The Impact of Parallel Programming Interfaces on the Aging of a Multicore Embedded Processor

Angelo Nery Crestani Vieira*, Paulo Silas Severo de Souza*, Wagner dos Santos Marques*,
Marcelo da Silva Conterato*, Tiago Coelho Ferreto*, Marcelo Caggiani Luizelli†,
Arthur Francisco Lorenzon†, Antonio Carlos S. Beck Filho‡, Fábio Diniz Rossi§, and Jorji Nonaka¶
*Pontifical Catholic University of Rio Grande do Sul, Brazil
†Federal University of Pampa, Brazil
‡Federal University of Rio Grande do Sul, Brazil
§Federal Institute of Education, Science and Technology Farroupilha, Brazil
¶RIKEN Center for Computational Science, Japan
E-mail: fabio.rossi@iffarroupilha.edu.br

***Abstract*—In order to meet the increasing performance demand of applications, the amount of cores in a single chip package has been increasing. However, the heat has been rising at a higher scale, which accelerates the aging process in modern processors. Therefore, wisely balancing the use of resources is important to extend its longevity. Frequency performance stagnates after a certain amount of concurrent threads starts executing. In such cases, the only result is a temperature rise that directly influences the aging process, reducing the processor lifetime. This unbalance between threads can be originated from many factors, which includes the way threads communicate and synchronize. Considering that those characteristics are related to the Parallel Programming Interface (PPI) used to parallelize the application, this work proposes to evaluate three widely used PPIs executing on an embedded multicore. We show that, depending on the characteristic of the application, by only switching from one PPI to another, it is possible to reduce the effects of aging. For that, we have developed a model based on the Arrhenius equation. We show that OpenMP has a lower impact on the processor aging for memory-bound applications: up to 38% and 68% lower than PThreads and MPI, respectively. On the other hand, PThreads presents the lowest impact on the processor aging for CPU-bound applications.**

***Keywords*—Aging, Embedded Systems, Parallel Programming.**

I. Introduction

The number of cores in a single chip has been rising to meet the performance demands of applications that run on high-end embedded systems (e.g., facial identification, object tracking, human body interaction, and neural networks). However, with the end of Dennard scaling [1], the power dissipated per area increases at each new processor generation. Therefore, heat dissipation has become even more relevant. Besides the well-known issues (e.g., cooling), it also accelerates the aging process of the hardware elements, reducing their lifetime. Aging is tremendously impacted by higher working temperatures. For example, even a small increase of 10-15°C in the working temperature may decrease the chip's lifespan by half [2]. On top of that, processor intensive applications may produce hotspots that will stimulate even more the aging process. Considering that aging makes systems more susceptible to several sorts of failures (e.g., electromigration, dielectric breakdown, and stress migration) [3], managing die temperature and keeping it as low as possible is key to reduce these issues.

At the architectural level, when running a parallel application, performance tends to increase. However, the processor temperature increases proportionally to the number of extra threads, since additional hardware components are activated (e.g., cores and caches). Therefore, there is a trade-off between performance gains and temperature rise, so to improve the processor lifetime it is necessary to consider the aging acceleration factor and how long the processor performs a given application, not one or another individually.

However, both of them are directly associated with the amount of threads executing, how they are distributed over the cores and the way they communicate and synchronize [4]. All these characteristics are directly related to the Parallel Programming Interface (PPI) used to parallelize the application. PPIs are used to speed up the development of parallel applications and make it as transparent as possible to the programmer, being OpenMP - Open Multi-Processing [5], PThreads - POSIX Threads [6], or MPI - Message Passing Interface [7] the most popular ones. Each one of these has different characteristics concerning the management (i.e., creation and finalization of threads/processes), workload distribution, communication, and synchronization. Therefore, each PPI will influence the application performance and processor temperature in different ways, which will affect the processor aging.

Based on the above, this paper assess how the aforementioned PPIs impacts aging, with the following contributions: (i) evaluate a mathematical model that estimates the processor degradation factor based on the heat wasted and the execution time of the application; (ii) perform an extensive analysis of several algorithms implemented using three different parallel programming interfaces widely used nowadays (PThreads, OpenMP, and MPI); and (iii) demonstrate that, depending on the characteristic of the application (i.e. CPU- or Memory-bound), it is possible to reduce the effects of aging by only switching from one PPI to another. The remainder of the paper is organized as follows. Section II discusses the related work and states our work. Section III presents and discusses the proposed model to estimate processor aging. The evaluation methodology and testbed is described in Section IV, while Section V presents and discusses the results. Finally, Section

978-1-7281-0397-6/19/$31.00 ©2019 IEEE

VI draws the final considerations and future work.

II. Related Work

a) *Processor aging*: M. Namaki-Shoushtari et al. [8] present ARGO, an aging-aware method uniformly allocate variables in the register files (RFs) of GPGPUs to disperse the heat dissipation more uniformly, impacting temperature and decreasing hardware aging. A. Bartolini et al. [9] propose a distributed and self-calibrating model, which uses thermal control on multicore architectures. The model was improved in [10] to choose the fitting processor working frequency to decrease the temperature. F. Reghenzani et al. [11] show a data-driven controller based on a model to optimize the resource allocation under specific thermal constraints. D. Zoni and W. Fornaciari [12] evaluates a power-gating approach to improve the hardware lifetime in network-on-chip buffers. A. Marongiu et al. [13] propose a workload allocation approach for reducing the processor aging when running OpenMP applications. F. Mulas et al. [14] investigates a thermal balancing scheme that employs task migration to control the cores' temperatures for MPSoCs architectures. T. Chantem et al. [15] perform a solution for allocating and scheduling tasks on an MPSoC architecture to overcome the processor aging. S. Corbetta and W. Bornaciari [16] explore the impact of different instruction allocation policies on the processor aging.

b) *Comparison between parallel programming interfaces:* Many works have evaluated the performance improvements and energy consumption considering different PPIs. Mallón et al. [17] evaluate the performance of MPI, UPC (Unified Parallel C), and OpenMP in multicore architectures through a subset of the NAS Parallel Benchmark. Balladini et al. [18] analyze the influence of OpenMP and MPI on the energy consumption and the behavior of systems at different clock frequencies of CPUs. Hua and Yang [19] present a performance analysis of OpenMP and MPI. Fellows et al. [20] compare the performance and energy consumption of OpenMP and Intel TBB (Threading Building Blocks) on embedded processors. Lorenzon, et al. [21], [22] show the impact of OpenMP applications on the static power consumption of the processor and memory system. Wang et al. [23] investigate the energy consumption of parallel applications implemented with OpenMP on the Intel Haswell processor microarchitecture. Salehian et al. [24] present a performance evaluation of OpenMP, Intel Cilk Plus, and C++11 on different multicore systems. Lima et al. [25] explore the performance and energy consumption of the OpenMP runtime system when executing on a non-uniform memory access platform.

c) *Our Contributions:* While the works discussed in item ***a*** propose solutions to reduce processor aging, works in item ***b*** evaluate the impact of PPIs on performance and energy only. Therefore, this is the first work that assess the impact of different PPIs on aging, evaluating different algorithms using an embedded platform. Therefore, our results will serve as a guide to decide what and when use each parallel programming interface when the aging is the metric of interest.

III. Aging Modeling

We estimate the processor aging using the Arrhenius equation [26], which determines how the increase in temperature stimulates the hardware component aging. Such equation applies: a regular processor temperature denoted in Kelvin (T_u); a Boltzmann's constant k that describes the relation between absolute temperature and the kinetic power included in each molecule of an ideal gas [27]; the minimum amount of power needed to start a chemical reaction (E_a) [28]; and the natural logarithms base (e). Based on the above, the aging acceleration factor per second (*Af*) at a current temperature (T_t denoted in Kelvin) is provided, as shown in Equation 1.

$$A_f = e^{\left(\frac{E_a}{k} * \left\{\frac{1}{T_u} - \frac{1}{T_t}\right\}\right)} \tag{1}$$

We used the *lm_sensors* (Linux monitoring sensors) application to get the current processor temperature per second directly from the hardware counters. The total processor aging can be represented as an integral of the *Af* over the whole application execution time (*Et*), as described in Equation 2.

$$ProcAging = \int_{i=0}^{Et} Af \tag{2}$$

Even thought the model above does not define a raw number for the circuit degradation (e.g. number of years before wear out), it gives a degradation factor over time, which can be efficiently used to compare aging between different setups.

IV. Methodology

Parallel programming interfaces: In our study, we consider the three parallel programming interfaces widely used nowadays: OpenMP, POSIX Threads, and MPI. The first two are used for shared memory programming, while the latter considers distributed memory programming (however, when executed over multicore architectures, MPI performs a bypass in local memory [29] to lower latency). **OpenMP** is a multi-platform API for shared-memory parallel programming in C/C++ and Fortran. It uses a set of compiler directives, library functions, and environment variables. OpenMP applies the Fork-Join model, where there is a main flow of execution. When the application needs to parallelize the code, new threads are created by Fork to distribute the work, forming parallel regions. When the processing of a parallel region ends, the threads are synchronized via Join [5]. **POSIX Threads** (PThreads) is a standard API for multi-threading programming in C/C++. The functions of the PThreads library allow fine adjustment concerning the grain size of the workload. The programmer explicitly defines the creation/termination of threads, the distribution of workload, and the control of execution [6]. **MPI** (Message Passing Interface) provides functions for C, C++, and subroutines for Fortran-77 and Fortran-95. An MPI program is defined as a group of processes that can exchange messages among them [7]. MPI is similar to PThreads concerning the obligation to explicitly exploit the parallelism. The programmer must use send and receive operations to define the cooperation among processes.

Benchmarks: Nine parallel applications from assorted benchmark suites and domains were chosen: **DFT:** Discrete Fourier Transform turns a finite sequence of equally-spaced instances of a function into a same-length sequence of equally-spaced instances. **PI:** Calculation of the Pi Number using infinite series. **GaS:** Gauss-Seidel is an iterative method for solving systems of linear equations. **MM:** Matrix Multiplication multiplies two square matrices (A and B) and stores the result in matrix C. **TR:** Turing Ring describes a space system

TABLE I: Embedded platform specifications.

Component	Specification
CPU	4 ARM Cortex-A53, 1.2GHz
Memory (RAM)	1GB LPDDR2 SDRAM
Power Supply	+5V Micro USB

that predators and prey interact in one location. **OE:** Odd-Even is a sorting algorithm, based on bubble sort that compares pairs of elements. **HS:** Harmonic Sum consists of a finite series that calculates the sum of arbitrary precision after the decimal point. **JC:** Jacobi determines the solution of linear systems involving a large percentage of zero coefficients. **GS:** Gram-Schmidt is a method for orthonormalizing a set of vectors in an inner product space. The applications were classified into three groups: CPU-Bound (DFT, PI, and GaS), balanced use of the resources (MM and TR), and memory-bound (OE, HS, JC, and GS). In order to classify these applications, Performance Application Programming Interface (PAPI [30]) was used to collect and analyze data related to the number of cycles, executed instructions, and the number of memory accesses (hits and misses in the L1 and L2 caches, and RAM).

Benchmark Implementation: The applications were implemented using the C language. Since the strategy used to parallelize the application influence its behavior during execution, we have followed the guidelines indicated by [5], [6], [7], and [31]. Thus, the OpenMP implementations were parallelized using parallel loops, splitting the number of loops iterations (*for*) between the threads. As the authors discuss in [5], this approach is ideal for applications that compute on uni and bi-dimensional structures, which is the case. On the other hand, as indicated by [6], [7], and [31], the approach using parallel tasks was utilized in PThreads and MPI implementations. In such cases, the loop iterations were distributed based on the best workload balancing between threads/processes. Moreover, the communication between MPI processes was implemented by using non-blocking operations, to provide better performance, as showed in [29].

Execution Environment: The experiments were performed on a multicore embedded processor, as shown in Table I. We used the Operating System Linux Ubuntu, Kernel v. 4.4.38-v7+armv7l and the compiler GCC5.4.0. The following distributions were used: OpenMP 4.0, PThreads/POSIX.12008, and OpenMPI 1.6. The benchmarks were split into 2, 4, and 8 threads/processes. Each configuration (algorithm, number of threads, and parallel programming interface) was executed ten times, with a standard deviation in both power consumption and performance of less than 0.5%. All algorithms had large inputs, which allowed the execution time to be as long as possible. Both the hardware and the operating system were configured to be in the same state at the beginning of each test.

V. Results and Discussion

Figure 1 presents the results for each application from the benchmark set. Each chart shows the processor aging in raw numbers (given by Equation 2) and the number of threads (*x-axis*), for each parallel programming interface. The sub-figures are organized in order, from the CPU-bound applications to the Memory-bound ones. Furthermore, Table II shows the results for the execution time and processor temperature (geometric mean for each benchmark class) for each PPI and number of threads.

TABLE II: Results for the geometric mean of each benchmark class.

	#Threads	Temperature (°C)			Execution Time (s)			Processor Aging		
		PT	*OMP*	*MPI*	*PT*	*OMP*	*MPI*	*PT*	*OMP*	*MPI*
CPU-B	2	56	59	61	533	566	543	9603	13018	14665
	4	68	68	69	249	290	306	11223	13050	14700
	8	71	70	69	195	233	295	10778	11923	14178
Balanced	2	59	60	60	474	464	512	10910	11611	12806
	4	68	68	69	240	254	288	10816	11447	13859
	8	66	66	65	284	316	378	11085	12361	13638
Mem-b	2	58	59	61	277	233	423	5832	5374	11519
	4	65	65	66	176	142	341	6336	5128	13331
	8	65	65	64	209	130	443	7541	4680	14642

Initially, we discuss the CPU-bound applications (DFT, PI, and GaS – Figure 1), in which PThreads presented the best results regardless of the number of threads. Although these applications are processor intensive and do not present higher communication demands, the overhead of managing the threads and the workload distribution in OpenMP was higher than the gains achieved by the parallelization [32]. In the same way, the large difference between PThreads to MPI is due to the overhead for creating the MPI processes and then performing send/receive operations. Moreover, because of the specific characteristics of the applications, it was not possible to implement the MPI versions using non-blocking messages, even though they do not have high communication demands, increasing even more the difference between PThreads and MPI. As one can observe in Table II, the best configuration is PThreads with only two threads, that is, the parallel programming interface and the number of threads that provides the lowest processor aging. Comparing to the best result achieved by OpenMP (8 threads) and MPI (8 threads), it represents a reduction of 20% and 32% in the processor aging, respectively.

For the applications with balanced use of resources (MM and TR), PThreads and OpenMP presented a similar impact on the processor aging. In this case, the best configuration found was with PThreads running 4 threads (Table II): it is only 5% and 15% better than the best result for OpenMP (4 threads) and MPI (2 threads). For this specific comparison between PThreads and OpenMP, the programmer may take into account secondary factors during the development process, such as ease of programmability, code reuse, and so on. However, the same is not true for the memory-bound applications (OE, HS, JC, and GS). In this case, OpenMP presented the lowest processor aging, regardless of the number of threads. Comparing to PThreads, the difference is related to the impact of context switching imposed by the use of a mutex to ensure synchronization, since the performance of this mechanism depends on the architecture and Operating System used [33]. Due to this overhead, PThreads was 38% slower than OpenMP for the memory-bound applications (on the average, for 8 threads), which impacted the processor aging.

Finally, analyzing the whole scenario for each benchmark class and the geometric mean of the entire benchmark set, Figure 2 shows the results of the processor aging of OpenMP and MPI normalized to PThreads (represented by the black line). One can observe that PThreads showed to be the best parallel programming interface for CPU-bound applications regardless of the number of Threads. In the most significant case (2 threads), it presented an impact on the aging process which

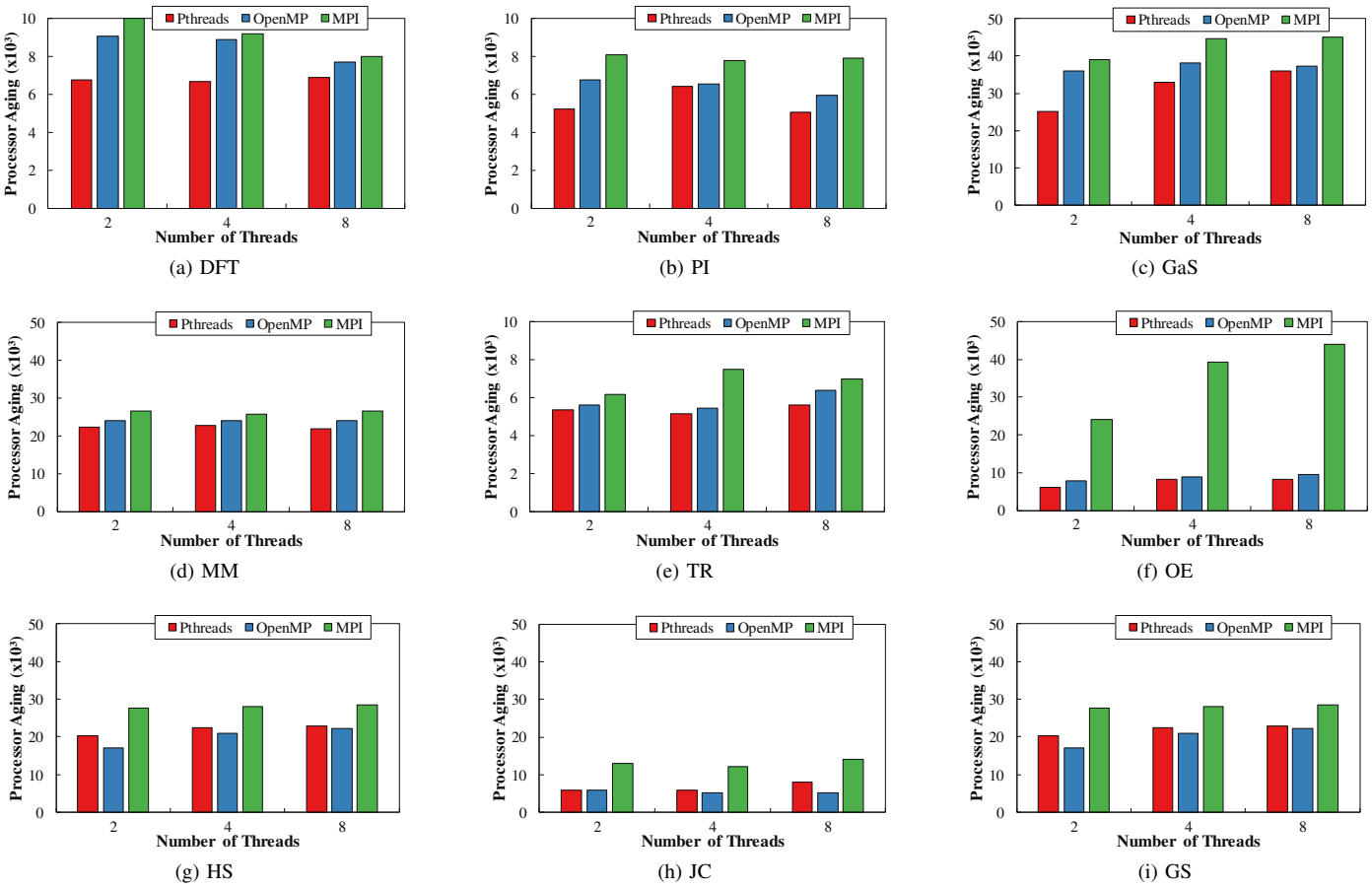

Fig. 1: Aging evaluations performed with three parallel programming interfaces using test sets with 2, 4, and 8 threads. The lower the value, the lower the impact on the lifetime of the processor.

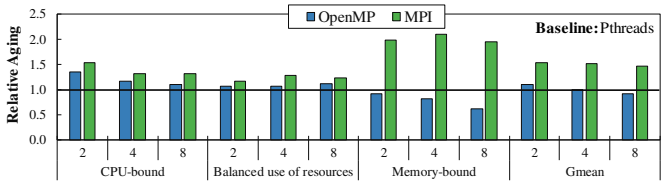

Fig. 2: Processor aging of OpenMP and MPI normalized to Pthreads (black line) for the geometric mean of each benchmark class and the entire benchmark set.

is 26% lower than OpenMP and 35% lower than MPI. As for applications that have balanced use of resources, PThreads and OpenMP presented similar results, with a difference of less than 5% between them. On the other hand, as the application has more accesses to memory due to communication and the mutex is more used to perform synchronization between threads in PThreads, OpenMP presented the best results for the memory-bound applications regardless the number of threads. In the most significant case, it has an impact on the aging process 38% lower than PThreads (8 threads) and 68% lower than MPI (8 threads/processes). When considering the geometric mean of the entire benchmark set, PThreads showed better results for 2 threads while OpenMP presented better results for 8 threads. This behavior happens because the greater the number of threads, the more synchronization is performed, and, as previously discussed, synchronization through mutex limited the performance gains of PThreads.

VI. Conclusion and Future Work

This work evaluated the impact of three widely used parallel programming interfaces (PThreads, OpenMP, and MPI) on the aging process of a real embedded multicore processor. Through an extensive set of executions, we have shown that, depending on the behavior of the application, the use of a specific parallel programming interface may present a higher or lower impact on the processor aging. For example, by using OpenMP for the execution of memory-bound applications, the processor ages 38% and 68% lower than PThreads and MPI, respectively. However, when CPU-bound applications are considered, PThreads has a lower impact on the processor aging than OpenMP (up to 26%) and MPI (up to 35%). As future work, we will consider heterogeneous architectures (e.g., system on chips) and expand our benchmark set to cover a large range of application behaviors and consider other factors, such as instruction-level parallelism and applications that are more control or data flow oriented.

REFERENCES

[1] S. Kamil, J. Shalf, and E. Strohmaier, "Power efficiency in high performance computing," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, April 2008, pp. 1–8.

[2] T. Chantem, X. S. Hu, and R. P. Dick, "Temperature-aware scheduling and assignment for hard real-time applications on mpsocs," *IEEE Trans. on VLSI Systems*, vol. 19, no. 10, pp. 1884–1897, Oct 2011.

[3] M. Shafique, S. Garg, J. Henkel, and D. Marculescu, "The eda challenges in the dark silicon era," in *ACM/EDAC/IEEE DAC*, June 2014, pp. 1–6.

[4] A. F. Lorenzon, C. C. D. Oliveira, J. D. Souza, and A. C. S. B. Filho, "Aurora: Seamless optimization of openmp applications," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1–15, 2018.

[5] B. Chapman, G. Jost, and R. v. d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[6] D. Butenhof, *Programming with POSIX Threads*, ser. Addison-Wesley professional computing series. Addison-Wesley, 1997. [Online]. Available: https://books.google.com.br/books?id=_xvnuFzo7q0C

[7] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI-The Complete Reference, Volume 1: The MPI Core*, 2nd ed. Cambridge, MA, USA: MIT Press, 1998.

[8] M. Namaki-Shoushtari, A. Rahimi, N. Dutt, P. Gupta, and R. K. Gupta, "Argo: Aging-aware gpgpu register file allocation," in *IEEE/ACM/IFIP CODES+ISSS*. NJ, USA: IEEE Press, 2013, pp. 30:1–30:9.

[9] A. Bartolini, M. Cacciari, A. Tilli, and L. Benini, "A distributed and self-calibrating model-predictive controller for energy and thermal management of high-performance multicores," in *DATE*. IEEE, 2011, pp. 1–6.

[10] M. Cacciari, A. Bartolini, A. Tilli, and L. Benini, "Thermal and energy management of high-performance multicores: Distributed and self-calibrating model-predictive controller," *IEEE Transactions on Parallel Distributed Systems*, vol. 24, pp. 170–183, 01 2013.

[11] F. Reghenzani, S. Formentin, G. Massari, and W. Fornaciari, "A constrained extremum-seeking control for cpu thermal management," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, ser. CF '18. New York, NY, USA: ACM, 2018, pp. 320–325.

[12] D. Zoni and W. Fornaciari, "Nbti-aware design of noc buffers," in *IMA-OCMC*. NY, USA: ACM, 2013, pp. 25–28.

[13] A. Marongiu, A. Acquaviva, and L. Benini, "Openmp support for nbti-induced aging tolerance in mpsocs," in *Symposium on Self-Stabilizing Systems*. Springer, 2009, pp. 547–562.

[14] F. Mulas, D. Atienza, A. Acquaviva, S. Carta, L. Benini, and G. De Micheli, "Thermal balancing policy for multiprocessor stream computing platforms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 12, pp. 1870–1882, 2009.

[15] T. Chantem, X. S. Hu, and R. P. Dick, "Temperature-aware scheduling and assignment for hard real-time applications on mpsocs," *IEEE Trans. on VLSI Systems*, vol. 19, no. 10, pp. 1884–1897, 2011.

[16] S. Corbetta and W. Fornaciari, "Nbti mitigation in microprocessor designs," in *GLSVLSI*, ser. GLSVLSI '12. New York, NY, USA: ACM, 2012, pp. 33–38.

[17] D. A. Mallón, G. L. Taboada, C. Teijeiro, J. Touriño, B. B. Fraguela, A. Gómez, R. Doallo, and J. C. Mouriño, "Performance evaluation of mpi, upc and openmp on multicore architectures," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, M. Ropo, J. Westerholm, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 174–184.

[18] J. Balladini, R. Suppi, D. Rexachs, and E. Luque, "Impact of parallel programming models and cpus clock frequency on energy consumption of hpc systems," in *Computer Systems and Applications (AICCSA), 2011 9th IEEE/ACS International Conference on*. IEEE, 2011, pp. 16–21.

[19] S. Hua and Z. Yang, "Comparison and analysis of parallel computing performance using openmp and mpi," *Open Automation and Control Systems Journal*, vol. 5, no. 1, 2013.

[20] K. Fellows, J. Torrellas, and S. Mitra, "A comparative study of the effects of parallelization on arm and intel based platforms," 2014.

[21] A. F. Lorenzon, M. C. Cera, and A. C. S. Beck, "On the influence of static power consumption in multicore embedded systems," in *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2015, pp. 1374–1377.

[22] ——, "Investigating different general-purpose and embedded multicores to achieve optimal trade-offs between performance and energy," *Journal of Parallel and Distributed Computing*, vol. 95, pp. 107 – 123, 2016, special Issue on Energy Efficient Multi-Core and Many-Core Systems, Part I. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731516300090

[23] B. Wang, D. Schmidl, and M. S. Müller, "Evaluating the energy consumption of openmp applications on haswell processors," in *International Workshop on OpenMP*. Springer, 2015, pp. 233–246.

[24] S. Salehian, J. Liu, and Y. Yan, "Comparison of threading programming models," in *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*. IEEE, 2017, pp. 766–774.

[25] J. V. Lima, I. Raïs, L. Lefèvre, and T. Gautier, "Performance and energy analysis of openmp runtime systems with dense linear algebra algorithms," in *2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*. IEEE, 2017, pp. 7–12.

[26] M. White, "Microelectronics reliability: physics-of-failure based modeling and lifetime evaluation," Pasadena, CA: Jet Propulsion Laboratory, National Aeronautics and Space Administration, 2008., Tech. Rep., 2008.

[27] R. Feynman, R. Leighton, and M. Sands, *The Feynman Lectures on Physics*, ser. The Feynman Lectures on Physics. Addison-Wesley, 1963, no. v. 1.

[28] J. H. Flynn and L. A. Wall, "A quick, direct method for the determination of activation energy from thermogravimetric data," *Journal of Polymer Science Part B: Polymer Letters*, vol. 4, no. 5, pp. 323–328, 1966.

[29] A. Chandramowlishwaran, K. Knobe, and R. Vuduc, "Performance evaluation of concurrent collections on high-performance multicore computing systems," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, vol. 00, 04 2010, pp. 1–12. [Online]. Available: doi.ieeecomputersociety.org/10.1109/IPDPS.2010.5470404

[30] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 3, pp. 189–204, Aug. 2000.

[31] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[32] A. F. Lorenzon, M. C. Cera, and A. C. Schneider Beck, "Performance and energy evaluation of different multi-threading interfaces in embedded and general purpose systems," *Journal of Signal Processing Systems*, vol. 80, no. 3, pp. 295–307, Sep 2015. [Online]. Available: https://doi.org/10.1007/s11265-014-0925-9

[33] H. Akkan, M. Lang, and L. Ionkov, "Hpc runtime support for fast and power efficient locking and synchronization," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2013, pp. 1–7.