Deliverable D5.1 – Technical description of the holistic design flow in CONTINUUM

Rabab Bouziane, Erven Rohou, Florent Bruguier, Guillaume Devic, Abdoulaye Gamatié, Guilherme Leobas, Marcelo Novaes, David Novo, Pierre-Yves Péneau, Fernando Magno Quintão Pereira, et al.

CONTINUUM

Project Ref. Number ANR-15-CE25-0007

D5.1 – Technical description of the holistic design flow in CONTINUUM

Version 2.0
(2019)
Final

Public Distribution

Main contributors:
R. Bouziane, E. Rohou (Inria); F. Bruguier, G. Devic, A. Gamatié, G. Leobas, M. Novaes, D. Novo, P.-Y. Péneau, F. Pereira, G. Sassatelli (LIRMM); S. Bernabovi, M. Chapman and P. Naudin (Cortus)

Project Partners: Cortus S.A.S, **Inria**, **LIRMM**

Project Partner Contact Information

Cortus S.A.S Michael Chapman 97 Rue de Freyr Le Génésis 34000 Montpellier France Tel: +33 430 967 000 E-mail: michael.chapman@cortus.com	**Inria** Erven Rohou Inria Rennes - Bretagne Atlantique Campus de Beaulieu 35042 Rennes Cedex France Tel: +33 299 847 493 E-mail: erven.rohou@inria.fr
LIRMM Abdoulaye Gamatié Rue Ada 161 34392 Montpellier France Tel: +33 4 674 19828 E-mail: abdoulaye.gamatie@lirmm.fr	

Table of Contents

Executive Summary

The CONTINUUM project aims to define a new energy-efficient compute node model, which will benefit from a suitable combination of efficient compilation techniques, emerging memory and communication technologies together with heterogeneous cores.

This deliverable provides a technical summary of the different studies that contribute to the holistic design flow of CONTINUUM: energy consumption optimization through innovative compilation approaches and cache memory system design, taking into account the integration of non-volatile memory technologies; a workload management technique enabling performance and energy improvements in heterogeneous multicore systems; and finally, the compute node architecture prototype designed with the Cortus core technology.

Please note that the contents of this deliverable is mainly based on some extracts published in conferences or journals by the consortium members of the CONTINUUM project. More technical details could be found in the corresponding references.

1 Introduction

The holistic design flow targeted by our project is motivated by the following considerations:

- the need for a co-design approach between both compiler and architecture designers, and technology experts. This should bring all the actors to tightly cooperate towards the seamless definition of the target compute node architecture, which able to answer the energy-efficiency challenge;
- beyond the choice of suitable CPU cores, the careful selection of suitable technologies could significantly contribute to reducing the expected system power consumption without compromising the performance. In particular, we explored emerging non-volatile memory technologies as the key solution;
- as a consequence, the need for revisiting existing compilation and runtime management techniques to adapt them to the specific features of the designed compute node, e.g., core and memory heterogeneity, which calls for some adaptive management of workloads and data.

According to the above considerations, we, therefore, study a system design "continuum" that seamlessly goes from software level to memory technology level via hardware architecture. Figure 1 depicts the different aspects involved in the considered design flow.

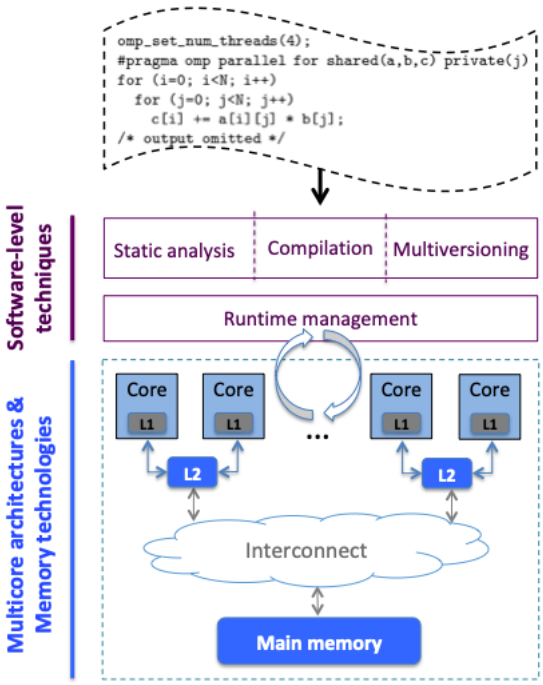

Figure 1: Holistic design flow of CONTINUUM project

The rest of this deliverable briefly describes each of these aspects, in based, in major part, on the dissemination material already published in the context of the CONTINUUM project: starting from

NVM technology integration implications, we respectively discuss the effort achieved by the project to optimize the energy consumption through innovative compilation and workload management techniques (Section 2); then, we briefly describe the final compute node architecture prototype designed with the Cortus core technology (Section 3). Finally, some concluding remarks are given (Section 4).

2 Software level: workload analysis and runtime optimization

We first describe some studies that have been done in the project up to now in order to suitably leverage NVM-based memory hierarchy. Then, we discuss our efforts for contributing to efficient workload management.

2.1 Portable NVM-oriented analysis and optimizations

We explore some compile-time analyses and optimization and software analysis, as a possible alternative to leverage the low leakage current inherent to emerging NVM technologies [12] for energy-efficiency. A major advantage is a flexibility and portability across various hardware architectures enabled by such an approach, compared to the hardware-oriented techniques found in the literature. Our proposal is inspired by some existing techniques such as the silent store elimination technique introduced by Lepak et al. [24] and worst-case execution time analysis techniques [41]. Our main contributions are summarized below.

Silent store elimination: profiling-based versus static analysis approaches. In [4, 5, 6], we proposed a *silent store* elimination technique through an implementation in the LLVM compiler [23]. Thanks to this implementation, a program is optimized once and run on any execution platform while avoiding silent stores. This is not the case of the hardware-level implementation. We evaluated the profitability of this silent store elimination for NVM cache memories. Silent stores have been initially proposed and studied by Lepak et al. [25]. They suggested new techniques for aligning cache coherence protocols and micro-architectural store handling techniques to exploit the value locality of stores. Their studies showed that eliminating silent stores helps to improve monoprocessor speedup and reduce multiprocessor data bus traffic.

Methods devoted to removing silent stores are meant to improve the system performance by reducing the number of write-backs. Bell et al. [3] affirmed that frequently occurring stores are highly likely to be silent. They introduced the notion of critical silent stores and show that all of the avoidable cache write-back can be removed by removing a subset of silent stores that are critical.

In our proposal, the silent store elimination technique is leveraged at compiler-level. This favors portability and requires no change to the hardware. We remind that this technique is not dedicated only to STT-RAM but to all NVMs. Here, STT-RAM is considered due to its advanced maturity and performance compared to other NVM technologies. Our approach concretely consists in modifying the code by inserting *silentness* verification before stores that are identified as likely silent. As illustrated in Figure 2, the verification includes the following instructions:

1. a load instruction at the address of the store;
2. a comparison instruction, to compare the to-be-written value with the already stored value;
3. a conditional branch instruction to skip the store if needed.

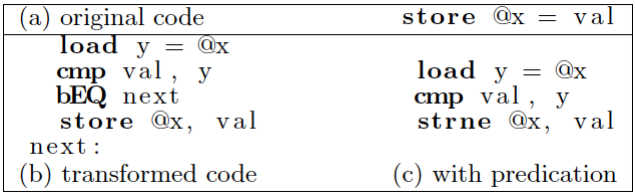

Figure 2: Silent store elimination: (a) original code stores `val` at address of `x`; (b) transformed code first loads the value at address of `x` and compares it with the value to be written, if equal, the branch instruction skips the store execution; (c) when the instruction set supports predication, the branch can be avoided and the store made conditional.

We showed that energy gains highly depend on the silentness percentage in programs, and on the energy consumption ratio of read/write operations costs for NVMs. We validated our proposal with the Rodinia benchmark suite while reporting up to 42% gain in energy for some applications. This validation relies on an analytic evaluation considering typical NVM parameters extracted from the literature.

The above silent store analysis was carried out based on program profiling, a complementary study leveraging static code analysis is proposed [36]. It aims at predicting the presence of silent stores prediction by analyzing the syntactic properties of programs, through a set of 127 features organized into eight categories. To validate the approach, we collected a number of benchmarks from well-known suites, such as SPEC CPU2006, MiBench, mediabench, BitBench, CoyoteBench, Trimaran, etc. In total, we reused 34 different collections, which gave us 222 programs to analyze.

Out of the 127 features, 100 have been observed in our benchmarks. Figure 3 shows them, together with the probability of silentness associated with their occurrence. Features in each category refer to different parts of the store instruction "ℓ : `p[i]` = `v`", and are described as follows:

- VALUETYPE. The type of variable v: `Vfp`: 32-bit floating point; `Vdb`: 64-bit floating point; `Vin`: integers (with different bitwidths); `Vpt`: pointer.
- VALUESIZE. The size of the type, in bits: `sz0`: size is unknown v; `sz1`: one bit; `sz8`: two to eight bits; `s16`: nine to 16 bits; `s32`: 16 to 32 bits; `szN`: more than 32 bits.
- VALUEDEPS. The operands (instructions, addresses and constants) that contribute to forming the value of v.
- VALUEORIGIN The last instruction or operation that produced the value v.
- POINTERTYPE. The type of p: `Pfp`: 32-bit floating point; `Pdb`: 64-bit floating point; `Pin`: integer; `Pst`: C-like struct; `Pay`: array; `Ppt`: another pointer; `Pvc`: SIMD vector.
- POINTERLOC. The location of the region pointed by p: `Msc`: static memory; `Msk`: stack; `Mhp`: heap; `Mar`: unknown location loaded from function argument; `Mfn`: unknown location returned by function; `Mph`: unknown location loaded from an SSA ϕ-function; `Mbt`: unknown location produced by a bitwise operation; `Mld`: other unknown locations;

Total	Feat	P1.0	P0.8	P0.6	P0.4	P0.2
1078	Vfp	0.13	0.24	0.28	0.32	0.38
3116	Vdb	0.08	0.14	0.15	0.19	0.22
27266	Vin	0.12	0.17	0.2	0.24	0.29
7914	Vpt	0.1	0.12	0.14	0.17	0.19
7915	sz0	0.1	0.12	0.14	0.17	0.19
142	sz1	0.33	0.47	0.54	0.61	0.65
1118	sz8	0.16	0.23	0.27	0.32	0.37
858	s16	0.1	0.21	0.23	0.27	0.34
24195	s32	0.12	0.17	0.2	0.24	0.29
5146	szN	0.09	0.15	0.17	0.2	0.23
614	Pfp	0.1	0.23	0.26	0.3	0.37
2087	Pdb	0.07	0.14	0.16	0.19	0.22
18523	Pin	0.09	0.14	0.17	0.21	0.25
10797	Pst	0.15	0.2	0.23	0.28	0.32
3532	Pay	0.15	0.22	0.24	0.28	0.33
3820	Ppt	0.08	0.1	0.1	0.12	0.14
1	Pvc	0	1	1	1	1
6281	Ozr	0.4	0.48	0.52	0.58	0.62
3857	Oin	0.05	0.1	0.12	0.15	0.18
466	Ofp	0.02	0.03	0.03	0.07	0.08
454	Ogb	0.09	0.15	0.18	0.22	0.27
427	Oay	0.16	0.2	0.21	0.22	0.24
5043	Oag	0.05	0.09	0.11	0.15	0.19
5322	Old	0.09	0.14	0.18	0.23	0.29
2578	Ocl	0.11	0.15	0.18	0.24	0.28
937	Omx	0.07	0.15	0.21	0.26	0.32
4075	Oad	0.05	0.08	0.11	0.13	0.16
3082	Oic	0	0.01	0.01	0.01	0.02
140	Ong	0.04	0.16	0.2	0.22	0.23
9	Ont	0	0	0.11	0.11	0.22
768	Omd	0.07	0.17	0.18	0.22	0.25
938	Omu	0.04	0.11	0.14	0.17	0.22
989	Obn	0.04	0.12	0.16	0.2	0.23
2728	Ocs	0.05	0.09	0.12	0.17	0.22
12	Oal	0	0	0	0.08	0.08
238	Oun	0.04	0.1	0.13	0.15	0.17
1030	Oiy	0.01	0.03	0.03	0.05	0.06
5306	Msc	0.19	0.23	0.25	0.26	0.28
18978	Msk	0.08	0.12	0.15	0.19	0.22
761	Mhp	0.16	0.22	0.24	0.28	0.3
6127	Mar	0.11	0.16	0.19	0.24	0.28
6075	Mld	0.14	0.22	0.26	0.31	0.36
1190	Mfn	0.15	0.2	0.23	0.25	0.28
930	Mph	0.06	0.12	0.15	0.19	0.25
7	Mbt	0	0	0.43	0.43	0.43
18800	Dzr	0.18	0.24	0.28	0.32	0.37
24733	Din	0.05	0.1	0.13	0.16	0.2
1353	Dfp	0.03	0.08	0.1	0.15	0.17
5181	Dgb	0.08	0.14	0.16	0.19	0.23
5008	Dfn	0.09	0.14	0.17	0.21	0.25

Total	Feat	1	0.8	0.6	0.4	0.2
2844	Dc?	0.07	0.1	0.11	0.13	0.15
11755	Dar	0.07	0.12	0.14	0.17	0.21
20216	Dld	0.06	0.11	0.13	0.17	0.21
5385	Dcl	0.09	0.13	0.16	0.2	0.24
590	Dcp	0.11	0.19	0.24	0.35	0.39
5185	Dph	0.07	0.15	0.19	0.23	0.28
392	Dsl	0.05	0.09	0.13	0.19	0.24
10883	Dad	0.04	0.09	0.11	0.14	0.17
3336	Dsb	0.05	0.11	0.13	0.17	0.22
3397	Dml	0.05	0.11	0.13	0.16	0.2
1249	Ddv	0.02	0.09	0.13	0.17	0.22
341	Drm	0.02	0.05	0.08	0.12	0.21
1660	Dsh	0.02	0.09	0.12	0.15	0.2
1397	Dbt	0.05	0.12	0.16	0.19	0.23
13275	Dgp	0.06	0.12	0.15	0.18	0.23
2233	Dcs	0.04	0.08	0.12	0.14	0.16
181	Dex	0.06	0.09	0.14	0.18	0.22
207	Dfi	0.03	0.08	0.11	0.14	0.14
648	Dif	0.03	0.07	0.11	0.15	0.19
15	Dip	0.13	0.13	0.33	0.33	0.33
86	Dpi	0.07	0.12	0.15	0.16	0.2
1955	Dtr	0.04	0.09	0.12	0.18	0.22
6876	Dsx	0.05	0.12	0.15	0.2	0.25
1753	Dzx	0.09	0.16	0.19	0.24	0.29
12290	Dal	0.05	0.09	0.12	0.16	0.21
289	Di?	0.03	0.1	0.13	0.15	0.17
1166	Smn	0.21	0.22	0.23	0.23	0.24
25142	Sl0	0.13	0.17	0.19	0.23	0.26
8957	Sl1	0.08	0.14	0.18	0.22	0.27
3543	Sl2	0.08	0.17	0.21	0.24	0.29
1274	Sl3	0.07	0.17	0.22	0.26	0.29
458	Sln	0.06	0.11	0.15	0.21	0.27
16197	Scm	0.13	0.17	0.19	0.23	0.26
5830	Scl	0.1	0.15	0.19	0.24	0.29
643	Spl	0.1	0.16	0.19	0.3	0.33
11032	Slx	0.08	0.15	0.19	0.24	0.29
6881	Smp	0.14	0.2	0.23	0.28	0.32
10631	Sms	0.1	0.15	0.18	0.23	0.27
13771	Ssl	0.08	0.15	0.19	0.23	0.28
8741	Sdl	0.07	0.13	0.16	0.21	0.25
3151	Svi	0.19	0.29	0.34	0.4	0.46
6174	Spi	0.05	0.1	0.14	0.18	0.22
2364	Erc	0.09	0.14	0.19	0.24	0.29
2364	Eaf	0.09	0.14	0.19	0.24	0.29
100	Es1	0.09	0.12	0.16	0.26	0.41
565	Es4	0.05	0.14	0.21	0.29	0.35
631	Es8	0.21	0.24	0.28	0.32	0.35
952	EsN	0.04	0.1	0.13	0.17	0.2
1122	Esp	0.05	0.12	0.18	0.25	0.31
3181	Etp	0.08	0.19	0.24	0.29	0.33

Figure 3: Probabilities of silentness distributed per features [36]. **Total** shows number of instructions that present each feature. The other columns show thresholds of silentness. Darker cells indicate a higher proportion of silent stores. For instance, 48% of stores with the feature `Ozr` were silent 80% or more of the time.

- LABELLOC. the location ℓ of the instruction within the program. `Smn`: within the main function; `Sl0`: not inside a loop. `Sl1`-`Sl3`: within a singly, doubly or triply nested loop; `Slx`:

within more than three loop nests. `Scm`: ℓ post-dominates the entry point of the function. Thus, if this function is invoked, the store happens compulsorily. `Scl`: ℓ is compulsory within the loop where it is located. `Spl`: ℓ exists within a compulsory loop, i.e., the loop's entry point post-dominates the function's entry point. `Slx`: ℓ is within a single-exit loop, i.e., after the loop runs, the program always reaches the same point. `Ssl`: ℓ is within a single-latch loop. A latch is a block *inside* the loop that leaves it. `Sdl`: ℓ dominates the single latch in the loop. `Smp`: in a basic block with multiple predecessors. `Sms`: in a basic block with multiple successors. `Svi`: reached only by invariant definitions of v. `Spi`: reached only by invariant definitions of p.

- POINTERSTRIDE. The offset i that is added to p when building the store address, e.g., `p + i`. We use LLVM's scalar evolution to recover this information. `Erc`: i is created by some recursive expression, e.g.: `i = i + 1`; `Eaf`: i is created by some affine expression, e.g., `i = 2 * b + c`; `Es1`: i has a stride of size 1; `Es4`: i has a stride of size 4; `Es8`: stride of size 8; `EsN`: regular stride, but its value is unknown; `Esp`: stride has the same size as the region pointed by p; `Etp`: i exists in a loop of known trip-count.

Partial WCET analysis for efficient NVM mapping. On the other hand, we proposed another type of program analysis, which aims at exploiting the variable NVM data retention capacities. This opens the opportunity to explore different energy/performance trade-offs, depending on the energy-efficiency requirements of a system. We introduced a variant of worst-case execution time analysis [40], to determine the partial worst-case lifetimes of the variables of a program, referred to as δ-WCET [7]. Based on this knowledge, the variables can be safely allocated to NVM memory banks accommodating their requirements in terms of data retention duration.

Our methodology is a two-step process, illustrated in Figure 4. First, we identify the *def-use* chains in the program. In other words, for each store instruction, we determine all the loads that can read the value previously written. Second, we compute the worst-case execution time between the store and all subsequent loads (step referred to as "δ-WCET"). For this purpose, we have developed a method to compute partial WCET estimates (see details in our previous work [8]).

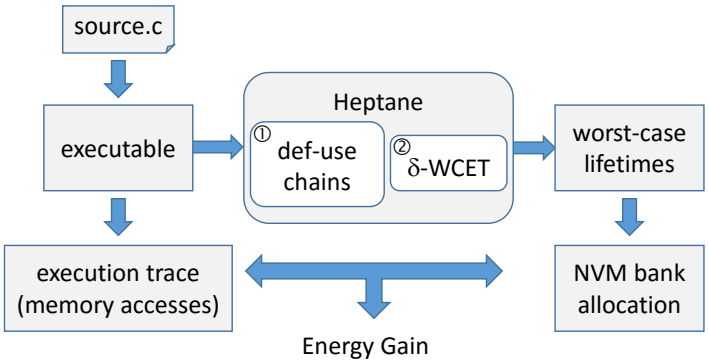

Figure 4: Sketch of our framework (input: C code).

We computed the dynamic energy gain, as illustrated in Figure 5 for the Malardalen benchmark-suite, w.r.t. 4 MB and 32 KB STT-RAM memory setups respectively. Here, the reported gain is computed against a baseline setup consisting of an STT-RAM memory with a retention time of 4.27 years.

The results show that we achieved with the small memory configuration up to 80 % of energy gain compared to the baseline (small memory with 4.27 years retention time) [7].

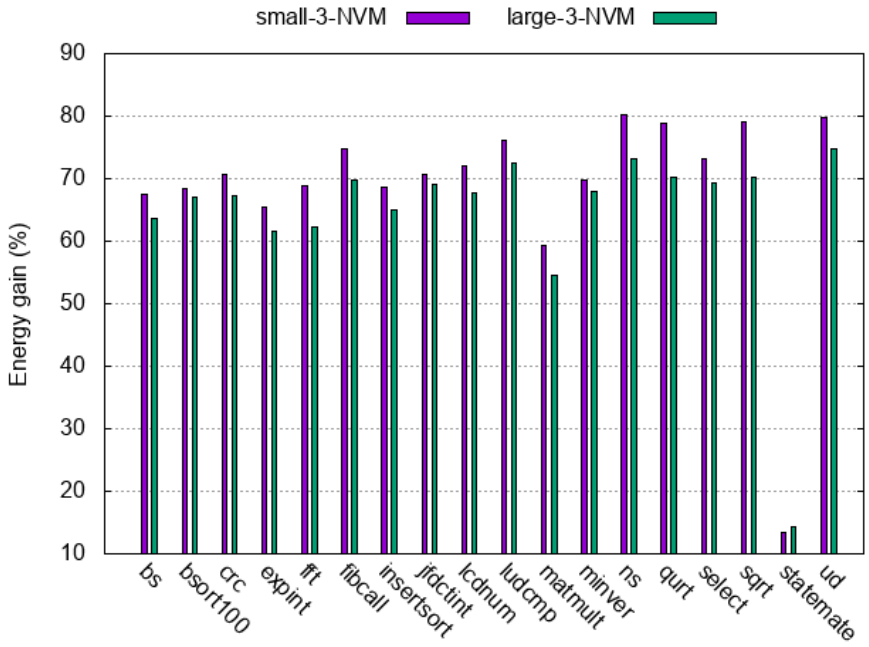

Figure 5: Energy savings evaluation

Exploring advanced cache replacement policies for last-level cache in NVM. We evaluated the impact of cache replacement policies in the reduction of cache memory write [35] in presence of STT-RAM. We observed that *state-of-the-art* Hawkeye cache replacement policy can be beneficial for larger last level caches [20, 34]. Our evaluations showed that the global system performance can be improved up to 14% for a multicore platform. This gain, combined with the drastic static energy reduction enabled by STT-RAM in place of usual SRAM, enhances the energy-efficiency by up to 27.7%.

For example, Figure 6 illustrates the effect of the Hawkeye policy over LRU, based on the study presented in [35]. In the configuration names, the prefixes M ("Medium"), B ("Big") and H ("Huge") respectively correspond to 2MB, 4MB and 8MB last level cache sizes. The considered architecture consists of an Intel Core i7 system, with a 3-level on chip cache hierarchy plus a main memory. We used the SPEC CPU2006 benchmark set. The reported results are normalized for each memory technology configuration combined with the Hawkeye policy, compared to its LRU counterpart. Typically, *H_stt_hawk* is normalized to *H_stt_lru*. For this experiment, we run the SRAM configuration that do not fit into area constraint to illustrate the effect of Hawkeye on SRAM and STT-MRAM for the same cache size. Both SRAM and STT-MRAM configurations follow the same trend regarding the MPKI [1] (Miss Per Kilo Instructions) reduction over LRU since the Hawkeye policy is not impacted by

[1]MPKI is a common metric used to assess LLC performance. It is defined as the total number of cache misses divided by the total number of executed instructions. One possibility to reduce the MPKI is to increase the cache size. The cache

cache latency. Moreover, we use a single core platform where parallel events cannot occur. Hence, eviction decision remains identical for a given cache size, regardless of the cache size. However, the average gain obtained with Hawkeye is slightly better with STT-MRAM. The performance gap between SRAM and STT-MRAM is 3.3% and 3.1%, respectively with LRU and Hawkeye. Hence, reducing the amount of *write-fill* has higher impact on STT-MRAM where writes are penalizing.

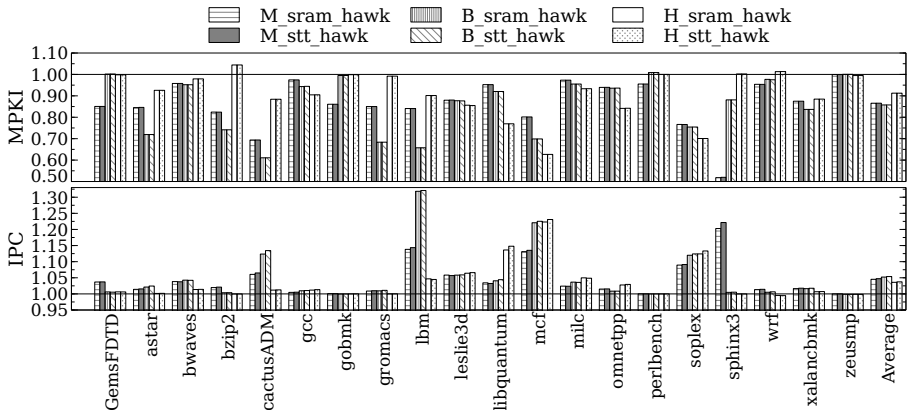

Figure 6: Performance impact of Hawkeye normalized to LRU

Figure 6 shows that the 8MB configuration is not as efficient as the 4MB configuration in terms of performance improvement. The average gain for the Instruction Per Cycle (IPC) for *H_sram_hawk* and *H_stt_hawk* is lower than *B_sram_hawk* and *B_stt_hawk*. This suggests an issue that can be due to either a larger LLC, or the Hawkeye policy, or both. On the other hand, we observe that Hawkeye increases the MPKI compared to LRU for a 8MB LLC. This is explained by some wrong eviction decisions made by Hawkeye. Indeed, the Hawkeye predictor exploits all cache accesses to identify the instructions that generate cache misses. Since a large cache size reduces the number of cache misses, it becomes hard for the predictor to learn accurately from a small set of miss events. Note that the performance for *H_stt_hawk* is still better than other configurations despite these inaccurate decisions.

It is important to notice that the above evaluation was conducted with the ChampSim simulator [1] used for the Cache Replacement Championship at ISCA'17 conference. Despite the notable efforts made recently for accelerating and making systematic gem5-based simulation [10, 29, 30, 38, 13, 14], we decided to use ChampSim which is more abstract and faster yet relevant-enough for our experiments.

2.2 Workload management

We addressed the workload management issue on multicore heterogeneous architectures, by investigating a few techniques. State-of-the-art approaches solve this problem dynamically, e.g., at runtime / operating system levels, or via a middleware [26, 28], or statically, e.g., via compilers [27, 33].

Compiler-assisted Adaptive Program Scheduling. In a proposed approach [31], we used the compiler to partition source code into program phases: regions whose syntactic characteristics lead to

contains more data and reduces the probability for a cache miss to occur. This results in penalties in terms of cache latency, energy and area.

similar runtime behavior. Reinforcement learning is used to map hardware configurations that best fit program regions. To validate our proposal, we implemented the *Astro* framework to instrument and execute applications in heterogeneous architectures. The framework collects syntactic characteristics from programs and instruments them using LLVM [23]. For a program region, the scheduler can take into account its corresponding characteristics collected statically, for immediate action. An action consists in choosing an execution configuration and collecting the "reward" related to that choice. Such feedback is then used to fine-tune and improve the subsequent scheduling decisions. We rely on reinforcement learning technique to explore a vast universe of configurations (or states), formed by different hardware setups and runtime data changing over time. However, the universe of runtime states is unbounded, and program behavior is hard to predict without looking into its source code. To speed up convergence, we resort to a compiler, which gives us two benefits: i) mining program features to train the learning algorithm, and ii) instrumenting programs to provide feedback to the scheduler on the code region currently under execution.

The heart of the Astro system is the Actuation Algorithm outlined in Figure 7. Actuation consists of *phase monitoring*, *learning* and *adapting*. These three steps happens at regular intervals, called *check points*, which, in Figure 7, we denote by i and i+1. The rest of this section describes these events.

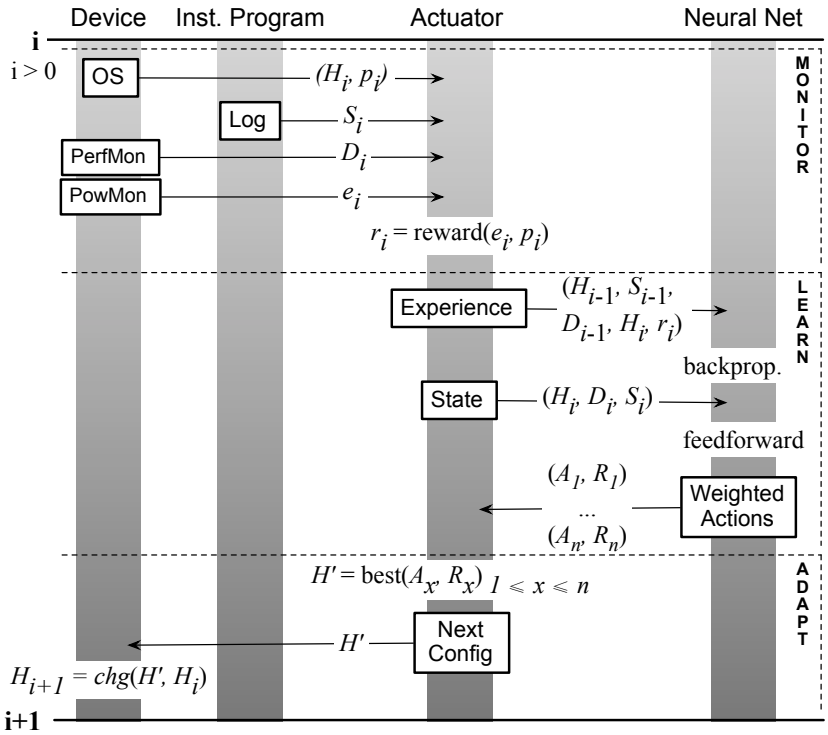

Figure 7: The actuation algorithm [31].

Monitoring step.

To collect information that will be later used, a monitor in Astro reads four kinds of data. Figure 7 highlights this data:

- From the Operating System (OS): current hardware configuration H and instructions p executed since last check point.
- From the Program (Log): the current program phase S.
- From the device's performance counters (PerfMon): the current hardware phase D.
- From the power monitor (PowMon [39]): the energy e consumed since the last checkpoint.

The monitor collects this data at periodic intervals, whose granularity is configurable. It was 500 milliseconds in our setup. The recording of the program phase is aperiodic, following from instrumentation inserted in the program by the compiler. Information is logged at the entry point of functions, and around library calls that might cause the program to enter a dormant state. The hardware configuration is updated whenever it changes. The metrics e and p let us define the notion of *reward* as follows:

Definition 1 (Reward)**.** *The reward is the set of observable events that determine how well the learning algorithm is adapting to the environment. The reward is computed from a pair* (e, p)*, formed by the* Energy Consumption Level e*, measured in Joules per second (Watt), and the* CPU Performance Level p*, measured in the number of instructions executed per second.*

The metric used in the reward is given by a weighted form of performance per watt, namely $MIPS^{\gamma}/Watt$, where γ is a design parameter that gives a boosting performance effect in the system. This is usually a trade-off between performance and energy consumption. To optimize for energy, we let $\gamma = 1.0$. A value of $\gamma = 2.0$ emphasizes performance gains: the reward function optimizes (in fact, maximizes the inverse of) the energy delay product per instruction, given by $Watt/IPS^2$; letting $IPS = I/S$ we have $(Watt \times S \times S)/I^2 = (Energy \times Delay)/I^2$. This aims to minimize both the energy and the amount of time required to execute thread instructions [9].

Learning step.

The learning phase uses the Q-learning algorithm. As illustrated in Figure 7, a key component in this process is a multi-layer Neural Network (NN) that receives inputs collected by the Monitor. The NN outputs the actions and their respective rewards to the Actuator so that a new system adaptation can be carried out. Following the common methodology, learning happens in two phases: *back-propagation* and *feed-forwarding*. During back-propagation, we update the NN using the experience data given by the Actuator (Figure 7). Experience data is a triple: the current state, the action performed and the reward thus obtained. The state consists of a hardware configuration (H_{i-1}), static features (S_{i-1}) and dynamic features (D_{i-1}) at check points i-1. The action performed at check point i-1 makes the system move from hardware configuration H_{i-1} to H_i. The reward is given by r_i, received after the action is taken. The NN consists of a number of layers including computational nodes, i.e., neurons. The input layer uses one neuron to characterize each triple $(state, action, reward)$. The output layer has one neuron per action/configuration available in the system. During the feed-forward phase, we perform predictions using the trained NN. Each node of the NN is responsible to accumulate the product of its associated weights and inputs. Given as input a state (H_i, D_i, S_i) at checkpoint i, the result of the feed-forward step is an array of pairs $A \times R$, where A is an *action*, and R is its *reward*,

estimated by NN. Actions determine configuration changes; rewards determine the expected performance gain, in terms of energy and time, that we expect to obtain with the change. We use the method of gradient descent to minimize a loss function given by the difference between the reward predicted by the NN, and the actual value found via hardware performance counters.

Adaptation step. At this phase, Astro takes an *action*. Together with states and rewards, actions are one of the three core notions in Q-learning, which we define below:

Definition 2 (Action)**.** *Action is the act of choosing the next hardware configuration H to be adopted at a given checkpoint.*

An action may change the current hardware configuration; hence, adapting the program according to the knowledge inferred by the Neural Network. Following Figure 7, we start this step by choosing, among the pairs $\{(A_1, R_1), \ldots, (A_n, R_n)\}$, the action A_x associated with the maximal reward R_x. A_x determines, uniquely, a hardware configuration H'. Once H' is chosen, we proceed to adopt it. However, the adoption of a configuration is contingent on said configuration being available. Cores might not be available because they are running higher privilege jobs, for instance. If the Next Configuration is accessible, Astro enables it; otherwise, the whole system remains in the configuration H_i active at check point i. Such choice is represented, in Figure 7, by the function $H_{i+1} = chg(H', H_i)$. Regardless of this outcome, we move on to the next check point, and to a new actuation round.

Machine learning-based performance model building. In [16], we kept investigating the definition of performance and energy consumption prediction models. Following the promising insights obtained from ar earlier work [15], we generalized our study to more machine learning techniques. We mainly focused on supervised machine learning techniques: Support Vector Machines (SVM) [18], Adaptive Boosting (AdaBoost) [37] and Artificial Neural Networks (ANNs) [19]. We considered a dataset consisting of mapping scenarios evaluated with the McSim simulator [22, 21]. Our evaluation showed that providing some limited training efforts, AdaBoost and ANNs can provide very good predictions, estimated around 84.8% and 89.05% correct prediction scores. This opens an interesting perspective for considering such models in the runtime mapping and scheduling decisions for heterogeneous architectures.

3 Hardware level: a multicore asymmetric architecture

We devised a novel original asymmetric multicore architecture comprising two execution islands: parallel and sequential [17]. While the former is devoted to highly parallelizable workloads for high throughput, the latter addresses weakly parallelizable workloads. Accordingly, the parallel island is composed of many low power cores and the sequential island is composed of a small number of high-performance cores. Our proposal integrates the cost-effective and inherently low power core technology provided by Cortus [2], one of the world-leading semiconductor IP companies in the embedded domain. These cores are highly energy (MIPS/μW) and silicon efficient (MIPS/mm^2) compared to existing technologies. We believe the massive usage of such embedded cores deserves attention to achieve the energy-efficient architectures required for high-performance embedded computing.

Another trade-off considered in our solution is the support of floating point arithmetic, which benefits certain operations in embedded applications, e.g., matrix inversion required for Multiple Input / Multiple Output (MIMO), Fast Fourier Transforms (FFTs) which often suffer from scaling problems in fixed point. As floating point units (FPUs) can be expensive in terms of area and power in the very low power cores being considered, it will be supported only by a subset of these cores.

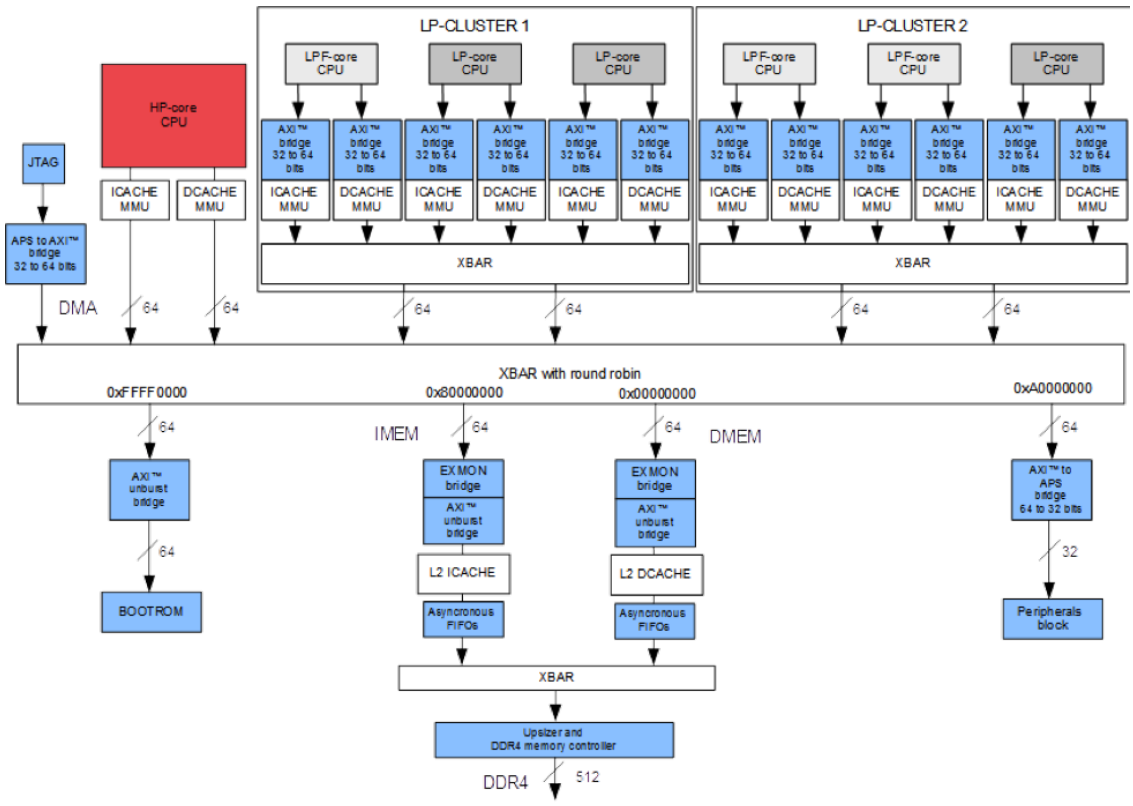

Figure 8: Sketch of the implemented heptacore system.

A prototype comprising 7 CPU cores has been devised (see Figure 8). One high performance CPU Core is implemented with Cortus APSX2 core IP. It is combined with two clusters that mix smaller CPU cores integrating hardware floating point support or not. Each core has a local cache. The relevance of such a heterogeneous multicore architecture has been recently illustrated [32, 11]. A

level 2 memory block caches all the memory access to the external DDR memory, reducing the memory access latency. As a proof-of-concept, a multi-thread execution model is defined for program execution. The workload management exploits the nature of programs, which is analyzable statically during compilation, e.g., computation versus memory intensiveness, floating-point intensive or not.

4 Conclusions

This deliverable presented a brief description of the techniques defined in the framework of the holistic design flow proposed by the CONTINUUM project. This covers: energy consumption optimization through innovative compilation approaches and cache memory system design, taking into account the integration of non-volatile memory technologies; a workload management technique enabling performance and energy improvements in heterogeneous multicore systems; and finally, the compute node architecture prototype designed with the Cortus core technology.

References

[1] The ChampSim simulator. https://github.com/ChampSim/ChampSim.

[2] Cortus SAS – Advanced Processing Solutions. http://www.cortus.com, July 2017.

[3] Bell, G. B., Lepak, K. M., and Lipasti, M. H. Characterization of silent stores. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2000.

[4] Bouziane, R., Rohou, E., and Gamatié, A. How could compile-time program analysis help leveraging emerging nvm features? In *2017 First International Conference on Embedded Distributed Systems (EDiS)*, pages 1–6, 2017. doi: 10.1109/EDIS.2017.8284031.

[5] Bouziane, R., Rohou, E., and Gamatié, A. LLVM-based silent stores optimization to reduce energy consumption on STT-RAM cache memory. In *European LLVM Developers Meeting (EuroLLVM'17), Saarbrucken, Germany*, 2017.

[6] Bouziane, R., Rohou, E., and Gamatié, A. Compile-time silent-store elimination for energy efficiency: an analytic evaluation for non-volatile cache memory. In Chillet, D., editor, *Proceedings of the RAPIDO 2018 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, Manchester, UK, January 22-24, 2018.*, pages 5:1–5:8. ACM, 2018. ISBN 978-1-4503-6417-1. doi: 10.1145/3180665.3180666. URL https://doi.org/10.1145/3180665.3180666.

[7] Bouziane, R., Rohou, E., and Gamatié, A. Energy-efficient memory mappings based on partial WCET analysis and multi-retention time STT-RAM. In Ouhammou, Y., Ridouard, F., Grolleau, E., Jan, M., and Behnam, M., editors, *Proceedings of the 26th International Conference on Real-Time Networks and Systems, RTNS 2018, Chasseneuil-du-Poitou, France, October 10-12, 2018*, pages 148–158. ACM, 2018. doi: 10.1145/3273905.3273908. URL https://doi.org/10.1145/3273905.3273908.

[8] Bouziane, R., Rohou, E., and Gamatié, A. Partial Worst-Case Execution Time Analysis. In *ComPAS 2018 - Conférence d'informatique en Parallélisme, Architecture et Système*, pages 1–8, Toulouse, France, July 2018. URL https://hal.inria.fr/hal-01803006.

[9] Brooks, D. M., Bose, P., Schuster, S. E., Jacobson, H., Kudva, P. N., Buyuktosunoglu, A., Wellman, J.-D., Zyuban, V., Gupta, M., and Cook, P. W. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20:26–44, November 2000. ISSN 0272-1732.

[10] Butko, A., Garibotti, R., Ost, L., Lapotre, V., Gamatié, A., Sassatelli, G., and Adeniyi-Jones, C. A trace-driven approach for fast and accurate simulation of manycore architectures. In *The 20th Asia and South Pacific Design Automation Conference, ASP-DAC 2015, Chiba, Japan, January 19-22, 2015*, pages 707–712. IEEE, 2015. doi: 10.1109/ASPDAC.2015.7059093. URL https://doi.org/10.1109/ASPDAC.2015.7059093.

[11] Butko, A., Bruguier, F., Novo, D., Gamatié, A., and Sassatelli, G. Exploration of performance and energy trade-offs for heterogeneous multicore architectures. *CoRR*, abs/1902.02343, 2019. URL http://arxiv.org/abs/1902.02343.

[12] Chen, A. A review of emerging non-volatile memory (nvm) technologies and applications. *Solid-State Electronics*, 125:25 – 38, 2016. ISSN 0038-1101. doi: https://doi.org/10.1016/j.sse.2016.07.006. URL http://www.sciencedirect.com/science/article/pii/S0038110116300867. Extended papers selected from ESSDERC 2015.

[13] Delobelle, T., Péneau, P., Gamatié, A., Bruguier, F., Senni, S., Sassatelli, G., and Torres, L. MAGPIE: System-level Evaluation of Manycore Systems with Emerging Memory Technologies. In *Workshop on Emerging Memory Solutions - Technology, Manufacturing, Architectures, Design and Test at Design Automation and Test in Europe - DATE'2017, Lausanne, Switzerland*, 2017.

[14] Delobelle, T., Péneau, P.-Y., Senni, S., Bruguier, F., Gamatié, A., Sassatelli, G., and Torres, L. Flot automatique d'évaluation pour l'exploration d'architectures à base de mémoires non volatiles. In *Conférence d'informatique en Parallélisme, Architecture et Système, Compas'16, Lorient, France*, 2016.

[15] Gamatié, A., Ursu, R., Selva, M., and Sassatelli, G. Performance prediction of application mapping in manycore systems with artificial neural networks. In *10th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip, MCSOC 2016, Lyon, France, September 21-23, 2016*, pages 185–192. IEEE Computer Society, 2016. doi: 10.1109/MCSoC.2016.17. URL https://doi.org/10.1109/MCSoC.2016.17.

[16] Gamatié, A., An, X., Zhang, Y., Kang, A., and Sassatelli, G. Empirical model-based performance prediction for application mapping on multicore architectures. *J. Syst. Archit.*, 98:1–16, 2019. doi: 10.1016/j.sysarc.2019.06.001. URL https://doi.org/10.1016/j.sysarc.2019.06.001.

[17] Gamatié, A., Devic, G., Sassatelli, G., Bernabovi, S., Naudin, P., and Chapman, M. Towards energy-efficient heterogeneous multicore architectures for edge computing. *IEEE Access*, 7:49474–49491, 2019. doi: 10.1109/ACCESS.2019.2910932. URL https://doi.org/10.1109/ACCESS.2019.2910932.

[18] Hearst, M. A., Dumais, S. T., Osuna, E., Platt, J., and Scholkopf, B. Support vector machines. *IEEE Intelligent Systems and their Applications*, 13(4):18–28, 1998. doi: 10.1109/5254.708428.

[19] Hopfield, J. J. Artificial neural networks. *IEEE Circuits and Devices Magazine*, 4(5):3–10, 1988. doi: 10.1109/101.8118.

[20] Jain, A. and Lin, C. Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 78–89. IEEE, 2016.

[21] Latif, K., Effiong, C. E., Gamatié, A., Sassatelli, G., Zordan, L. B., Ost, L., Dziurzanski, P., and Soares Indrusiak, L. An Integrated Framework for Model-Based Design and Analysis of Automotive Multi-Core Systems. In *FDL: Forum on specification & Design Languages*, Work-in-Progress Session, Barcelona, Spain, September 2015. URL https://hal-lirmm.ccsd.cnrs.fr/lirmm-01418748.

[22] Latif, K., Selva, M., Effiong, C., Ursu, R., Gamatié, A., Sassatelli, G., Zordan, L., Ost, L., Dziurzanski, P., and Indrusiak, L. S. Design space exploration for complex automotive applications: an engine control system case study. In *Proceedings of the 2016 Workshop on Rapid Simulation and Performance Evaluation - Methods and Tools, RAPIDO@HiPEAC 2016, Prague, Czech Republic, January 18, 2016*, pages 2:1–2:7. ACM, 2016. doi: 10.1145/2852339.2852341. URL https://doi.org/10.1145/2852339.2852341.

[23] Lattner, C. and Adve, V. S. LLVM: A compilation framework for lifelong program analysis & transf. In *CGO'04*, pages 75–88, 2004.

[24] Lepak, K. M. and Lipasti, M. H. Silent stores for free. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, pages 22–31, New York, NY, USA, 2000. ACM. ISBN 1-58113-196-8. doi: 10.1145/360128.360133. URL http://doi.acm.org/10.1145/360128.360133.

[25] Lepak, K. M., Bell, G. B., and Lipasti, M. H. Silent stores and store value locality. *Transactions on Computers*, 50(11), 2001.

[26] Margiolas, C. and O'Boyle, M. F. P. Portable and transparent software managed scheduling on accelerators for fair resource sharing. In *CGO*, pages 82–93. ACM, 2016.

[27] Mendonça, G., Guimarães, B., Alves, P., Pereira, M., Araújo, G., and Pereira, F. M. Q. a. DawnCC: Automatic annotation for data parallelism and offloading. *TACO*, 14(2):13:1–13:25, 2017.

[28] Nishtala, R., Carpenter, P. M., Petrucci, V., and Martorell, X. Hipster: Hybrid task manager for latency-critical cloud workloads. In *HPCA*, pages 409–420. IEEE, 2017.

[29] Nocua, A., Bruguier, F., Sassatelli, G., and Gamatié, A. Elasticsimmate: A fast and accurate gem5 trace-driven simulator for multicore systems. In *12th International Symposium on Reconfigurable Communication-centric Systems-on-Chip, ReCoSoC 2017, Madrid, Spain, July 12-14, 2017*, pages 1–8. IEEE, 2017. doi: 10.1109/ReCoSoC.2017.8016146. URL https://doi.org/10.1109/ReCoSoC.2017.8016146.

[30] Nocua, A., Bruguier, F., Sassatelli, G., and Gamatié, A. A gem5 trace-driven simulator for fast architecture exploration of openmp workloads. *Microprocess. Microsystems*, 67:42–55, 2019. doi: 10.1016/j.micpro.2019.01.008. URL https://doi.org/10.1016/j.micpro.2019.01.008.

[31] Novaes, M., Petrucci, V., Gamatié, A., and Pereira, F. Poster: Compiler-assisted adaptive program scheduling in big.little systems. In *24rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '19. ACM, 2019.

[32] Novo, D., Nocua, A., Bruguier, F., Gamatié, A., and Sassatelli, G. Evaluation of heterogeneous multicore cluster architectures designed for mobile computing. In Niar, S. and Saghir, M. A. R., editors, *13th International Symposium on Reconfigurable Communication-centric Systems-on-Chip, ReCoSoC 2018, Lille, France, July 9-11, 2018*, pages 1–8. IEEE, 2018. doi:

10.1109/ReCoSoC.2018.8449376. URL https://doi.org/10.1109/ReCoSoC.2018.8449376.

[33] Nugteren, C. and Corporaal, H. Bones: An automatic skeleton-based C-to-CUDA compiler for GPUs. *TACO*, 11(4):35:1–35:25, 2014.

[34] Péneau, P., Novo, D., Bruguier, F., Sassatelli, G., and Gamatié, A. Performance and energy assessment of last-level cache replacement policies. In *2017 First International Conference on Embedded Distributed Systems (EDiS)*, pages 1–6, 2017. doi: 10.1109/EDIS.2017.8284032.

[35] Péneau, P., Novo, D., Bruguier, F., Torres, L., Sassatelli, G., and Gamatié, A. Improving the performance of STT-MRAM LLC through enhanced cache replacement policy. In Berekovic, M., Buchty, R., Hamann, H., Koch, D., and Pionteck, T., editors, *Architecture of Computing Systems - ARCS 2018 - 31st International Conference, Braunschweig, Germany, April 9-12, 2018, Proceedings*, volume 10793 of *Lecture Notes in Computer Science*, pages 168–180. Springer, 2018. ISBN 978-3-319-77609-5. doi: 10.1007/978-3-319-77610-1_13. URL https://doi.org/10.1007/978-3-319-77610-1_13.

[36] Pereira, F. M. Q., Leobas, G. V., and Gamatié, A. Static prediction of silent stores. *TACO*, 15(4): 44:1–44:26, 2019. URL https://dl.acm.org/citation.cfm?id=3280848.

[37] Schapire, R. E. *Explaining AdaBoost*, pages 37–52. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-41136-6. doi: 10.1007/978-3-642-41136-6_5. URL https://doi.org/10.1007/978-3-642-41136-6_5.

[38] Senni, S., Delobelle, T., Coi, O., Péneau, P., Torres, L., Gamatié, A., Benoit, P., and Sassatelli, G. Embedded systems to high performance computing using STT-MRAM. In Atienza, D. and Natale, G. D., editors, *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, pages 536–541. IEEE, 2017. doi: 10.23919/DATE.2017.7927046. URL https://doi.org/10.23919/DATE.2017.7927046.

[39] Walker, M. J., Diestelhorst, S., Hansson, A., Das, A. K., Yang, S., Al-Hashimi, B. M., and Merrett, G. V. Accurate and stable run-time power modeling for mobile and embedded cpus. *TCAD*, 36(1):106–119, 2016.

[40] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., and Stenström, P. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3), May 2008. ISSN 1539-9087. doi: 10.1145/1347375.1347389. URL https://doi.org/10.1145/1347375.1347389.

[41] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., and Stenström, P. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008. ISSN 1539-9087. doi: 10.1145/1347375.1347389. URL http://doi.acm.org/10.1145/1347375.1347389.