binary search algorithm, which determines f_{max} within a range $[low, high]$. Initially, we set $low = 1200Mhz$ and $high = 1500Mhz$, and set f_{max} equal to the middle of the interval ($1350Mhz$). Noticeably, although we increase the frequency, we do not increase the supply voltage. During the execution of the application, we detect any hardware traps raised (e.g., due to the execution of an illegal instruction) and infinite loops (if the execution takes much longer than the time required to run the application at the nominal frequency). If the application runs successfully, the produced output is compared against the correct golden output in order to detect any Silent Data Corruptions (SDCs).

To account for any non-deterministic behavior during execution, we run the application 1024 times for each tested frequency, which provides a confidence level of 99% and an error margin of 2%. If all runs complete successfully, the region $[low, f_{max}]$ is marked as safe, the low bound is increased to $low = f_{max}$, and f_{max} is adjusted accordingly (to the middle of the interval). Else, if erroneous behavior is detected, we mark the region $[f_{max}, high]$ as unsafe, and the high bound is decreased to $high = f_{max}$. The algorithm terminates when the interval width becomes less than $5mV$. No human intervention is needed, since we reboot after a CPU Crash using an external hardware watchdog.

3 Compiler Optimizations Analysis

In this work we use the *gcc 4.9.3* compiler. While modern compilers provide users with specific options to optimize their code, individual optimizations are usually grouped in higher-level options, such as O0, O1, O2, O3, Os. Our study only considers these options.

We use several applications/kernels taken from various benchmark suites [**?**,5, 12]. In this study, we analyze Sobel, DCT, Inversek2j, Blackscholes, Swaptions, Fluidanimate, Sjeng and Libquantum. Sobel is a 2D filter for edge detection in images. Discrete Cosine transform (DCT) is a module of the JPEG compression and decompression algorithm. Inversek2j is a robotics benchmarks that calculates the angles of a 2-joint arm using the kinematic equation. Blackshcoles implements a mathematical model for a market of derivatives, which calculates the buying and selling of assets to as to reduce the financial risk. Fluidanimate applies the smoothed particle hydrodynamics method to compute the movement of fluid in consecutive time steps. Swaptions uses the Heath-Jarow-Morton framework to price a portofolio of swaptions. Sjeng is a chess-player application that finds the next move via a combination of alpha-beta and priority proof-number tree searches. Finally, Libquantum simulates a quantum computer.

Compiler optimizations aim at improving performance, we first analyze the effects of the different optimization levels (O0, O1, O2, O3, Os) on the execution time and energy consumption of our benchmark applications. Increasing the optimization level augments the previous set of optimizations with additional ones. In the case of Os, the compiler uses most, but not all, of the O2 optimizations, together with some extra optimizations that decrease the size of the executable.

Figure 1 shows the normalized energy consumption and execution time of the different compiler optimization levels with respect to O0. As expected, the higher the compiler effort the greater the performance and the energy gain. DCT presents the higher speedup when using the $O3$ optimizations. On the other hand, Inversek2j shows almost no speedup when compiled with increasing optimization levels. This is because it extensively uses trigonometric functions that are included in an already optimized version of the standard C library. According to our measurements, the different optimization levels do not impact CPU power consumption in a significant way, except of the case of Os level, which in some applications (blackscholes, DCT, inversek2j) increases the power

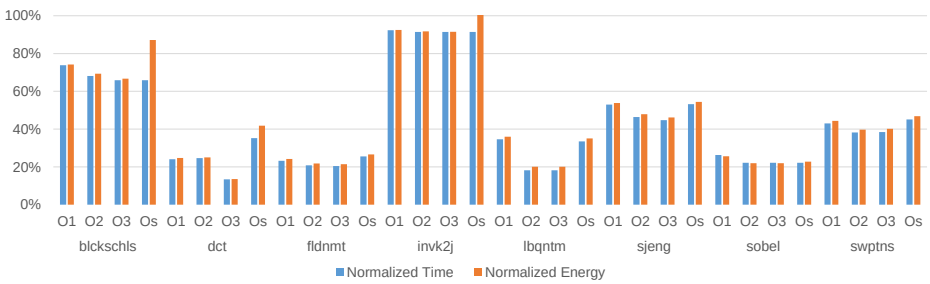


Fig. 1. Execution time and energy consumption of the application benchmarks for the different compiler optimization levels, relative to O0.

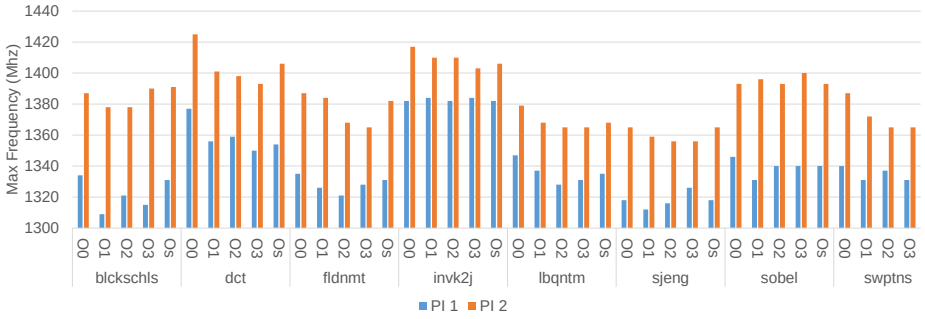


Fig. 2. Maximum frequency at which the application benchmarks run correctly for each compiler optimization level, on two of the raspberry PIs.

consumption. This is due to the instruction selection performed on this optimization level as well as that alignment and function inlining is not performed. In any case typically, the larger energy gains that are achieved when using higher optimization levels are mainly due to the reduced execution times.

Figure 2 illustrates the experimentally identified f_{max} for the different optimizations levels, on two of the raspberry PIs; the results for the other two PIs are similar, and are not shown here for brevity. The exploitable extra frequency ranges from 9% to 19% of the nominal CPU frequency ($1200Mhz$). The highest frequency at which all applications can be executed reliably, is equal to $1309, 1356, 1346, 1356Mhz$ for the four raspberry PIs, corresponding to a CPU part-specific *static* frequency margin of $109, 156, 146, 156Mhz$, respectively. The workload-specific dynamic frequency margin for the four raspberry PIs is equal to $75, 69, 69, 69Mhz$ respectively

Different optimization levels impact the dynamic frequency margin and can increase or decrease f_{max} by up to $32Mhz$ for a given application. Interestingly, O0 has a wider margin than higher optimization levels for the same application, in 62.5% of the configurations (combinations of different CPU parts and different applications). Despite the increased f_{max} of O0, the decrease in the execution time due to the extra frequency margin is relatively small, resulting in lower energy gains compared to higher optimization levels. Thus, using higher opti-

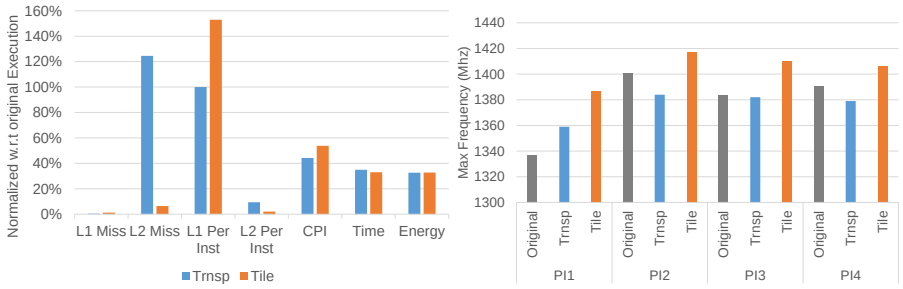


Fig. 3. (a) Performance metrics and energy concumption of the transposed and tiled MM versions, with respect to the original implementation. (b) Maximum frequency for all raspberry PIs and all MM implementations.

mization levels is more beneficial not only in terms of performance but also in terms of energy gains, even though these have smaller frequency margins than O0. When comparing the remaining optimization levels (O1,O2,O3,Os) there is no dominant optimization level in terms of frequency margins. On the other hand, in 80% of the total cases $O3$ is the most energy efficient optimization level.

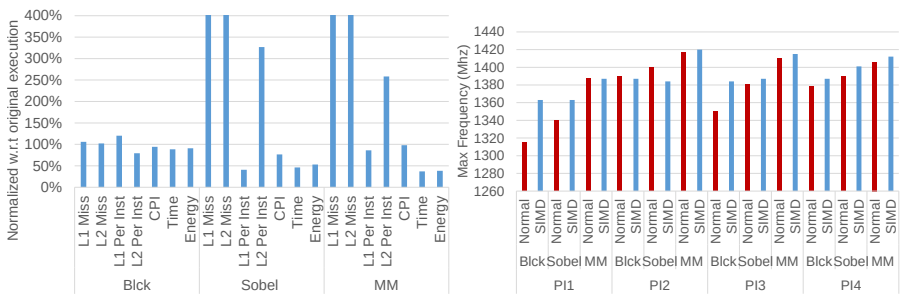
4 Source Code Transformations

Developers very often try to reduce the execution time of their applications by employing more efficient algorithms, optimizing memory accesses, reducing the number of instructions, or using special instructions for parallel processing and vectorization. In this study, we optimize a Matrix Multiplication (MM) kernel by using more efficient memory access patterns as well as Single Instructions Multiple Data (SIMD) instructions. In both cases we observe the effects of the optimizations on the energy efficiency, the execution time and the frequency margins of the different benchmark versions.

4.1 Memory Access Pattern Optimizations

The matrix multiplication (MM) kernel performs multiplication between two floating point matrices ($C = A * B$). We consider three different implementations/versions. The so-called original version accesses the first matrix (A) in a row-wise fashion and the second matrix (B) in a column wise fashion. The second implementation, performs a multiplication with the transposed B^T matrix, which is allocated on a new 2D-array. Finally, the third version uses a tiled version of the matrix multiplication. The size of the tile is equal to the cache line size (64 bytes).

Figure 3a presents the performance metrics and energy consumption of the transposed and tiled MM versions, normalized to the original implementation. As expected, both optimized versions have significantly lower L1-cache misses.



(a) (b)

Fig. 4. (a) Normalized performance metrics and energy consumption of the three benchmarks, with respect to the implementations without SIMD instructions. (b) Maximum frequency for all raspberry PIs and benchmarks.

They also demonstrate a significantly decreased CPI, which directly translates to performance and energy gains.

Figure 3b presents the maximum frequency of the different MM versions. In contrast to the compiler optimization analysis, where the non-optimized versions exhibit larger frequency margins, the memory access optimizations present mixed results. On the one hand, in all raspberry PIs, the largest margins are found for the tiled MM version, which in one of the raspberry PIs (PI1) yields an increase on the maximum frequency of up to $50Mhz$ compared to the original version. On the other hand, in three out of four PIs, the transposed MM version has lower frequency margins than the original. Also, the extra frequency margins result on average to an additional performance gain of 2.5%. We also observe margin variations across different parts, this difference can reach up to $63Mhz$ when comparing the original version of PI1 with the same version on PI2.

4.2 SIMD Optimizations

We use SIMD instructions to optimize the execution time and energy efficiency of Blackscholes, Sobel and tiled MM. Figure 4a presents the performance metrics and energy consumption, relative to the normal versions of the benchmarks without SIMD instructions. Figure 4b shows the frequency margins of the benchmarks for the four raspberry PIs.

As can be seen, when using SIMD instructions the execution time of Sobel and MM is decreased to 36% and 46% of the normal versions, respectively. This speedup is mirrored to energy gains since the power consumption does not increase significantly when using SIMD instructions. Blackscholes does not show any reduced execution time because many math functions used by that benchmark do not have a SIMD equivalent function. In PI1 the SIMD version of the blackscholes benchmark greatly increase the frequency margin by $44Mhz$. This increase in frequency provides an extra performance gain of 3.5% on top of the performance gain obtained by the SIMD instructions. In general, the use of SIMD instructions actually increases the maximum frequency by *1%* on average.

5 Related Work

In [1] the authors explore the effects of compiler optimization on the vulnerability of HPC parallel applications in the presence of radiation-induced soft errors which effect the Static Random Access Memory. Their work focus on what happens when errors occur, in our case we identify the maximum frequency in which timing errors do not occur for different optimization.

Many research approaches have emerged in the last few years that relax conservative guardbands to improve energy efficiency. Prior work focusing on commercially available chips include [9, 2, 3, 10, 7, 13]. In particular in [9] the authors present an automated system-level analysis on multi-core CPUs based on the ARMv8 64-bit architecture when pushed to operate in scaled voltage conditions. Due to the manifestation of SDCs before system crashes, the authors propose a severity function that can predict safe, SDC-free undervolt levels for each core of the processor. Based on this function and the corresponding core V_{min} resulted from the offline characterization, they produce a linear regression model that tries to predict the safe V_{min} of a core for any workload. The same authors present a study for two commercial x86-64 microprocessors [10]. The heuristics presented in [2] and [3] that dynamically reduce voltage margins while always preserving safe operation, are based on the error correction ECC hardware built on modern processors such as the server-class Intel Itanium 9560. The rate of ECC correctable errors is used as an indicator on how to adjust the V_{dd} voltage. Authors in [13] exploit the large margins available when only one core in a server-class 8-core Power7+ processor is utilized, turning under-utilized margin into power and performance benefits. A study of the voltage margins on several Kepler and Fermi GPUs is presented in [7]. They show that high energy efficiency can be achieved by shaving conservative guardbands in modern GPUs. In our work we focus on the frequency margins in contrast to these works which focus on the voltage margins. To the best of our knowledge we are the first that study the effect of optimizations to the margins of the system.

6 Conclusions

The impact of compiler optimizations on applications performance have been widely studied in the past. However, as we approach the exascale era, it can be worthwhile to understand the new trade-offs between application energy consumption and safety margins. Our study on four raspberry PIs equipped with an ARM Cortex A53 processor reveal wide frequency margins, up to 18% of the nominal operating frequency, as well as considerable margins variations across different CPU parts. Interestingly, typically non-optimized compiler code demonstrates wider margins than the optimized one. Moreover, memory optimizations which greatly increase the performance of an application also increase the width of the frequency margins. Finally, using SIMD instructions usually increases the frequency margins by a factor of 1%.

References

1. Ashraf, R.A., Gioiosa, R., Kestor, G., DeMara, R.F.: Exploring the Effect of Compiler Optimizations on the Reliability of HPC Applications. In: In Processing on the International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 1274–1283 (2017)
2. Bacha, A., Teodorescu, R.: Using ECC Feedback to Guide Voltage Speculation in Low-Voltage Processors. In: In Proceedings of 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). pp. 306–318 (Dec 2014). https://doi.org/10.1109/MICRO.2014.54
3. Bacha, A., Teodorescu, R.: Dynamic Reduction of Voltage Margins by Leveraging On-chip ECC in Itanium II Processors. SIGARCH Comput. Archit. News **41**(3), 297–307 (Jun 2013). https://doi.org/10.1145/2508148.2485948, http://doi.acm.org/10.1145/2508148.2485948
4. Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., et al.: Exascale computing study: Technology challenges in achieving exascale systems. Tech. rep., DARPA IPT. (2008)
5. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC Benchmark Suite: Characterization and Architectural Implications. In: Proceedings of the 17th international conference on Parallel architectures and compilation techniques (PACT). pp. 72–81. ACM (2008)
6. Das, S., Roberts, D., Lee, S., Pant, S., Blaauw, D., Austin, T., Flautner, K., Mudge, T.: A Self-Tuning DVS Processor Using Delay-Error Detection and Correction. Solid-State Circuits, IEEE Journal of **41**(4) (2006)
7. Leng, J., Buyuktosunoglu, A., Bertran, R., Bose, P., Reddi, V.J.: Safe Limits on Voltage Reduction Efficiency in GPUs: A Direct Measurement Approach. In: In Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). pp. 294–307 (2015). https://doi.org/10.1145/2830772.2830811
8. Measurement Computing: USB 205 Data Acquisition USB Device
9. Papadimitriou, G., Kaliorakis, M., Chatzidimitriou, A., Gizopoulos, D., Lawthers, P., Das, S.: Harnessing Voltage Margins for Energy Efficiency in Multicore CPUs. In: Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, (MICRO). pp. 503–516 (2017)
10. Papadimitriou, G., Kaliorakis, M., Chatzidimitriou, A., Magdalinos, C., Gizopoulos, D.: Voltage Margins Identification on Commercial x86-64 Multicore Microprocessors. In: In Proceedings of the 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS). pp. 51–56 (2017). https://doi.org/10.1109/IOLTS.2017.8046198
11. rsta2: Circle: A c++ bare metal programming environment for the raspberry pi, https://github.com/rsta2/circle
12. Yazdanbakhsh, A., Mahajan, D., Lotfi-Kamran, P., Esmaeilzadeh, H.: AXBENCH: A Multi-Platform Benchmark Suite for Approximate Computing. IEEE Design & Test (2016)
13. Zu, Y., Lefurgy, C.R., Leng, J., Halpern, M., Floyd, M.S., Reddi, V.J.: Adaptive Guardband Scheduling to Improve System-Level Efficiency of the POWER7. In: In Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). pp. 308–321 (2015)