Architectural exploration of heterogeneous memory systems

Marcos Horro, Gabriel Rodríguez, Juan Touriño, Mahmut T. Kandemir

***Abstract*—Heterogeneous systems appear as a viable design alternative for the dark silicon era. In this paradigm, a processor chip includes several different technological alternatives for implementing a certain logical block (e.g., core, on-chip memories) which cannot be used at the same time due to power constraints. The programmer and compiler are then responsible for selecting which of the alternatives should be used for maximizing performance and/or energy efficiency for a given application. This paper presents an initial approach for the exploration of different technological alternatives for the implementation of on-chip memories. It hinges on a linear programming-based model for theoretically comparing the performance offered by the available alternatives, namely SRAM and STT-RAM scratchpads or caches. Experimental results using a cycle-accurate simulation tool confirm that this is a viable model for implementation into production compilers.**

***Index Terms*—scratchpad memories, cache memories, heterogeneous architectures, dark silicon, power wall, memory wall, gem5.**

I. Introduction

THE end of Dennard scaling has brought processor performance to a near standstill in recent years, pushing architects towards designing increasingly parallel computers. Even with this paradigm shift, the power wall is expected to soon limit multicore scaling. A dark silicon future [1] has been proposed, in which chips will incorporate a high number of very specialized hardware such as vector execution units, tunable memory hierarchies and even different heterogeneous core technologies. In this paradigm, users, compilers and runtime cooperate to choose which parts of the core to use, under a certain power budget, for executing a given application under given QoS restrictions.

This paper focuses on the analysis and simulation of heterogeneous memory hierarchies using the gem5 simulator, a widely used tool developed by key players in the industry such as ARM, Intel, or Google. Its biggest potential is the ability to create new components or modify existing ones in order to perform architectural exploration. Extensions to simulate scratchpad memories in this framework are proposed, and used to assess the power-performance trade-off of different memory configurations and technologies. Profiling data is fed to an operational research framework that decides which memory units to enable from an available pool. This mathematical framework takes variable placement decisions by checking whether an access presents reuse that is easily exploitable by a regular cache, or whether the LRU algorithm and cache conflicts make it advisable to manually allocate the variable to a scratchpad to better take advantage of the available locality. Taking into account the hardware characteristics of the different available memory modules, access and transfer costs for each possible allocation are calculated, and a final allocation is decided considering also memory sizes and power budgets.

This paper is organized as follows. Section II introduces scratchpad memories. Section III analyzes the gem5 framework and discusses the simulation of scratchpad memories in the system. The mathematical model for data allocation is presented in Section IV. Experimental results using SRAM caches and STT-RAM scratchpad memories demonstrate the feasibility of the proposed approach, as shown in Section V. Finally, Section VI concludes the paper.

M. Horro, G. Rodríguez and J. Touriño are with the Department of Electronics and Systems, Universidade da Coruña, {`marcos.horro,grodriguez,juan`}`@udc.es`.

M. T. Kandemir is with the Department of Computer Science and Engineering, Pennsylvania State University, `kandemir@cse.psu.edu`.

II. Scratchpad memories

The importance and complexity of the on-chip memory hierarchy has increased with the advances in processor performance [2], in an attempt to bridge the gap between memory and processor speeds. Nowadays it is possible to find several different types of memories integrated in a single hierarchy, e.g., private vs. shared ones, or exclusive vs. inclusive. Scratchpad memories (SPMs) are one type of fast, random-access memories which are sometimes used as an alternative to cache memories. The main feature of scratchpads is its programmability, i.e., the possibility of handling the data allocated using special-purpose instructions placed by the compiler or programmer. In contrast, the contents of cache memories are controlled automatically by the hardware, for instance using the LRU algorithm. The use of scratchpad memories is widely extended in embedded systems, e.g. real time systems. SPM guarantees a fixed access latency whereas an access to the cache may result in a miss thereby incurring longer latency due to off-chip access [3] [4]. This unpredictability of caches is undesirable to meet hard timing constraints. Besides, scratchpad memories, having no tag array, are potentially more efficient energy- and performance-wise. Nevertheless, cache memories are largely used in general purpose processors, mainly because efficiently employing a scratchpad memory implies recompiling the code for different SPM features (e.g., size), while the LRU algorithm employed by caches is capable of automatically exploiting locality in a reasonable way. The use of one or the other flavor of on-chip memory becomes then a performance/effort choice. To exploit this trade-off, several specific purpose architectures have implemented scratchpad memories:

- Cell IBM [5]: this architecture was used in the nodes of the MareNostrum supercomputer at the Barcelona Supercomputing Center [6] and in the main core of PlayStation 3, as well as in premium Toshiba TVs [7], amongst others.
- PlayStation 2 [8]: this console included small scratchpad memories managed by the CPU and GPU.
- Digital Signal Processor (DSP) [9]: SODA's architecture includes a private scratchpad memory for each processing unit and a global scratchpad memory that can be accessed for any unit.
- Knights Landing's architecture Intel Xeon Phi [10]: this architecture uses MCDRAM memories. The processor has access to an addressable memory with high bandwidth, and therefore the concept is similar to that of scratchpad memories. However it is also remarkable that, in contrast with the previous examples, Intel Xeon Phi is a manycore architecture.

There has also been at least an implementation of scratchpad memories in a general purpose architecture, as Cyrix 6X86MX [11] already implemented a "scratchpad RAM". This 32-bit x86-compatible microprocessor, released in 1996, implemented a large Primary Cache of 64kB. This cache could be turned into a scratchpad RAM memory. The cache area set aside as scratchpad memory acted as a private memory for the CPU and did not participate in cache operations.

TABLE I
DIFFERENCES BETWEEN SIMULATORS ACCORDING TO THE FOLLOWING CRITERIA: WHETHER THE PLATFORM IS OR NOT IN CURRENT DEVELOPMENT, INSTRUCTION SET ARCHITECTURE (ISA) SUPPORTED AND ACCURACY. DATA EXTRACTED FROM [12] [13]

Sim.	**Dev.**	**ISA(s)**	**Accuracy**
Simics	Yes	Alpha, ARM, M68k, MIPS, PowerPC, SPARC, x86	Functional
SimFlex	No	SPARC, x86 (requires Simics)	Cycle
GEMS	No	SPARC (requires Simics)	Timing
m5	No	Alpha, MIPS, SPARC	Cycle
MARSS	No	x86 (requires QEMU)	Cycle
OVPsim	Yes	ARM, MIPS, x86	Functional
PTLsim	No	x86 (requires Xen and KVM/QEMU)	Cycle
Simple Scalar	No	Alpha, ARM, PowerPC, x86	Cycle
gem5	Yes	Alpha, ARM, MIPS, PowerPC, SPARC, x86	Cycle

III. INTEGRATION OF SCRATCHPAD MEMORIES IN GEM5

Simulators are crucial tools in architectural development and exploration, since they allow reasonably accurate estimations with almost negligible cost in comparison with lithographic or FPGA solutions. Thus, there have appeared many approaches with different granularity focused on concrete subsystems and other with holistic nature. Table I compares different simulators with both private licenses, as Simics, and open source licenses, which are the rest.

gem5 is a framework for performing cycle-accurate computer architecture simulations. The main reasons for choosing gem5 over the rest in this work have been: (i) its current development and implication of different organizations such as Google, Intel or ARM, as well as its large community of users; (ii) its open source license; and (iii) the possibility of simulating different ISAs. In addition, the modularity of the platform allows an easy modification of the system and the integration of new components. It provides two modes of operation: full system and system call emulation mode. The latter is interesting for executing benchmarks without loading an OS image. Regarding memory hierarchy simulation, there are two modes: classic and Ruby. An advantage of the classic mode is its simplicity. In the Ruby system memories are specified as finite state machines, and it is focused on studying the impact of using different cache coherence protocols.

In this paper, the main goal of including scratchpad memories in the system is to reduce energy consumption. Cache memory energy constitutes a major part of modern processor consumption [14] [15], the main reason being the need to activate both the tag and data regions to obtain a certain line, while a random-access memory only needs a single array access. Besides, cache misses also imply penalties in energy consumption.

Moreover, the concept of scratchpad memories does not differ from conventional memories, i.e., RAM memory, as their contents can be allocated and programmed by the application code. Thus, in order to implement scratchpad memories (SPM) in gem5, the starting point has been the memory modules already implemented (`SimpleMemory` class). We have adapted this implementation in order to add new parameters, such as latencies.

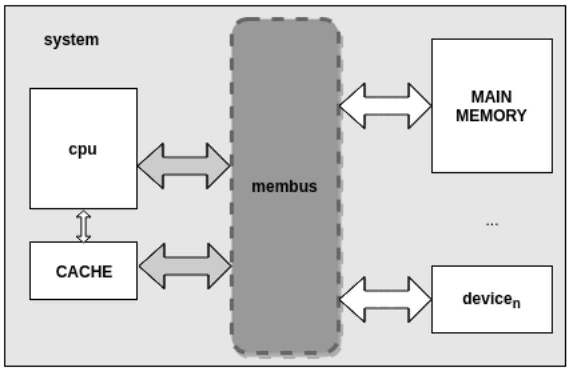

Fig. 1. Original architectural scheme in gem5

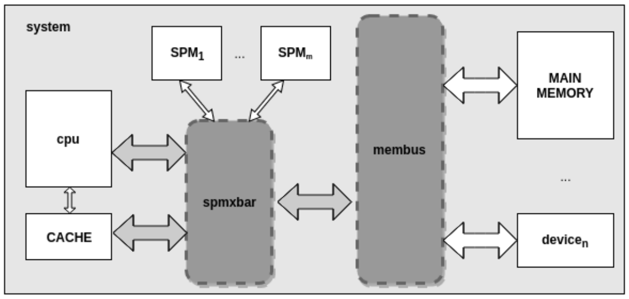

Fig. 2. Architectural scheme including SPM in gem5

Regarding the connection of these memories to the system, we considered enabling a new and specific port in the CPU. However, this was deemed a nonportable approach, since each different processor included in gem5 would need to be independently adapted. For this reason, we decided to use the already implemented crossbars. In the gem5 classic memory system, CPUs are provided with three ports: a system port and two ports for data and instruction caches. The system port, by default, is connected to the main bus (`membus`) where all memory devices are connected, e.g. main memory, I/O devices, etc. We modified these connections, creating a new crossbar which induces no latencies nor overheads where scratchpad memories are connected. This configuration connects CPU, cache memories, `membus` and scratchpad memories. Basically, this crossbar acts like a "bridge" between all the components. Figures 1 and 2 illustrate the difference between the original configuration and the modified architecture proposed.

The last step in order to integrate these memories in the system is the possibility to access their content. For this purpose the original ISA has been modified (in our case we chose x86 for familiarity reasons), adding an instruction to explicitly allocate a range of physical addresses onto the scratchpad memory. In order to simplify this allocation, the physical range chosen starts after the main memory range. This way, in order to integrate more than one scratchpad memory in the system, the next SPM's range starts right after the scratchpad memory instantiated before, i.e:

$$\begin{aligned}\text{range}(SPM_1) =& [|\text{main memory}|, |\text{main memory}| + |SPM_1|]\\ \text{range}(SPM_i) =& [\text{last\_addr}(SPM_{i-1}) + 1,\\ & \text{last\_addr}(SPM_{i-1}) + |SPM_i| + 1]\\ \forall i >& 1\end{aligned}$$

Thus, this instruction maps dynamically the region given, returning a reference in the program. This instruction has been generalized, allowing the reservation of memory in different scratchpad memories.

IV. Mathematical Model

Given that the proposed architecture integrates programmable memories, it is necessary to decide where to allocate a specific variable of a program. In other words, decide whether a variable should reside into main memory and be accessed through a regular cache, or whether it should be copied to one of the available scratchpad memories.

We propose a linear programming system that minimizes a target function to control memory allocation. One advantage of this approach is the possibility to change the objective function easily to target performance and even add or remove restrictions to model total power limitations, memory sizes, or other types of QoS. In the scope of this paper, the target function is the dynamic energy (E_{DYN}), calculated as the addition of the energies of each individual access, either read or write, multiplied by the energy consumption factor of each memory (i.e. the dynamic energy consumed by each read or write) and a binary decision variable. The introduced restrictions are: (i) a variable can only be allocated to a single memory, and (ii) the total size of the variables allocated to a scratchpad memory must be less or equal to the size of such memory. The size of each variable depends on the type of the variable (e.g. double, integer, etc.) and the number of elements in case of a k-dimensional vector. This proposal is captured in the equations in Figure 3, where $|\psi|$ is the size of the variable ψ, $|SPM_i|$ is the size of SPM_i, $f_i(\psi)$ refers to the energetic cost of accessing scratchpad i for variable ψ, and analogously $h(\psi)$ refers to the energetic cost of accessing main memory. N_r and N_w refer to the number of reads and writes respectively.

$$\psi \in \Psi \equiv \text{ set of variables}$$

$$\min E_{DYN} = \sum_{\psi\in\Psi}\Big(M(\psi)h(\psi) + \sum_{i=1}^{n} S_i(\psi)f_i(\psi)\Big)$$

$$S_i(\psi) = \begin{cases}0 & \psi \text{ in } SPM_i\\ 1 & \psi \text{ not in } SPM_i\end{cases}$$

$$M(\psi) = \begin{cases}0 & \psi \text{ in main memory}\\ 1 & \psi \text{ not in main memory}\end{cases}$$

$$M(\psi) + \sum_{i=1}^{n} S_i(\psi) = 1; \forall\psi \in \Psi$$

$$\sum_{\psi\in\Psi} S_i(\psi)|\psi| \le |SPM_i|, \forall i \in [1, n]$$

where

$$|\psi| = \text{type}(\psi) * N_\psi$$

$$f_i(\psi) = \sum_{\psi\in\Psi} E_{SPM\_r}(i)N_r(\psi) + E_{SPM\_w}(i)N_w(\psi)$$

$$h(\psi) = \sum_{\psi\in\Psi} E_{MM\_r}N_r(\psi) + E_{MM\_w}N_w(\psi)$$

Fig. 3. Linear programming system

Since energy savings from using scratchpad memories come from the reduction of accesses to main memory, the proposed system would recommend allocating the variables that fit into the scratchpad memory. This can be detrimental in some cases, e.g., if an LRU cache could take advantage efficiently of the locality present in the pattern of accesses of a given variable (cache-friendly access). Hence, we decided to introduce a new restriction: force a particular variable to be allocated to main memory if the code accesses it in a cache-friendly way. This restriction is reflected in Equation (1).

$$C(\psi) = \begin{cases} 0 & \psi \text{ not } \textit{cache-friendly} \\ 1 & \psi \textit{ cache-friendly} \end{cases} \tag{1}$$
$$M(\psi) \geq C(\psi) \ \ \forall \psi \in \Psi$$

The value of this binary variable allows to introduce external information, e.g., data obtained by profiling or analyzing data at compile time. In our current system, the value of this variable is determined in a very simple analytical way: a regular access to a variable is considered cache-friendly if two consecutive accesses to a k-dimensional array ($k > 1$) in a loop nest may reside in the same cache line.

As a corollary, the following expressions are exposed and commented briefly:

$$A[i, j] \equiv \text{regular, cache-friendly access}$$
$$A[i, B[j]] \equiv \text{irregular access, considered non cache-friendly}$$
$$A[i, j * 8] \equiv \text{regular, but not necessarily cache-friendly access}$$

In the last case the cache-friendliness of the access depends on the data type and cache line size. For instance, working with 8-byte double values and 64-byte cache lines two consecutive accesses would never reside in the same cache block. In the general case:

$$\exists a_i \ \ / \ \ addr(a_{i+1}) - addr(a_i) \leq |cb| \tag{2}$$

Where a_i and a_{i+1} are two consecutive accesses, $addr(x)$ is the address of x and $|cb|$ is the cache line size. With the above, we have developed a prototype of this integer linear programming system in R. We currently do not consider temporal locality between different accesses to the same variable.

V. Experimental results

A. *Configuration*

Besides implementing the architecture proposed in the simulator, it is important to test its performance and compare it with other architectural alternatives. For this purpose, we have chosen the PolyBench/C suite [16]. The main advantages of this suite are its open source license and having simple kernels to analyze for our model, i.e., nested loops with affine accesses, making it simple to calculate the number of reads and writes of each variable.

Another important aspect in order to perform a correct configuration of our architecture is the selection of the characteristics of our memory modules, i.e., energy consumption and latencies. For this purpose we have used CACTI [17] and NVSim [18]. Actually, NVSim is a CACTI extension, since it works using this tool, but adding other features and technologies. These tools have been used to validate the configurations of our architectures.

We have tested our proposal using two different architectures: a general purpose one, and a modified architecture including scratchpad memories. The general purpose architecture consists of an x86 processor and a traditional memory hierarchy: two cache levels and a main memory (see Figure 1). Regarding the modified architecture (see Figure 2), the most remarkable detail of the configuration is the inclusion of a scratchpad memory between the CPU and main memory, as explained before. In order to perform meaningful comparisons, this scratchpad memory replaces the last-level cache memory in the traditional memory hierarchy. This is an area equivalent replacement, in other words, the L2 cache, is replaced with a scratchpad memory which occupies roughly the same die area. As a proof of concept of our system we explore, besides SRAM, the usage of STT-RAM memories, providing better energy consumption and capacity characteristics. Table II summarizes the different memory configurations used in our tests, including the energy consumed for both read and write accesses and also the leakage power of each module. Note the huge difference in the leakage power of both technologies. STT-RAM also has a higher density, increasing the size of the memory modules by a factor of approximately four; however in detriment of the access latencies. The results of this experiment exemplify the energetic and temporal trade-off in the technological selection.

B. *Results*

Throughout this experiment we have talked about different features: area, latencies and energy, mostly. In the section above we talked about the different configurations of the memory hierarchy, also referring to the different technologies. Nevertheless, the theoretical premises may not be reflected in the execution of the benchmarks.

In order to measure the execution time of the programs, we set breakpoints at the interest regions, focusing on the kernel of the program (the SCoPs [19] in PolyBench codes). Thus gem5 gives us statistics about the execution time and energy consumed by the memory hierarchy. In order to obtain homogeneous results, we have used gem5 to obtain the number of references to SPM, both reads and writes, as well as the execution time of the kernel of the program. With this data it is possible to calculate the dynamic energy consumed by scratchpad memories (see Equation 4 below). Nevertheless, energetic results given by gem5 are quite scarce regarding memories, and in order to obtain more detailed results, we have used McPAT [20]. For this purpose we have developed a novel parser (gem5McPATparse [21]) that searches the output files of gem5 for the parameters and statistics that serve as input for McPAT, and translates them generating the corresponding XML input file. These translations are based on [22]. In this way, the energy consumed by cache memories is calculated as the addition of the dynamic power and leakage power multiplied by the execution time (see Equation 5).

Finally, static energy is calculated using the architectural parameters provided by NVSim and the execution time of a particular program (see Equation 3). The addition of all equations is reflected in Equation 6.

$$E_{STATIC}(t) = t_{exec} * P_{leakage} \tag{3}$$

$$E_{SPM} = N_r * E_{SPM\_r} + N_w * E_{SPM\_w} \tag{4}$$

TABLE II
COMPARISON OF DIFFERENT TECHNOLOGIES. EACH ROW CORRESPONDS TO A CERTAIN CACHE OR SCRATCHPAD MEMORY CONFIGURATION.

SRAM (caches)							
cache	mm^2	Latencies (ns)			Energies		
kB		Read	Miss	Write	Hit/miss (pJ)	Write (pJ)	Leak (mW)
256	0.229	2.258	0.083	1.588	72	25	336.330
512	0.380	2.669	0.107	1.996	112	21	600.112
1024	0.741	3.452	0.144	2.773	214	36	1180.407
2048	1.343	9.989	0.149	7.941	378	24	2141.436
4096	2.619	11.52	0.222	9.037	383	290	4288.790
STT-RAM (scratchpads)							
SPM	mm^2	Latencies (ns)			Energies		
kB		Read	Miss	Write	Read (pJ)	Write (pJ)	Leak (mW)
1024	0.183	2.221	N.A.	5.686	195.251	205.024	84.809
2048	0.348	2.364	N.A.	5.744	228.512	242.614	146.194
4096	0.696	2.499	N.A.	5.812	276.137	290.231	292.389
8192	1.311	3.055	N.A.	6.038	388.324	383.871	568.592
16384	2.488	5.036	N.A.	7.739	516.687	465.678	640.935

$$P_{cache} = P_{cache\_LEAK} + P_{cache\_DYN}$$
$$E_{DYNcache} = P_{cache} * t_{exec} \quad (5)$$
$$E_{DYN}(t) = E_{MM} + E_{DYNcache} + E_{SPM}$$

$$E_{TOT}(t) = E_{STATIC}(t) + E_{DYN}(t) \quad (6)$$

Each execution was performed on an Intel Xeon E5-2660 Sandy Bridge 2.20 Ghz node, with 64 GB of RAM. All the results have been normalized with respect to the results obtained for `cache_256kB`, which corresponds to the first row of Table II. The most remarkable aspects observed in these executions are commented in the following paragraphs.

Regarding **temporal performance**, scratchpad architectures present worse results with small matrix sizes, in other words, with a small working set. The explanation for this issue is the fact that the decision system forces arrays with non cache-friendly accesses to be allocated to scratchpad memory. As a consequence, for small working sets where L1 cache could allocate this data without damaging locality the scratchpad memory increases latencies and dynamic energy consumed. The reason is that the SPM has been designed as a replacement for L2, but it bypasses L1 in these situations. Nevertheless, when the size of these arrays increases, the decision system plays a major role since the L1 cache can no longer allocate all the desired data, provoking conflicts when accessing new elements. In these cases, using scratchpad memories mitigates cache memory penalties caused by non cache-friendly accesses and capacity issues. This can be observed in Figure 4 for the `2mm` benchmark. Still, there are also cases where this behavior is missing, as shown in Figure 5 for the `bicg` code. In this case, the decision system does not provide any performance enhancement since the data set works properly through cache memory, and losing the last-level cache is counterproductive. Analyzing the output of the decision system for the benchmarks whose performance worsens with working set size, we conclude that the reason for this is that some of the arrays which should be allocated to SPM from a locality point of view do not fit the available SPM space. Future versions of the decision system need to incorporate tiling transformations, or some means to store large arrays in a piecewise fashion into the SPM so that locality can be exploited in these cases.

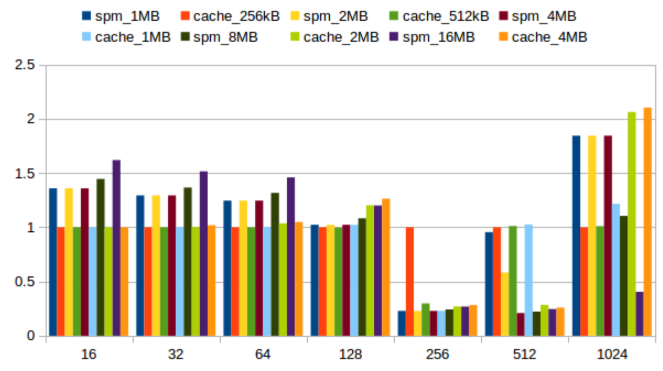

Fig. 4. Normalized execution time of the `2mm` benchmark

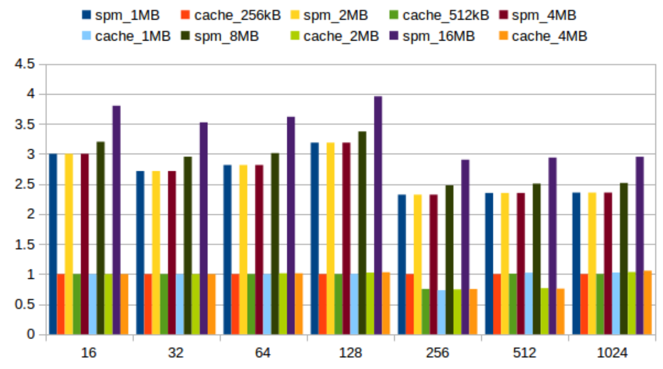

Fig. 5. Normalized execution time of the `bicg` benchmark

Following a similar pattern to temporal performance, the **dynamic energy** in architectures with scratchpads is higher for small working sets, since the L1 cache has lower dynamic energy per access and no capacity issues arise. Similarly, when increasing the size of arrays, architectures without a scratchpad memory issue more accesses to main memory, raising the dynamic energy consumed (see Figures 7 and 8). Nonetheless,

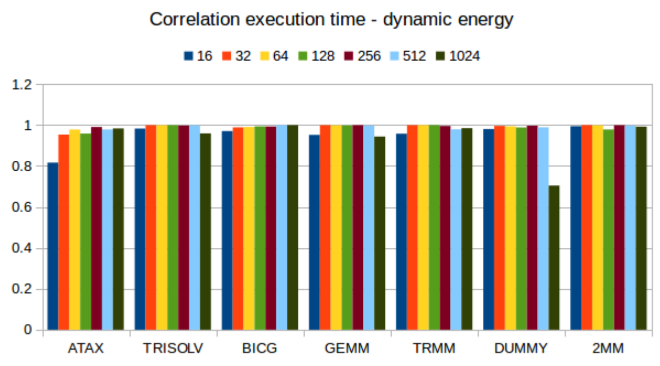

Fig. 6. Correlation between execution time and dynamic energy for all benchmarks

it is also remarkable what we can observe in `bicg`: there is no performance improvement and the dynamic energy is also significantly higher. The increase in dynamic energy is caused by the same issues identified when analyzing performance. In fact, the correlation between execution time and dynamic energy is extremely high in all cases, as illustrated in Figure 6.

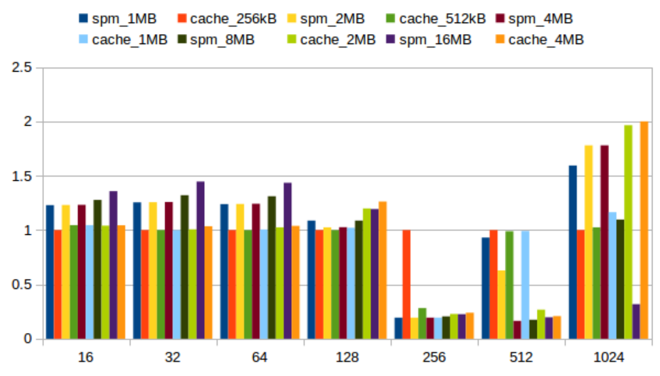

Fig. 7. Normalized dynamic energy of the `2mm` benchmark

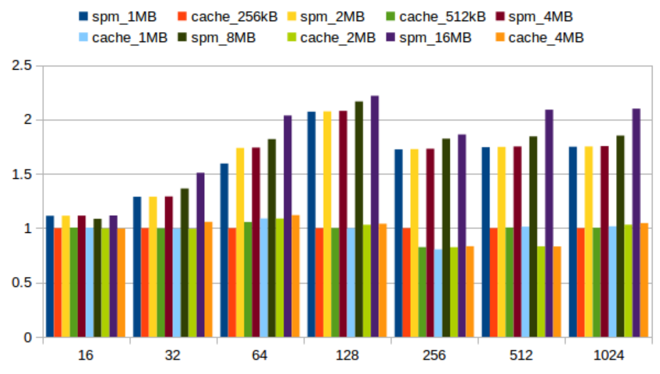

Fig. 8. Normalized dynamic energy of the `bicg` benchmark

Regarding **static energy**, as expected, despite temporal differences, in general scratchpad memories have a lower static consumption due to the technology chosen (STT-RAM against SRAM). This situation can be observed in Figures 9 and 10.

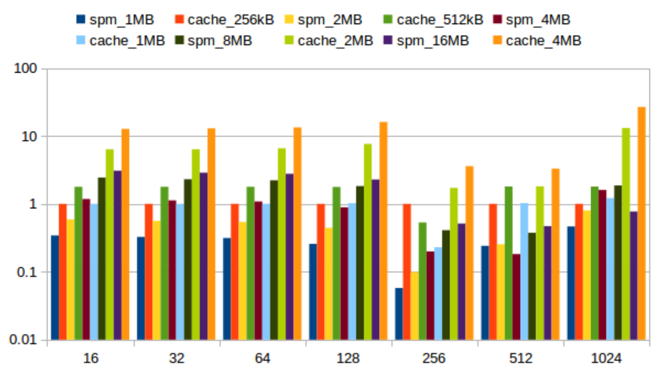

Fig. 9. Normalized static energy of the `2mm` benchmark

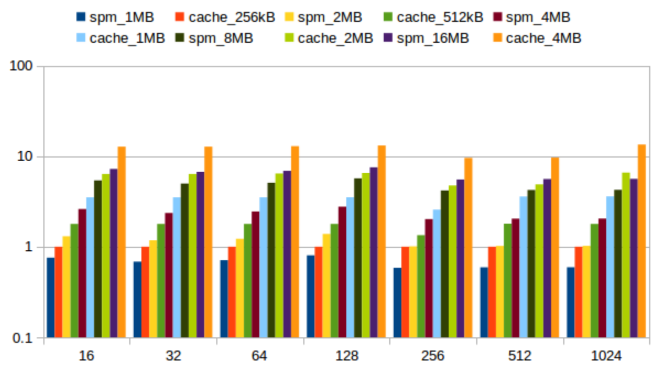

Fig. 10. Normalized static energy of the `bicg` benchmark

The executions for the remaining benchmarks are illustrated in Figure 11. It is readily observable the repetition of the patterns commented before in the rest of the benchmarks.

VI. Discussion

The imperious requirement to use efficiently the available resources in our system presents interesting challenges. In this article general purpose architectures using different memory hierarchies are analyzed. For this purpose well-known modeling and simulation tools have been used in order to demonstrate the reliability of the results. A linear programming system to decide to which available module a certain variable of the program must be allocated in order to improve energy consumption has been developed. We have compared cache and RAM memories equivalent in terms of area, using SRAM and STT-RAM technologies, respectively. Experimental results demonstrate that: (i) without a remarkable penalty in time, energy efficiency can be improved, mostly due to low leakage power in scratchpad memories using STT-RAM; (ii) the use of scratchpad memories with a simple decision algorithm can also improve temporal performance for certain benchmarks; (iii) there are some cases where the algorithm does not choose properly the location of variables, but this behavior is easily correctable considering the size of the L1 cache; and (iv) as the size of the working set grows, the energy improvements are more evident due to, in some cases, the success of the decider choosing the most profitable location for a variable, but mostly due to the weighing of leakage power.

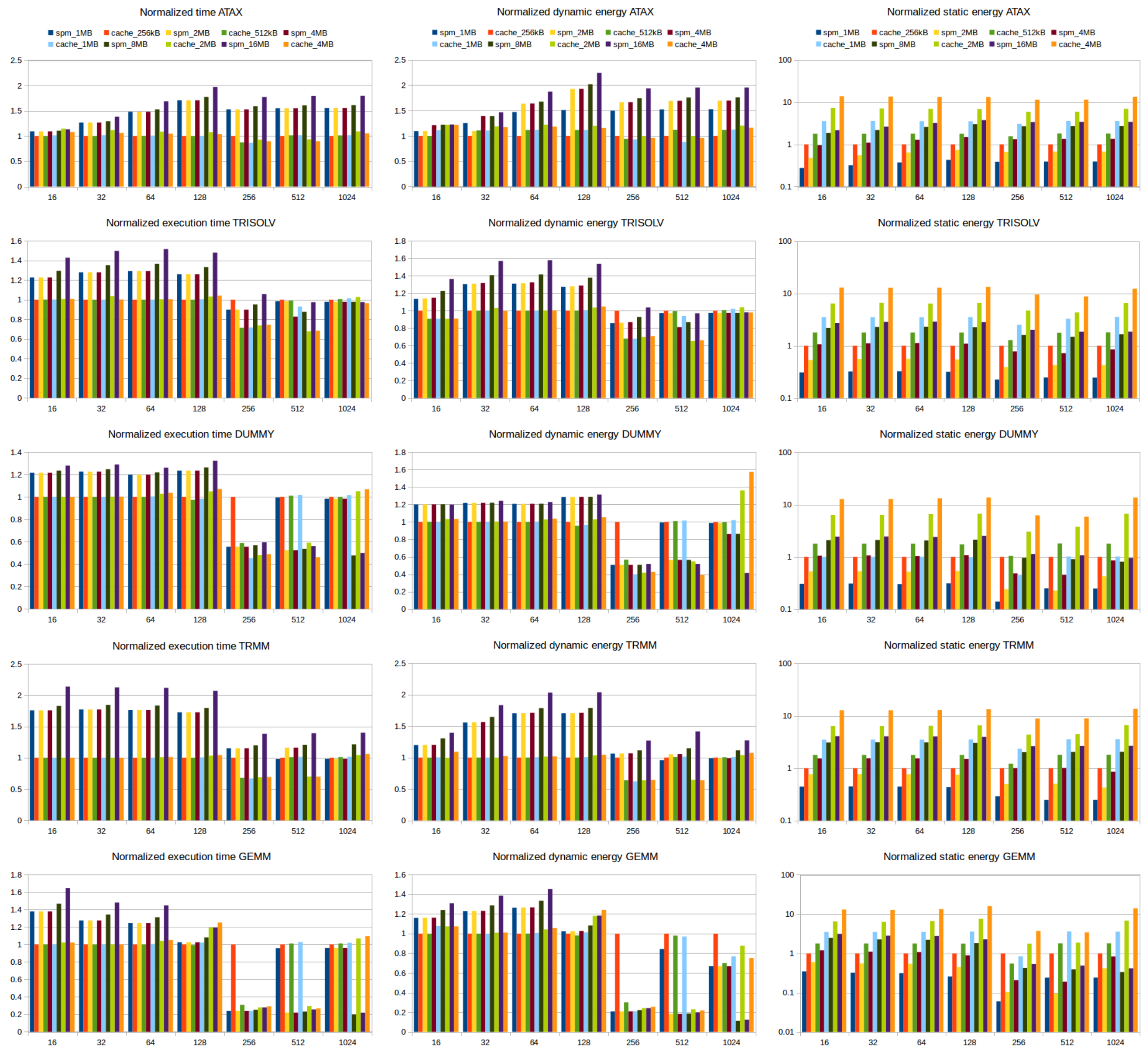

Fig. 11. Results of all executions

VII. Acknowledgements

This work is supported by the Ministry of Economy and Competitiveness of Spain and FEDER funds of the EU (Project TIN2013-42148-P).

References

[1] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, New York, NY, USA, 2011, pp. 365–376. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000108

[2] L. Yan, L. Dongsheng, Z. Duoli, D. Gaoming, W. Jian, G. Minglun, W. Haihua, and G. Luofeng, "Performance evaluation of the memory hierarchy design on CMP prototype using FPGA," in *IEEE 8th International Conference on ASIC*, Oct 2009, pp. 813–816.

[3] Vivy Suhendra, Chandrashekar Raghavan, Tulika Mitra, "Integrated Scratchpad Memory Optimization and Task Scheduling for MPSoC Architecture," *School of computing. National University of Singapore*, 2006.

[4] B. Anuradha and C. Vivekanandan, "Usage of scratchpad memory in embedded systems 2014; state of art," in *Third International Conference on Computing Communication Networking Technologies (ICCCNT)*, July 2012, pp. 1–5.

[5] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic processing in cell's multicore architecture," *IEEE Micro*, vol. 26, no. 2, pp. 10–24, March 2006.

[6] Barcelona Supercomputing Center (BSC), "Cell Superscalar (CellSs) User's Manual," http://www.bsc.es/media/2296.pdf, 2009.

[7] M. Takayama and R. Sakai, "Parallelization Strategy for CELL TV," in *1st Workshop on Applications for Multi and Many Core Processors*, 2010.

[8] T. M. Conte, *Computer Architecture: A Quantitative Approach. Appendix E: Embedded Systems*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[9] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, "Soda: A low-power architecture for software radio," *SIGARCH Comput. Archit. News*, vol. 34, no. 2, pp. 89–101, May 2006.

[10] A. Sodani, "Intel® Xeon Phi^{TM} Processor "Knights Landing" Architectural Overview," https://www.nersc.gov/assets/Uploads/KNL-ISC-2015-Workshop-Keynote.pdf, 2015.

[11] IBM, "IBM 6X86MX Microprocessor," http://datasheets.chipdb.org/IBM/x86/6x86MX/mx_full.pdf, 1998.

[12] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli, "Accuracy evaluation of gem5 simulator system," in *7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, July 2012, pp. 1–7.

[13] A. Gutierrez, J. Pusdesris, R. Dreslinski, T. Mudge, C. Sudanthi, C. Emmons, M. Hayenga, and N. Paver, "Sources of error in full-system simulation," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2014, pp. 13–22.

[14] R. Banakar, S. Steinke, L. Bo-Sik, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: Design alternative for cache on-chip memory in embedded systems." *In Proceedings of the 10th International Symposium on Hardware/Software Codesign*, pp. 73–78, 2002.

[15] G. Rodríguez, J. Touriño, and M. T. Kandemir, "Volatile STT-RAM Scratchpad Design and Data Allocation for Low Energy," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, pp. 38:1–38:26, Dec. 2014. [Online]. Available: http://doi.acm.org/10.1145/2669556

[16] L. Pouchet, "PolyBench/C: the Polyhedral Benchmark suite," http://web.cse.ohio-state.edu/~pouchet/software/polybench/, 2015.

[17] S. J. E. Wilton and N. P. Jouppi, "CACTI: an enhanced cache access and cycle time model," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 5, pp. 677–688, May 1996.

[18] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "NVSim: A Circuit-Level Performance, Energy, and Area Model for Emerging Nonvolatile Memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 7, pp. 994–1007, July 2012.

[19] A. Kumar and S. Pop, "SCoP Detection: A Fast Algorithm for Industrial Compilers," in *6th International Workshop on Polyhedral Compilation Techniques on IMPACT*, January 2016.

[20] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*, 2009, pp. 469–480.

[21] M. Horro, "gem5McPATparse," https://github.com/markoshorro/gem5McPATparse, 2016.

[22] F. Endo, "Online Auto-Tuning for Performance and Energy through Micro-Architecture Dependent Code Generation," Theses, Université Grenoble Alpes, Sep. 2015. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01285964