GENETIC HEURISTICS FOR REDUCING MEMORY ENERGY CONSUMPTION IN EMBEDDED SYSTEMS

Maha Idrissi Aouad
INRIA Nancy, Grand Est / LORIA, 615 Rue du Jardin Botanique, 54600 Villers-Lès-Nancy, France

René Schott
IECN, LORIA, Nancy-Université, Université Henri Poincaré, 54506 Vandoeuvre-Lès-Nancy, France

Olivier Zendra
INRIA Nancy, Grand Est / LORIA, 615 Rue du Jardin Botanique, 54600 Villers-Lès-Nancy, France

Keywords: Energy consumption reduction, Genetic heuristics, Memory allocation management, Optimizations.

Abstract: Nowadays, reducing memory energy has become one of the top priorities of many embedded systems designers. Given the power, cost, performance and real-time advantages of Scratch-Pad Memories (*SPMs*), it is not surprising that SPM is becoming a common form of SRAM in embedded processors today. In this paper, we focus on heuristic methods for SPMs careful management in order to reduce memory energy consumption. We propose Genetic Heuristics for memory management which are, to the best of our knowledge, new original alternatives to the best known existing heuristic (*BEH*). Our Genetic Heuristics outperform BEH. In fact, experimentations performed on our benchmarks show that our Genetic Heuristics consume from 76.23% up to 98.92% less energy than BEH in different configurations. In addition they are easy to implement and do not require list sorting (contrary to BEH).

1 INTRODUCTION

Reducing energy consumption of embedded systems is a topical and very crucial subject. Many systems are energy-constrained and, despite batteries progress, these systems still have a limited autonomy. Numerous systems are concerned: small one and large one. Small systems are mainly our daily life objects, such as: cell phones, laptops, PDAs, MP3 players, etc. For the large systems, we have clusters, mainframes, super computers, etc. These systems are more and more energy greedy. Accordingly, memory will become the major energy consumer in an embedded system. In fact, trends in (ITRS, 2007) show that Systems on Chip (*SoC*) consumption is dominated by dynamic and static memory consumption and that memory will occupy a larger place in SoC.

Thus, different options to save energy, hence increase autonomy, exist. These various approaches can be classified in two main categories: hardware optimizations and software optimizations. Hardware techniques fall beyond the scope of this paper, but a large amount of literature about them is available (see first parts of (Graybill and Melhem, 2002)). In this paper, we will focus on software, compiler-assisted techniques in order to optimize energy consumption in memory.

Most authors rely on Scratch-Pad Memories (*SPMs*) rather than caches. The interested reader can look at (Benini and Micheli, 2000) for a comprehensive list of references. Cache memories, although they help a lot with program speed, do not always fit in embedded systems. In fact, cache memory is random access memory (*RAM*) that a computer microprocessor can access more quickly than it can access regular RAM. As the microprocessor processes data, it looks first in the cache memory and if it finds the data there (from a previous reading of data), it does not have to do the more time-consuming reading of data from larger memory (Tanenbaum, 2005). But caches increase the system size and its energy cost because of cache area plus managing logic. In contrast,

Idrissi Aouad M., Schott R. and Zendra O. (2010).
GENETIC HEURISTICS FOR REDUCING MEMORY ENERGY CONSUMPTION IN EMBEDDED SYSTEMS.
In *Proceedings of the 5th International Conference on Software and Data Technologies*, pages 394-402
DOI: 10.5220/0003002503940402
Copyright © SciTePress

SPMs have interesting features. SPM also known as local store in computer terminology, is a high-speed internal memory used for temporary storage of calculations, data, and other work in progress. It can be considered as similar to an L1 cache in that it is the memory next closest to the ALU's after the internal registers, with explicit instructions to move data from and to main memory. Like cache, therefore, SPM consists of small, fast SRAM, but the main difference is that SPM is directly and explicitly managed at the software level, either by the developer or by the compiler, whereas cache requires extra dedicated circuits. Its software management makes it more predictable as we avoid cache miss cases which is an important feature in real-time embedded systems. Compared to cache, SPM thus has several advantages (Idrissi Aouad and Zendra, 2007). SPM requires up to 40% less energy and 34% less area than cache (Banakar et al., 2002). Further, the run-time with an SPM using a simple static knapsack-based (Banakar et al., 2002) allocation algorithm is 18% better as compared to a cache. Contrarily to (Banakar et al., 2002), (Ben Fradj et al., 2005) distinguish between static and dynamic energy. They also show the effectiveness of using an SPM in a memory architecture where a saving about 35% in energy consumption is achieved when compared to a memory architecture without an SPM. (Absar and Catthoor, 2006) use statistical methods and the Independent Reference Model (*IRM*) to prove that SPMs, with an optimal mapping based on access probabilities, will always outperform the direct-mapped cache, irrespective of the layout influencing the cache behavior. Additionally, SPM cost is lower.

The rest of the paper is organized as follows. Section 2 describes some existing heuristics and related works for managing memory data allocation. Section 3 presents our approach based on Genetic Heuristics to find the optimized memory data allocation in order to reduce energy consumption. Section 4 gives the memory energy consumption model we used in order to estimate the energy consumed by our different heuristics. Section 5 shows the various experimental results obtained. Finally, Section 6 concludes and gives some perspectives.

2 EXISTING HEURISTICS AND RELATED WORKS

The approaches presented in this section try to help in determining the optimized memory allocation in order to reduce energy consumption according to memory type (access speed, energy cost, large number of miss access cases, etc.) and application behavior. In order to do so, these methods use profile data to gather memory access frequency information. This information can be collected either statistically by analyzing the source code of the application or dynamically by profiling the application (number of times that data is accessed, data size, access frequency, etc.).

In these techniques, because of the reduced size of SRAM, one tries to optimally allocate data in it in order to realize energy savings. An approach is to place interesting data in memory with a low access cost (*SPM*) whereas the other data will be placed in a memory with low storage cost (*DRAM*). Thus, most of the authors use one of the three following strategies:

Allocate Data into SPM by Size. The smaller data are allocated into SPM as there is space available else they are allocated in DRAM. This method has the advantage of being simple to implement since it only considers the size of the data but has the disadvantage of allocating the largest data in the main memory (*DRAM*). These largest data could be often accessed, which will imply a very few energy economy.

Allocate Data into SPM by Number of Accesses. The most frequently accessed/used data are allocated into SPM as there is space available else they are allocated in DRAM. This strategy is optimal than the previous one, since the most frequently accessed/used data will be allocated in a memory that consumes less energy and therefore will achieve more savings as explained and demonstrated in (Sjödin et al., 1998) and (Steinke et al., 2002). However, we can note granularity problems in some cases such as a structure which only one part is often accessed/used.

Allocate Data Memory into SPM by Number of Accesses and Size (*BEH*). This is somehow a combination of the two previous strategies. The idea here is to combine their advantages. If we consider the example of a structure in which only a part is the most frequently accessed/used, we take into account the average number of access to this structure. This avoids granularity problems. Here, data are sorted according to their ratio (access number/size) in descending order. The data with the highest ratio is allocated first into SPM as there is space available. Otherwise it is allocated in DRAM. This heuristic uses a sorting method which can be computationally expensive for a large amount of data. In addition to that, this sorting method will not work very well in a dynamic perspective where the maximum capacity of the SPM is not known in advance. This is, so far, the best known existing heuristic (*BEH*).

In these techniques, most authors model the problem as a 0/1 integer linear programming (*ILP*) problem and then use an available IP solver to solve it. The main trade-offs between these approaches revolve around the objects considered (arrays, loops, global, heap or stack variables, etc.). In (Avissar et al., 2002), because of the reduced SRAM size, the least used variables are first allocated to slower memory banks (*DRAM*), while the most frequently used variables are kept in fast memory (*SRAM*) as much as possible. They consider global and stack variables and choose between SPM and cache, while (Steinke et al., 2002; Wehmeyer et al., 2004) consider global variables, functions and basic blocks and choose between SPM banks. When (Udayakumaran et al., 2002) consider static and global variables and choose between SPM banks also. The previous techniques are all based on the frequency of data accesses. In contrast, another approach is to focus on data that is the most cache-conflict prone (Panda et al., 1997), (Truong et al., 1998). In the rest of this paper, we will refer to the strategy BEH as a basis for our memory energy optimizations.

3 OUR GENETIC APPROACH

Genetic algorithms (*GAs*) are adaptive methods which may be used to solve search and optimization problems. They are based on the genetic processes of biological organisms (Sivanandam and Deepa, 2007).

By starting with a population of possible solutions and changing them during several iterations, GAs hope to converge to the fittest solution. Each solution is represented through a chromosome, which is just an abstract representation. The process begins with a set of potential solutions or chromosomes that are randomly generated or selected. Over many generations, natural populations evolve according to the principles of natural selection and survival of the fittest. For generating new chromosomes, GA can use both crossover and mutation techniques. Crossover involves splitting two chromosomes and then, for example, combining one half of each chromosome with the other pair. Mutation involves flipping a single bit of a chromosome. The chromosomes are then evaluated using a certain fitness criterion and the ones which satisfy the most this criterion are kept while the others are discarded. This process repeats until the population converges toward the optimal solution. The basic genetic algorithm is summarized in Figure 1.

There are several advantages to the Genetic Algorithm such as their parallelism and their liability. They require no knowledge or gradient information

```
SELECT random population of n chromosomes.
EVALUATE the fitness f(x) of each chromosome
x in the population.
LOOP
 SELECT two parent chromosomes from a
 population.
 CROSS OVER the parents to form new children
 with a crossover probability Pc.
 MUTATE new children with a mutation
 probability Pm.
 Place new offspring in the new population.
 Use new generated population for a further
 sum of the algorithm.
 EXIT if the end condition is satisfied and
 return best solution.
END LOOP
```

Figure 1: A basic genetic algorithm.

about the response surface, they are resistant to becoming trapped in local optima and they perform very well for large-scale optimization problems. GAs have been used as heuristics to solve difficult problems (such as NP-hard problems) for machine learning and also for evolving simple programs. Applications of Genetic Algorithms include: nonlinear programming, stochastic programming, signal processing and combinatorial optimization problems such as the Traveling Salesman Problem, Knapsack Problem, sequence scheduling, graph coloring (just to name a few). For further applications, the interested reader can see (Zheng and Kiyooka, 1999).

3.1 Our Optimization Problem

Our problem is a combinatorial optimization problem like the famous knapsack problem (H. Kellerer and Pisinger, 2004). Suppose memory is a big knapsack and data are items. We want to fill this knapsack that can hold a total weight of W with some combination of items from a list of N possible items each with weight w_i and value v_i so that the value of the items packed into the knapsack is maximized. This problem has a single linear constraint, a linear objective function which sums the values of the items in the knapsack, and the added restriction that each item will be in the knapsack or not.

If N is the total number of items, then there are 2^N subsets of the item collection. So an exhaustive search for a solution to this problem generally takes exponential running time. Therefore, the obvious brute force approach is infeasible. Some dynamic programming techniques also have exponential running time, but have proven useful in practice. Here, we take a different approach, and investigate the problem using genetic methods. In this paper, we propose a heuristic based on GAs to solve the problem of optimizing the memory data allocation in order to reduce memory energy consumption. This heuristic outperforms the best known existing heuristic (*BEH*) presented in Section 2 and the Tabu Search heuristic presented in (Idrissi Aouad et al., 2010).

3.2 Key Elements

In this section, we explain some basic terminologies and operators we used with our Genetic Heuristics.

- **Individual.** An individual is a single solution.
- **Population.** A population is a collection of individuals. The two important aspects of population used in GAs are the initial population generation and the population size. For each problem, the population size will depend on the complexity of the problem. Ideally, the initial population should have a gene pool as large as possible in order to be able to explore the whole search space. Hence, to achieve this, the initial population is, in most of the cases, chosen randomly. In our case, we randomly selected the initial population.
- **Encoding.** If N is the total number of data, then a solution point is just a finite sequence s of N terms such that $s[n]$ is either 0 or the size of the n_{th} data. $s[n] = 0$ if and only if the n_{th} data is not selected in the solution point. This solution must satisfy the constraint of not exceeding the maximum SPM capacity (i.e. $\sum_{i=1}^{N} s[i] \leq C$).
- **Fitness.** The fitness of an individual is the value of an objective function. The fitness not only indicates how good the solution is, but also corresponds to how close the chromosome is to the optimal one. In our Genetic Heuristic, at each stage (*generation*), the solution points are evaluated for fitness (according to how much of the memory capacity they fill), and the best and worst performers are identified.
- **Crossover.** Crossover is the process of taking two parent solutions and producing from them a child. After the selection process, the population is enriched with better individuals. Here, when conditions are ripe for breeding, the best solution mates with a random (non-extreme) solution, and the offspring replaces the worst one. The child inherits a part of each parent's genes. In our Genetic Heuristic, we used the following crossover techniques and probability:
  - **Single Point Crossover.** The two mating chromosomes are cut once at a randomly selected crossover point and the sections after the cuts are exchanged.
  - **Two Points Crossover.** In two points crossover, two crossover points are chosen randomly and the contents between these points are exchanged between two mated parents.
  - **Crossover Probability.** Crossover Probability (P_c) is a parameter to describe how often crossover will be performed. If there is no crossover, offspring are exact copies of parents. If $P_c = 1$, then all offspring are made by crossover. If $P_c = 0$, whole new generation is made from exact copies of chromosomes from old population. Crossover is made in hope that new chromosomes will contain good parts of old chromosomes and therefore the new chromosome will be better.
- **Mutation.** Mutation prevents the algorithm to be trapped in local optima. It is seen as a background operator to maintain genetic diversity in the population by varying the gene pool. The genes of a solution are just its sequence terms. In our Genetic Heuristic, a mutation of a solution is a random change of up to half of its current genes. A gene change sets a term to 0 if the term is currently nonzero, and sets a term to the corresponding data size if the term is currently zero.
  - **Mutation Probability.** The mutation probability (P_m) decides how often parts of chromosome will be mutated. If there is no mutation, offspring are generated immediately after crossover without any change. If $P_m = 1$, whole chromosome is changed, if $P_m = 0$, nothing is changed. Mutation should not occur very often, because then GA will in fact change to random search.
- **Search Termination.** The termination or convergence criterion brings the search to a halt. There are various stopping conditions: the maximum number of generations reached, after a specified time has elapsed, if there is no change to the population's best fitness for a specified number of generations or during an interval of time, etc. Here, we stop the Genetic Heuristic after a certain amount of time.

4 MEMORY ENERGY ESTIMATION MODEL

In order to compute the energy cost of the system for each configuration, we propose in this section an energy consumption estimation model for our considered memory architecture composed by an SPM, a DRAM and an instruction cache memory. In our model, we distinguish between the two cache write policies: write-through and write-back. In a Write-Through cache (*WT*), every write to the cache causes a synchronous write to the DRAM. Alternatively, in a Write-Back cache (*WB*), writes are not immediately mirrored to the main memory. Instead, the cache tracks which of its locations have been written over and then, it marks these locations as dirty. The data in these locations is written back to the DRAM when those data are evicted from the cache (Tanenbaum, 2005). Our proposed energy consumption estimation model is presented below:

$$
\begin{aligned}
E &= E_{tspm} + E_{tic} + E_{tdram} \\
E &= N_{spmr} * E_{spmr} && (1) \\
&+ N_{spmw} * E_{spmw} && (2) \\
&+ \sum_{k=1}^{N_{icr}} [h_{i_k} * E_{icr} + (1 - h_{i_k}) * [E_{dramr} + E_{icw} \\
&\quad +(1 - WP_i) * DB_{i_k} * (E_{icr} + E_{dramw})]] && (3) \\
&+ \sum_{k=1}^{N_{icw}} [WP_i * E_{dramw} + h_{i_k} * E_{icw} + (1 - WP_i) * \\
&\quad (1 - h_{i_k}) * [E_{icw} + DB_{i_k} * (E_{icr} + E_{dramw})]] && (4) \\
&+ N_{dramr} * E_{dramr} && (5) \\
&+ N_{dramw} * E_{dramw} && (6)
\end{aligned}
$$

Lines (1) and (2) represent respectively the total energy consumed during a reading and during a writing from/into SPM. Lines (3) and (4) represent respectively the total energy consumed during a reading and during a writing from/into instruction cache. When, lines (5) and (6) represent respectively the total energy consumed during a reading and during a writing from/into DRAM. The various terms used in our energy consumption estimation model are explained in Table 1.

5 EXPERIMENTAL RESULTS

For our experiments, we consider a memory architecture composed by a Scratch-Pad Memory, a main memory (*DRAM*) and an instruction cache memory. We make sure to take similar features for the cache memory and the SPM in order to compare their energy performance fairly. We performed experiments with eleven benchmarks from six different suites: MiBench (Guthaus et al., 2001), SNU-RT, Mälardalen, Mediabenchs, Spec 2000 and Wcet Benchs. Table 2 gives a description of these benchmarks.

Table 1: List of terms.

Term	Meaning
E_{tspm}	Total energy consumed in SPM.
E_{tic}	Total energy consumed in instruction cache.
E_{tdram}	Total energy consumed in DRAM.
E_{spmr}	Energy consumed during a reading from SPM.
E_{spmw}	Energy consumed during a writing into SPM.
N_{spmr}	Reading access number to SPM.
N_{spmw}	Writing access number to SPM.
E_{icr}	Energy consumed during a reading from instruction cache.
E_{icw}	Energy consumed during a writing into instruction cache.
N_{icr}	Reading access number to instruction cache.
N_{icw}	Writing access number to instruction cache.
E_{dramr}	Energy consumed during a reading from DRAM.
E_{dramw}	Energy consumed during a writing into DRAM.
N_{dramr}	Reading access number to DRAM.
N_{dramw}	Writing access number to DRAM.
WP_i	The considered cache write policy: WT or WB. In case of WT, $WP_i = 1$ else in case of WB then $WP_i = 0$.
DB_{i_k}	Dirty Bit used in case of WB to indicate during the access k if the instruction cache line has been modified before ($DB_i = 1$) or not ($DB_i = 0$).
h_{i_k}	Type of the access k to the instruction cache. In case of cache hit, $h_{i_k} = 1$. In case of cache miss, $h_{i_k} = 0$.

In order to compute the energy cost of the system for each configuration, we used our developed energy consumption estimation model presented in Section 4. This model is based on the OTAWA framework (Cassé and Rochange, 2007) to collect information about number of accesses and on the energy consumption estimation tool CACTI (Wilton and Jouppi, 1996) in order to collect information about energy per access to each kind of memory. OTAWA (Open Tool for Adaptative WCET Analysis) is a framework of C++ classes dedicated to static analyses of programs in machine code and to the computation of Worst Case Execution Time (*WCET*). OTAWA is freely available (under the LGPL license) and is designed to support different architectures like PowerPC, ARM or M68HC. In our case, we focus on

Table 2: List of benchmarks.

Benchmark	Suite	Description
Sha	MiBench	The secure hash algorithm that produces a 160-bit message digest for a given input.
Bitcount	MiBench	Tests the bit manipulation abilities of a processor by counting the number of bits in an array of integers.
Fir	SNU-RT	Finite impulse response filter (signal processing algorithms) over a 700 items long sample.
Jfdctint	SNU-RT	Discrete-cosine transformation on 8x8 pixel block.
Adpcm	Mälardalen	Adaptive pulse code modulation algorithm.
Cnt	Mälardalen	Counts non-negative numbers in a matrix.
Compress	Mälardalen	Data compression using lzw.
Djpeg	Mediabenchs	JPEG decoding.
Gzip	Spec 2000	Compression.
Nsichneu	Wcet Benchs	Simulate an extended Petri net. Automatically generated code with more than 250 if-statements.
Statemate	Wcet Benchs	Automatically generated code.

PowerPC architectures. In our model, we distinguish between the two cache write policies: Write-Through (*WT*) and Write-Back (*WB*) as explained in Section 4. Our presented Tabu Search algorithm and the BEH strategy have been implemented with the C language on a PC Intel Core 2 Duo, with a 2.66 GHz processor and 3 Gbytes of memory running under Mandriva Linux 2008.

5.1 Genetic Heuristics with Both Crossover and Mutation

In this subsection, we investigate the performances of our Genetic Heuristics using both genetic operators: crossover and mutation. These genetic operators are explained in Section 3.2. After making experiments on the standard benchmarks presented in Table 2, we find that our Genetic Heuristics achieve the same energy savings as BEH. This is what we were expecting, due to the fact that they both give the optimal solution thanks to our developed backtracking algorithm. This is true for the standard benchmarks we use as they contain uniform data leading to a big number of local minima. Thus, in order to put some trouble in the BEH strategy and see if it still gives the best solution, we decided to modify slightly our benchmarks. Concretely, our modification consists in adding one variable to each benchmark. This variable is very common to all benchmarks as it performs an output (so did not change the benchmarks features) and is big enough to provide energy savings if it is chosen for an SPM allocation. We refer to a modified benchmark as benchmarkCE.

In these experiments, we generate 30 different executions for each of our Genetic Heuristics as the solution given differs from an execution to another. *Genetic 1* represents the results obtained for $P_m = 0.1$, $P_c = 0.5$ and for a single point crossover. *Genetic 2* represents the results obtained for $P_m = 0.1$, $P_c = 0.5$ and for a two points crossover. When *Genetic (1,2)* refers to the average results obtained on 30 executions of *Genetic 1* and 30 executions of *Genetic 2*. We mix them that way, as the results obtained with *Genetic 1* are equal to those obtained with *Genetic 2*.

Figure 2 presents the results obtained when comparing BEH and our Genetic Heuristics on our modified benchmarks assuming the write-back cache policy. In the following, as the shapes of curves obtained when comparing BEH and Genetic Heuristics on the modified benchmarks assuming the Write-Through cache policy (WT) or the Write-Back cache policy (WB) are slightly the same, only the results obtained with the write-back cache policy are plotted. One can show that $E_{WTmode} \neq E_{WBmode}$.

As we can see from this figure, *Genetic (1,2)* achieves better performances than BEH on energy savings on our benchmarks. In fact, these results show that *Genetic (1,2)* consumes from 78.18% (StatemateCE) up to 98.92% (ShaCE) less energy than BEH in the WT mode on one hand. On the other hand, *Genetic (1,2)* consumes from 76.23% (StatemateCE) up to 98.92% (ShaCE) less energy than BEH in the WB mode.

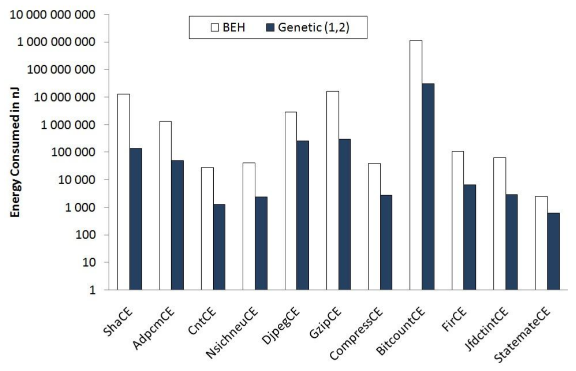

Figure 2: Energy consumed by modified benchmarks with WB mode.

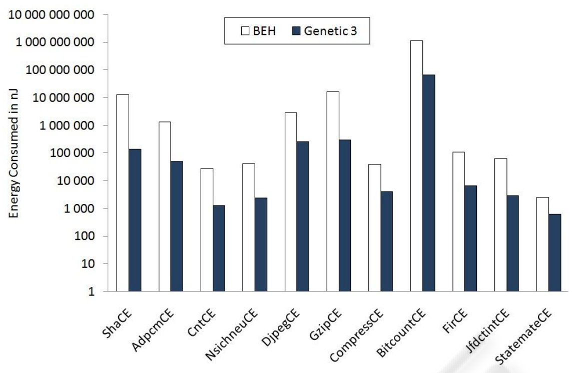

Figure 3: Energy consumed by modified benchmarks with WB mode without crossover.

For BEH, although we used the modified benchmarks, we still obtain the same energy savings as with the standard benchmarks. The BEH strategy did not give the optimal solution anymore as one could expect as proven by our developed backtracking algorithm, but *Genetic (1,2)* gives it. This is normal due to the fact that BEH is a sort of access number/size of data as we explain in Section 2. The variable we add in each benchmark has a given access number/size (this ratio depends on the data profiling of each benchmark) so that this variable is not a priority in the sorting made by the BEH method. This is done on purpose so that when it will be the turn of this variable to be treated by BEH, the remaining space in the SPM will not be enough to take this variable and hence it will be allocated in main memory. Whereas, an optimal solution would be to start by allocating this variable first into the SPM. In contrast, our Genetic Heuristics are robust enough to overcome this problem and find the optimal solution as it can be noticed.

5.2 Genetic Heuristics with either Crossover or Mutation

In this subsection, we look if removing mutation or crossover operator has a real impact on the performances of our Genetic Heuristics. Curious to know more, we decided to make some additional experiments. First, we start by omitting the crossover operation. In these experiments, we generate 30 different executions for our Genetic Heuristic as the solution given differs from an execution to another. These results are average results obtained on 30 executions of *Genetic 3*. Where: *Genetic 3* represents the results obtained for $P_m = 0.5$. Figure 3 presents the results obtained when comparing our Genetic Heuristic to BEH on our modified benchmarks assuming the write-back cache policy and without crossover. Reminding that $E_{WTmode} \neq E_{WBmode}$.

As we can see from this figure, *Genetic 3* achieves better performances than BEH on energy savings on our benchmarks. In fact, these results show that *Genetic 3* consumes from 78.18% (StatemateCE) up to 98.92% (ShaCE) less energy than BEH in the WT mode on one hand without crossover. On the other hand, *Genetic 3* consumes from 76.23% (StatemateCE) up to 98.92% (ShaCE) less energy than BEH in the WB mode without crossover.

Then, we continue by omitting the mutation operation. In these experiments, we generate 30 different executions for each of our Genetic Heuristics as the solution given differs from an execution to another. *Genetic 4* represents the results obtained for a single point crossover. *Genetic 5* represents the results obtained for a two points crossover. These results are average results obtained on 30 executions of *Genetic 4* and 30 executions of *Genetic 5*. Figure 4 shows the results obtained when comparing our Genetic Heuristics to BEH on our modified benchmarks assuming the write-back cache policy and without mutation. Reminding that $E_{WTmode} \neq E_{WBmode}$.

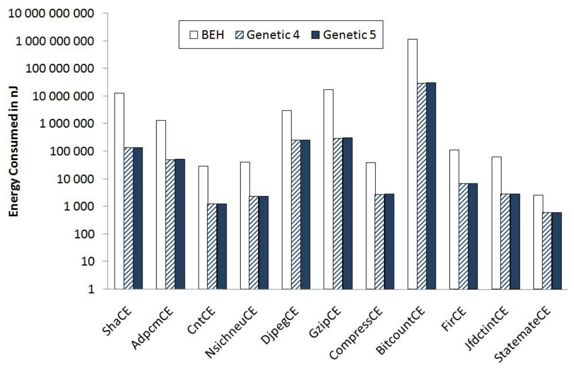

Figure 4: Energy consumed by modified benchmarks with WB mode without mutation.

As we can see from Figure 4, both *Genetic 4* and *Genetic 5* achieve better performances than BEH on energy savings on our benchmarks. In fact, these results show that both *Genetic 4* and *Genetic 5* consume from 78.18% (StatemateCE) up to 98.92% (ShaCE) less energy than BEH in the WT mode on one hand without mutation. On the other hand, both *Genetic 4* and *Genetic 5* consume from 76.23% (StatemateCE) up to 98.92% (ShaCE) less energy than BEH in the WB mode without mutation.

In some cases, it is not necessary to apply both genetic operators (crossover and mutation) to achieve good results. Thus, omitting one of the two genetic operators still allows GAs to converge. In fact, we can see that our Genetic Heuristics outperforms BEH on the modified benchmarks. Regardless if we are using either crossover or mutation operator or both of them, we achieve the same energy savings on our modified benchmarks.

6 CONCLUDING REMARKS AND FURTHER RESEARCH ASPECTS

In this paper, we have proposed a general energy consumption estimation model able to be adapted to different memory architecture configurations. We also have proposed new Genetic Heuristics for reducing memory energy consumption in embedded systems which are more efficient than the best known existing method (*BEH*). In fact, our Genetic Heuristics manage to consume nearly from 76% up to 98% less memory energy than BEH in different memory configurations. In addition our Genetic Heuristics are easy to implement and do not require list sorting (contrary to BEH). Comparisons of execution times of BEH and Genetic Heuristics will be included in the full version of this paper. In future work, we plan to explore hybrid heuristics and other evolutionary heuristics (Markov Decision Processes, Simulated Annealing, ANT method, Particle Swarm technique, etc.) for solving the problem of reducing memory energy consumption.

ACKNOWLEDGEMENTS

The authors are grateful to anonymous referees for their comments and suggestions.

This work is financed by the french national research agency (ANR) in the Future Architectures program.

REFERENCES

Absar, J. and Catthoor, F. (2006). Analysis of scratch-pad and data-cache performance using statistical methods. In *ASP-DAC*, pages 820–825.

Avissar, O., Barua, R., and Stewart, D. (2002). An optimal memory allocation scheme for scratch-pad-based embedded systems. *Transaction. on Embedded Computing Systems.*, 1(1):6–26.

Banakar, R., Steinke, S., Lee, B., Balakrishnan, M., and Marwedel, P. (2002). Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES*, pages 73–78, New York, NY, USA. ACM Press.

Ben Fradj, H., Ouardighi, A. E., Belleudy, C., and Auguin, M. (2005). Energy aware memory architecture configuration. In *MEDEA '04: Proceedings of the 2004 workshop on MEmory performance*, volume 33, pages 3–9. ACM.

Benini, L. and Micheli, G. D. (2000). System-level power optimization: techniques and tools. *IEEE Design and Test*, 17(2):74–85.

Cassé, H. and Rochange, C. (2007). OTAWA, Open Tool for Adaptative WCET Analysis. In *Design, Automation and Test in Europe (Poster session "University Booth") (DATE), Nice, 17/04/07-19/04/07*, page (electronic medium), http://www.date-conference.com/. DATE. Poster session.

Graybill, R. and Melhem, R. (2002). *Power aware computing*. Kluwer Academic Publishers, Norwell, MA, USA.

Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. (2001). Mibench: A free, commercially representative embedded benchmark suite. In *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA. IEEE Computer Society.

H. Kellerer, U. P. and Pisinger, D. (2004). *Knapsack Problems*. Springer, Berlin, Germany.

Idrissi Aouad, M., Schott, R., and Zendra, O. (2010). A Tabu Search Heuristic for Scratch-Pad Memory Management. In *ICSET'2010: Proceedings of International Conference on Software Engineering and Technology*. To appear, in press.

Idrissi Aouad, M. and Zendra, O. (2007). A Survey of Scratch-Pad Memory Management Techniques for low-power and -energy. In *2nd ECOOP Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2007)*.

ITRS (2007). System drivers. http://www.itrs.net/Links/2007ITRS/2007_Chapters/2007_SystemDrivers.pdf.

Panda, P. R., Dutt, N., and Nicolau, A. (1997). Efficient utilization of scratch-pad memory in embedded processor applications. In *DATE*.

Sivanandam, S. N. and Deepa, S. N. (2007). *Introduction to Genetic Algorithms*. Springer Publishing Company, Incorporated.

Sjödin, J., Fröderberg, B., and Lindgren, T. (1998). Allocation of global data objects in on-chip ram. In *Workshop on Compiler and Architectural Support for Embedded Computer Systems*. ACM.

Steinke, S., Wehmeyer, L., Lee, B., and Marwedel, P. (2002). Assigning program and data objects to scratchpad for energy reduction. In *DATE*, page 409. IEEE Computer Society.

Tanenbaum, A. (2005). *Architecture de l'ordinateur 5e édition*. Pearson Education.

Truong, D. N., Bodin, F., and Seznec, A. (1998). Improving cache behavior of dynamically allocated data structures. In *International Conference on Parallel Architectures and Compilation Techniques (IEEE PACT)*, pages 322–329.

Udayakumaran, S., Narahari, B., and Simha, R. (2002). Application specific memory partitioning for low power. In *ACM COLP 2002 (Compiler and Operating Systems for Low Power)*. ACM Press.

Wehmeyer, L., Helmig, U., and Marwedel, P. (2004). Compiler-optimized usage of partitioned memories. In *WMPI*.

Wilton, S. and Jouppi, N. (1996). Cacti: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*.

Zheng, Y. and Kiyooka, S. (1999). Genetic algorithm applications. http://www.me.uvic.ca/∼zdong/courses/mech620/GA_App.PDF.

SciTePress
Science and Technology Publications