Architecture Extensions for Efficient Management of Scratch-Pad Memory

José V. Busquets-Mataix, Carlos Catalá, and Antonio Martí-Campoy

Department of Computer Engineering, Universidad Politécnica de Valencia,
Camino de Vera s/n, 46022-Valencia, Spain
{vbusque,amarti}@disca.upv.es, ccatala@grupoazahar.com

Abstract. Nowadays, many embedded processors include in their architecture on-chip static memories, so called scratch-pad memories (SPM). Compared to cache, these memories do not require complex control logic, thus resulting in increased efficiency both in silicon area and energy consumption. Last years, many papers have proposed algorithms to allocate memory segments in SPM in order to enhance its usage. However, very few care about the SPM architecture itself, to make it more controllable, more power efficient and faster. This paper proposes architecture extensions to automatically load code into the SPM whilst it is fetched for execution to reduce the SPM updating delays, which motivates a very dynamic use of the SPM. We test our proposal in a derivation of the Simplescalar simulator, with typical embedded benchmarks. The results show improvements, on average, of 30.6% in energy saving and 7.6% in performance compared to a system with cache.

Keywords: Embedded processors, memory architecture, scratch-pad memory.

1 Introduction

In recent years, the commercial popularity of mobile embedded devices such as phones, PDAs, cameras, MP4 players, etc. has attracted strong economic interests. As a consequence, much research effort has been accomplished to increase the computing power of such devices to be able to incorporate as much functionality as possible. However this increment in performance has been not accompanied by a equivalent increment in battery technology. Despite the great step forward due to *lithium-ion* batteries, since then, larger energy consumption requires larger battery size. Consequently, while battery technology slowly advances, the effort should be made to reduce the energy consumption of mobile embedded devices.

Compared to general purpose computing, there is an important key characteristic in these devices that may be exploited: most of the workload is somewhat fixed and known at design time. Therefore, some techniques may be used to allocate code and data objects to a lower stage in the memory hierarchy (i.e. SPM).

In general computing, caches have played a decisive role in providing the memory bandwidth required by processors. In fact, they became as one important technique to

J.L. Ayala et al. (Eds.): PATMOS 2011, LNCS 6951, pp. 43–52, 2011.

reduce the famous "memory bottleneck". The caches memory has a very dynamic and unpredictable behavior, capable of adapting its contents to any unknown workload. However, it is not energy efficient because it requires tag memory and the hardware comparison logic. Some authors have becoming to identify the memory subsystem as the energy bottleneck of the entire system [3].

High energy consumption of the cache, and predictable workload in embedded computing, has led the SPM memory to emerge as an efficient alternative to caches. In addition to its energy efficiency, it is fully predictable, playing an important role in real-time systems. The disadvantages are derived from the fact that the SPM is basically a small and fast memory mapped into the address space of main memory. Therefore, its operation must be done explicitly by mapping memory objects by the linker and loader, or by programming.

To do so, many approaches have been presented this decade to carefully select the contents to be stored in SPM to improve energy and/or performance. Orthogonal to them, this paper presents an original approach to reduce the overhead resulting from updating the SPM contents at run time.

The contributions of the paper are new architectural extensions to dynamically control the SPM. The difference among others solutions is that SPM loading is done on the fly whilst code is fetched from memory for execution, with minimum time and energy cost. This fact will enable allocating techniques to dynamically adapt the contents of the SPM to the program run at a reduced delay cost. Moreover, these techniques will trade-off to favor frequent updates, adapting the SPM contents to the program flow in a more effective and precise way

Our proposal only requires small changes on the processor design. Our interest is to obtain a realistic solution, simple enough to be implemented in a real world.

This paper is structured as follows. Section 2 reviews the related work. Section 3 proposes the overall architecture of our approach. The experimental setup is explained in section 4. Section 5 discusses the experimental results obtained. Finally, section 6 concludes the paper.

2 Related Work

In the literature, there are many works that focus on reducing the energy consumption and/or increasing performance by means of the effective use of SPM memories. These papers present the SPM as worthy alternative to cache memories, when energy and not only performance is important.

Many of these studies present a range of techniques on the allocation of code in the SPM which can be divided into two types. First, those of a static approximation where the contents of the SPM are assigned in advance and remain unchanged during program execution [1], [2], and [4]. Second, the ones that perform a dynamic update of the SPM contents at run time: Egger et al. [5] [6] and [7], Hyungmin Cho et al. [8], Janapsatyat et al. [9], Steinke et al. [10], Polleti et al. [11] , Lian Li et al. [13] and Doosan et al. [19]. The latter have the advantage of adapting the contents of the SPM to the program run but at the cost of periodically reloading the SPM contents.

There are some papers that propose hardware extensions to better control the SPM [9], [11], [12], and [13]. In [9] Janapsatyat et al. introduce a special set of instructions at compile time in a number of key points using a heuristic algorithm, which trigger a hardware controller that manages the flow of data to the SPM. To the best of our knowledge, this is the technique that better approximates to ours. However, the main advantage of our solution is that it requires fewer instructions and less control logic to operate.

Some papers propose the use of DMA to reduce the cost of copying data from main memory to the SPM [11], [19]. The main difference to our proposal is the larger die size and energy cost of this approach by using the DMA.

3 Architecture Extensions

Our proposal is based on a number of changes in the processor architecture, which may be classified in two categories. The first contain small changes required in the processor hardware design in order to support our approach. Second, three new instructions have been added to the instruction set. Below we explain these changes.

3.1 Hardware Design

Basically, the memory hierarchy is composed by the SPM and main memory. However, in order to take advantage of the spatial locality, we have added a prefetching buffer (see figure 1). This buffer behaves like a small cache memory with only one line in size. It will help in reduce the energy power and latency for those sequential fragments of code that are not selected to reside in SPM.

The SPM will be updated dynamically at run time on the fly to contain loops and functions that are executed frequently. No explicit load instructions are needed. The processor has three execution modes: *memory* mode, *SPM* mode, and *SPM function* mode. Changes among modes are controlled by three specific instructions.

In *memory* mode, instructions are brought to instruction decoder from main memory through the one-line-buffer (OLB) to exploit spatial locality. In *SPM* and *SPM function* mode, instructions are fetched from SPM memory. However, before the instructions may be used from SPM, they should be loaded to SPM from memory. This mechanism may be compared to a cache miss.

These modes require some hardware changes. We add a second program counter, so called SPM_PC and a tag register containing the memory address of the first instruction in SPM. A refinement of the technique proposes that the SPM may be split in independent partitions or blocks. Each one will include a tag register (as shown in figure 2). This schema may be used to hold in SPM different functions and/or loops at the same time. However, this architecture is not comparable to cache, since SPM partitions are much larger than cache lines, and consequently, there are only few tag registers in SPM compared to cache. We also need a small tag controller for comparison and update, and a mechanism to invalidate the whole SPM partition in one cycle.

3.2 Architectural Issues

To deal with the executions modes proposed in former section, three new instructions have been added to the processor architecture: *SPM_start*, *SPM_call_start* and *SPM_end*. Both *SPM_start* and *SPM_call_start* include an immediate field containing the SPM block number to be used. This block is selected by the programmer. These instructions are inserted into the original code by the compiler or programmer to tell the processor which pieces of code are selected to execute from SPM. For instance, when a loop is selected, two instructions are inserted to mark the bounds of the code: *SPM_start* at the beginning of the loop code, and *SPM_end* at the end.

When a *SPM_start* instruction is executed, the following actions take place: first, processor changes to *SPM* running mode. Second, the counter SPM_PC is initialized to the beginning of the SPM block (explicitly chosen by the *SPM_start*). Next, the physical address of the instruction *SPM_start* is compared to the tag register corresponding to the chosen SPM block. In case both addresses are equal, it means the contents of the SPM block correspond to the instructions in main memory that follow the *SPM_start*. Thus, the code is fetched from SPM. Both SPM_PC and PC counters are incremented simultaneously to point to two instances of the same instruction, one in main memory, and a copy in SPM. This schema will allow continuing execution from main memory once the end of the SPM code is reached (either by reaching the end of SPM block, or reaching *SPM_end* instruction). This allows dealing with loops larger than the SPM block. Knowing SPM size, loops may also spread several SPM blocks.

If tag comparison misses, the running code is not in the SPM block. The new starting address is copied to the tag register and the SPM block contents are invalidated. Next, the instructions are fetched from main memory to perform both, SPM load, and execution. This operation may be compared to a cache cold start. For the second and following pass of the loop, instructions are fetched from SPM.

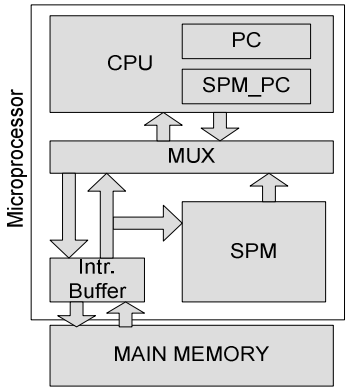

Fig. 1. Architecture

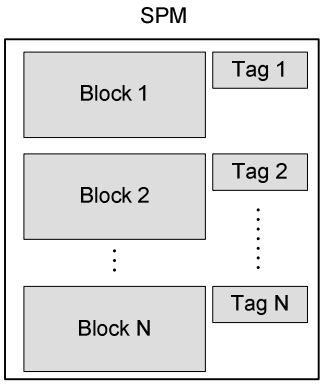

Fig. 2. SPM blocks

The former mechanism works well for loops and pieces of code frequently used. However, we propose an additional instruction, *SPM_call_start,* to deal with functions that the source code is not available to the programmer (i.e. functions in compiled libraries). It works as follows:

When the PC reaches a *SPM_call_start*, processor changes to *SPM_function* mode. Instructions are fetched from memory until a "call to function" instruction is found. Once the jump is taken, the target address (beginning of the function) is compared to the tag of the selected SPM block. The process that follows is somewhat similar to the one described for the *SPM_start* instruction. There is only one additional difference. The processor mode is changed to *memory* mode, once the end of the function is reached (return instruction). Therefore, it is not necessary the insertion of the *SPM_end*. This process may be seen in figure 3. Any call instruction to allocated functions, must be preceded by a *SPM_call_start*. Otherwise, the functions will be executed normally from main memory, without any SPM benefit.

There are also some particular cases that require a detailed explanation. The SPM is loaded at the same time instructions are brought from memory for execution. Therefore, it is possible to have some instructions only in memory until a given iteration requires their execution (i.e. *if then else* structure inside the loop). The processor must realize whether a location in SPM contains a valid instruction. In cache, this is solved at line by line basis, through costly tag comparison. Our approach cope with this issue performing a quick erase (invalidate) of the entire SPM block. This is accomplished in one processor cycle by a special hardware attached to the SPM memory addressing circuitry. The approach takes care to avoid any overhead in controlling the SPM. See figure 4 for the overall fetching process.

The SPM is mapped in the same physical address space than main memory. A key benefit of our approach is that it does not require virtual memory manager, allowing its use in medium to small embedded processor. However, we have to take special care of any flow change (*jump, call*). For a piece of code that is brought from memory to SPM, for the point of view of the architecture, the instructions are changing their physical addresses.

The main structures that require jump instructions are *loop* and *if then else*. Both of them use branch instructions with relative offset, thus no absolute addressing is necessary. Regarding function calling, the returning address is stored in stack. The processor have to select either pushing the SPM_PC or the PC depending on where is placed the call instruction. When the return is executed, the program flow returns to the caller code, irrespective it is in SPM or in memory. The processor has to switch automatically between *memory* and *SPM* modes.

4 Experimental Setup

In order to compare the cache against the SPM, we have used the simulator *Vatios* [14]. *Vatios* is a simulator based on the popular *SimpleScalar* framework [15]. Similarly to *Wattch* simulator [16], *Vatios* adds a model to calculate the energy consumption of both entities, memory and processor.

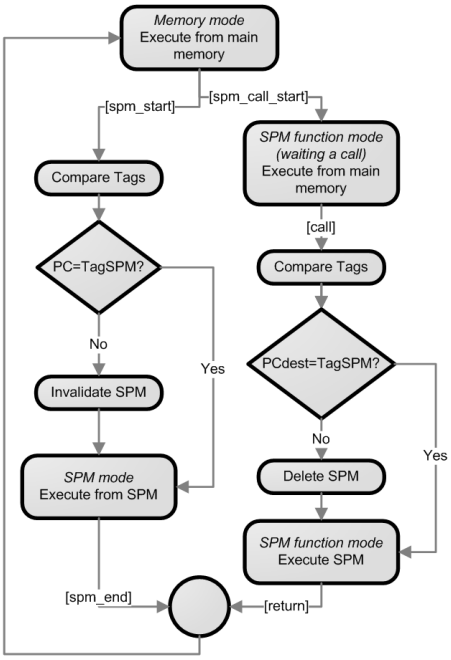

Fig. 3. Fetching process

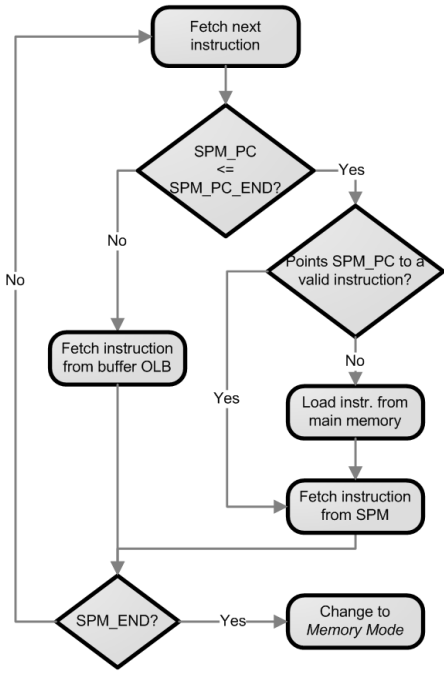

Fig. 4. Instruction fetch in SPM mode

Vatios presents a series of advantages in the calculation of the energy consumption with respect to *Wattch*. To calculate the energy consumption of the SPM and the cache, *Vatios* is based on the energy model called *Cacti* [17]. The SPM energy efficiency are due basically to the reduced control circuitry compared to the cache. To allow a fair comparison, both memories have been simulated using the same manufacturing technology.

Attending to the intended target architecture of this technique, we have selected realistic benchmarks. In particular, we have chosen a collection of programs selected by the The Mälardalen WCET research group [18]. They are representative programs for embedded systems, and mainly intended to be used in WCET analysis tools. We have selected the following:

Bsort100: Bubblesort program. Tests the basic loop constructs, integer comparisons, and simple array handling by sorting 100 integers.

Cnt: Counts non-negative numbers in a matrix. Nested loops, well-structured code.

Compress: Data compression program. Adopted from SPEC95 for WCET-calculation. Only compression is done on a small buffer containing totally random data.

Cover: Program for testing many paths. A loop containing many switch cases.

Expint: Series expansion for computing an exponential integral function. Inner loop that only runs once, structural WCET estimate gives heavy overestimate.

Fdct: Fast Discrete Cosine Transform. Many calculations based on integer array elements.

Fir: Finite impulse response filter (signal processing algorithms) over a 700 items long sample. Inner loop with varying number of iterations, loop-iteration dependent decisions.

The processor architecture of the simulator has been modified to incorporate the proposed approach. Since the simulator version that we used only offers a cache memory, we have implemented the SPM from scratch. We have considered the speed and energy models for this kind of memory. The decoder unit has accommodated the new instructions and, the necessary additional registers (SPM program counter) have been added. We have validated the correctness of the implemented extensions by exhaustive running of real workload.

The SPM control instructions have been inserted into the code by heuristics. We have not used any automatic allocation technique. The hot spots in the program can be easily identified by profiling. Benchmarks that are focused to WCET have help in this task.

The experimental process is as follows: first, from the C source code of benchmarks, we have added the SPM control instructions. The resulting code is compiled with *sslittle-gcc*. The binary programs are simulated by the *Sim-Vatios* simulator to obtain a trace. This trace is used by the tool *Power-Vatios* to obtain the energy consumption.

Regarding the hardware configuration, the benchmarks are simulated over three different cache or SPM sizes: 128, 256 and 512 bytes. The cache is direct mapped. The SPM has only one block. This is due the fact that the programs considered do not have concurrent hot spots. Therefore, the optimal configuration is a larger and unique block, but it may be updated frequently, thanks to the reduced overhead of the approach.

5 Experimental Results

The obtained results are depicted in figure 5 to 10. We can see that for 128B (Figures 5 and 6) the SPM performs better in both performance and energy consumption across all benchmarks. The SPM provides an average improvement in performance by 17%, and 29% in energy consumption with respect to the cache. The better results are displayed for the *fir* benchmark in which our approach obtains an improvement in performance of 44% and energy consumption 53%.

For 256B sizes (Figures 7 and 8) for the eight benchmarks used, only *expint* has better performance using a cache. This program consists of two nested loops where the outer loop cannot be entirely placed into the SPM and the most inner loop is executed only under certain circumstances. This makes that the cache takes advantage for this case. Summarizing, the SPM 256B has a 9% improvement in performance and 31% in energy consumption with respect to the cache.

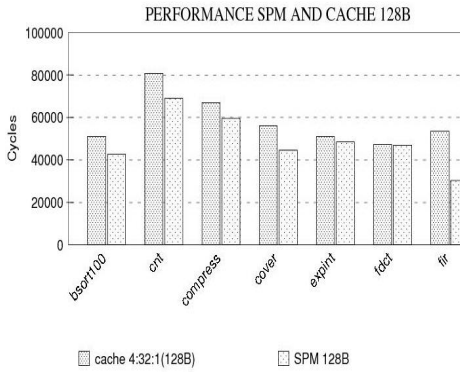

Fig. 5. Perfomance 128B

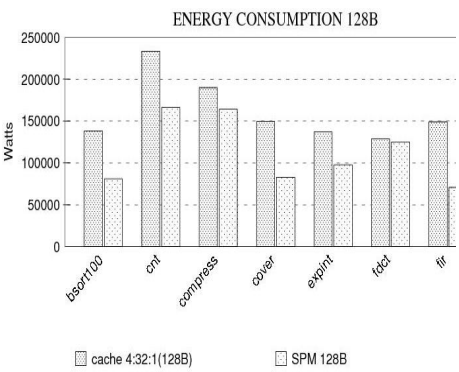

Fig. 6. Energy comsumption 128B

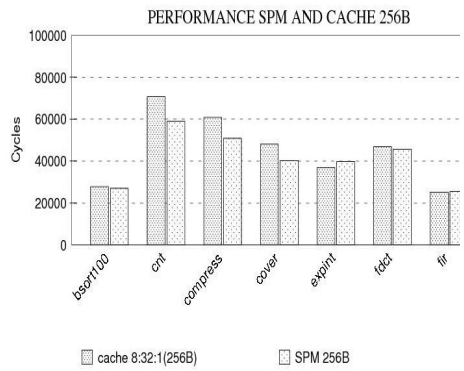

Fig. 7. Performance 256B

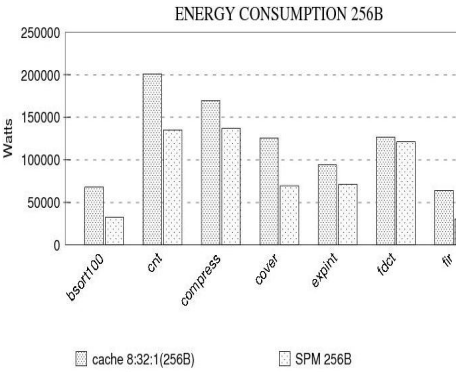

Fig. 8. Energy consumption 256B

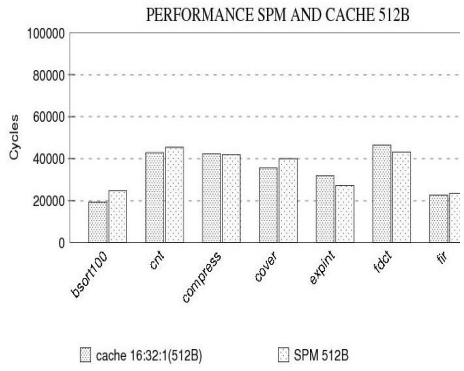

Fig. 9. Performance 512B

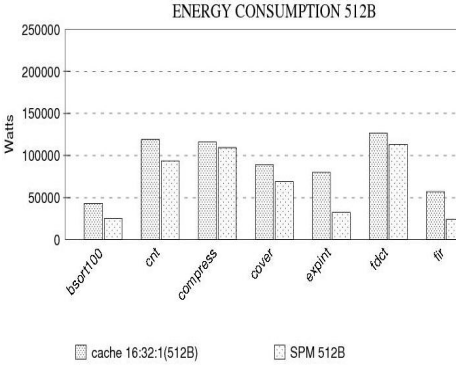

Fig. 10. Energy Consumption 512B

Finally, for SPM and cache sizes of 512B (Figures 9 and 10) we note that with respect to the performance, both of them behave similarly, but the cache shows the best results in five of the eight benchmarks. These differences are not significant showing, in average, a 3% in performance loss for the SPM with respect to the cache. The differences are larger in terms of energy consumption, but in this case the SPM outperform the cache in all benchmarks, presenting an average of 32% improvement with respect to the cache.

In general, we can observe that SPM slightly beats the cache in performance (7.6% on average), but largely reduces the energy consumption by 30.6% on average. It is important to mention that this results would be even better for SPM, if we were used a cache of the same silicon die size than the SPM. For simplicity reasons, we have compared directly both structures with same byte sizes. Many other works in the literature use the more fair comparison over the same die size.

6 Conclusions and Future Work

This paper has presented an original approach to better control the scratch-pad memory in embedded processors in order to reduce energy consumption. The key idea is to reduce as much as possible the overhead resulting from updating the SPM contents at run time. This allows allocating techniques to dynamically adapt the contents of the SPM to the workload execution, maximizing the number of hot spots that may be loaded into SPM.

The technique is orthogonal and complementary to many solutions presented to allocate objects in SPM. Those solutions may benefit and increase the effectiveness adopting the proposed architectural extensions.

The proposed technique has been compared to a instruction cache over a typical workload for embedded systems. On average, compared to a processor with an on-chip instruction cache of the same byte size, our approach improves performance by 7,6% and reduces energy consumption by 30,6%. For certain workloads, our approach has reached an increment of 44% in performance, and a reduction in power around 53%.

This paper has exploited the reductions on energy of the SPM. Future work will focus on the predictable nature of the SPM to exercise our technique in order to obtain better worst case execution times for real-time systems.

Acknowledgments. This research was sponsored by local Government “Generalitat Valenciana” under project GV07/ 2007/122.

References

1. Banakar, R., Steinke, S., Lee, B.-S., Balakrishnan, M., Marwedel, P.: Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In: CODES 2002, pp. 73–78 (2002)
2. Verma, M., Wehmeyer, L., Marwedel, P.: Cache-Aware Scratchpad Allocation Algorithm. In: DATE 2004, pp. 1264–1269 (2004)

3. Verma, M., Marwedel, P.: Advanced memory optimization techniques for low-power embedded processors, pp. I-XII, 1–188. Springer, Heidelberg (2007)
4. Nguyen, N., Dominguez, A., Barua, R.: Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. In: CASES 2005, pp. 115–125 (2005)
5. Egger, B., Kim, C., Jang, C., Nam, Y., Lee, J., Min, S.L.: A dynamic code placement technique for scratchpad memory using postpass optimization. In: CASES 2006, pp. 223–233 (2006)
6. Egger, B., Lee, J., Shin, H.: Scratchpad memory management for portable systems with a memory management unit. In: EMSOFT 2006, pp. 321–330 (2006)
7. Egger, B., Lee, J., Shin, H.: Dynamic scratchpad memory management for code in portable systems with an MMU. ACM Trans. Embedded Comput. Syst. 7(2) (2008)
8. Cho, H., Egger, B., Lee, J., Shin, H.: Dynamic data scratchpad memory management for a memory subsystem with an MMU. In: LCTES 2007, pp. 195–206 (2007)
9. Janapsatya, A., Parameswaran, S., Ignjatovic, A.: Hardware/software managed scratchpad memory for embedded system. In: ICCAD 2004, pp. 370–377 (2004)
10. Balakrishnan, M., Marwedel, P., Wehmeyer, L., Grunwald, N., Banakar, R., Steinke, S.: Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory. In: ISSS 2002, pp. 213–218 (2002)
11. Poletti, F., Marchal, P., Atienza, D., Benini, L., Catthoor, F., Mendias, J.M.: An integrated hardware/software approach for run-time scratchpad management. In: DAC 2004, pp. 238–243 (2004)
12. Li, L., Gao, L., Xue, J.: Memory Coloring: A Compiler Approach for Scratchpad Memory Management. In: IEEE PACT 2005, pp. 329–338 (2005)
13. Lee, L.H., Moyer, B., Arends, J.: Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In: ISLPED 1999, pp. 267–269 (1999)
14. Victorio, J.A., Torres Moren, E.F., Yúfera, V.V.: Vatios: Simulador de Procesador con Estimación de Potencia. XVIII Jornadas de Paralelismo, Zaragoza (2007)
15. Burger, D., Austin, T.M.: The SimpleScalar Tool Set Version 2.0. Technical Report 1342, Computer Sciences Department. University of Wisconsin–Madison (May 1997)
16. Brooks, D., Tiwari, V., Martonosi, M.: Wattch: a framework for architectural-level power analysis and optimizations. In: ISCA 2000, pp. 83–94 (2000)
17. Tarjan, D., Thoziyoor, S., Jouppi, N.: CACTI 4.0, P. HPL-2006- 86 20060606
18. The Mälardalen WCET research group. The Mälardalen WCET benchmarks homepage, `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`
19. Cho, D., Pasricha, S., Issenin, I., Dutt, N.D., Ahn, M., Paek, Y.: Adaptive Scratch Pad Memory Management for Dynamic Behavior of Multimedia Applications. IEEE Trans. on CAD of Integrated Circuits and Systems (TCAD) 28(4), 554–567 (2009)