Iterative Compilation for Energy Reduction

Stefan Valentin Gheorghita, Henk Corporaal, and Twan Basten*

***Abstract*— The rapidly increasing number of architectural changes in embedded processors puts compiler technology under an enormous stress. This is emphasized by new demands on compilers, like requirements to reduce static code size, energy consumption or power dissipation. Iterative compilation has been proposed as an approach to find the best sequence of optimizations (such as loop transformations) for an application, in order to improve its performance. In this paper, we study both the effect of several loop transformations on energy consumption as well as the possibility of using the iterative compilation method in order to find the best compiled code for energy. From analyzed benchmarks, we conclude that in most cases the decrease in energy consumption is coming together with performance improvement. However, the best compiled code for performance is not always the best one for energy. Thus, a combined energy-performance factor may be considered to evaluate the compiled code in order to have good compromises. As iterative compilation has been proven to be a good approach to compilation for performance, we may conclude that it is also promising for the compilation for energy problem.**

***Index Terms*— Iterative Compilation, Program Optimization, Energy Consumption, Loop Transformations**

I. Introduction

EACH YEAR, computing technology starts being used in new areas. The number of computers in use is growing rapidly and the increasing demand for greater performance in all areas of computing leads to an exponential growth of hardware performance. Due to increasing system complexity, hardware diversity may also increase. The variety of processors, with different instruction sets and different numbers of functional units and memory hierarchies and sizes, is increasing, especially in the embedded systems area where backward compatibility is often sacrificed for performance.

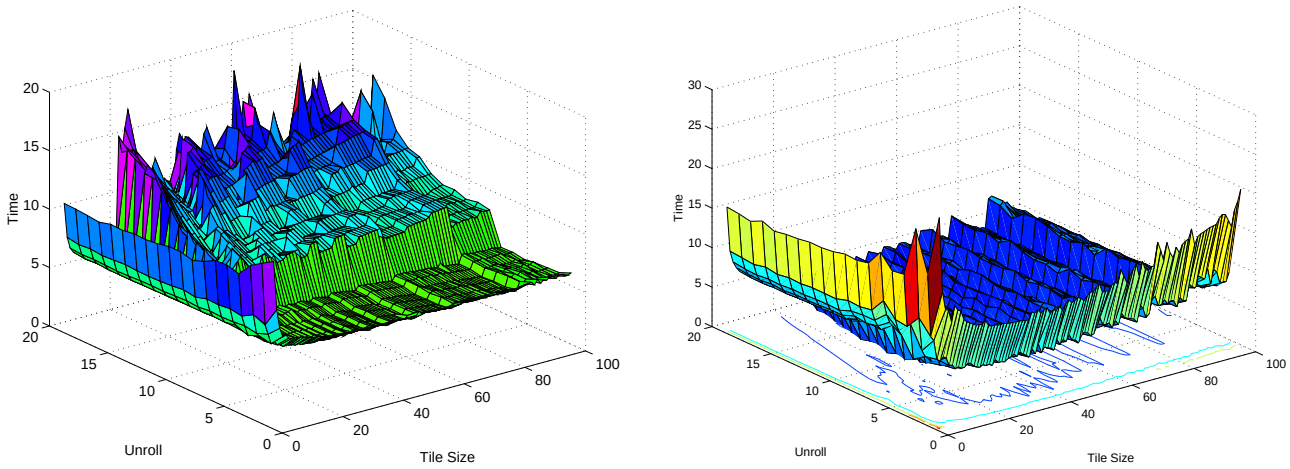
The rapid range of architectural changes puts compiler technology under an enormous stress. At the same time, besides performance, new demands are added to compilers, in order to optimize applications for different criteria, like static code size, energy consumption or power dissipation. The size of compiled code is important because of the limited size of embedded-systems memory, and also because of the importance of transferring applications over the internet. The rise of mobile embedded systems (like mobile phones, PDAs, digital cameras) directs the interest of compilers also to reducing the energy consumption and power dissipation during application execution.

In order to improve application performance, compilers try to aggressively analyze and optimize programs. Over time, it became clear that efficient code generation requires source-to-source restructuring, for example, to enable vectorization or parallelization, to exploit cache hierarchy or to reduce power. Most of these transformations do loop restructuring. Although a large number of transformations have been recognized as being potentially beneficent [20], it is not easy to find which transformations should be used for a given application, in which order to apply them, and to which code sections. This longstanding open problem is called the *phase-order problem*. In traditional compilers, this problem is approached by hard coding a sequence of transformations [20]. This sequence is largely based on observed behavior for a small number of benchmarks. The values of transformation parameters are either fixed or computed using a simplified machine model. In some circumstances, profiling is employed. However, because of the complexity of modern processors and their associated back-end compilers, this approach has many difficulties in delivering optimal code. Moreover, the transformations are highly interdependent and the effect of several compound transformations is extremely difficult to predict. For example, for two transformations, tiling [10, 17] and loop unrolling [7], Fig. 1 (from [15]) shows the execution time of Matrix-Matrix Multiplication for several tile sizes and unroll factors on two different platforms. It can be observed that a small deviation from good tile sizes or unroll factors can cause a huge increase in the execution time and even a slowing down compared to the original program. Also, the effect of tiling and unrolling for the two different platforms differs widely. This figure suggests that it is difficult to find an analytical model that will correctly predict execution times, since this model has to predict these highly irregular graphs.

As an approach to solving this problem, in [15, 16], Knijnenburg, Kisuki and O'Boyle have proposed the concept of *iterative compilation*, where many variants of the source program are generated and the best one is selected by actually profiling these variants on the target hardware. In theory, this approach can find the optimal version of the program by simply considering all the possibilities. However, in practice, the search space is extremely large and the execution of transformed versions of the source program is extremely time consuming, particularly if the number of transformations taken into account grows. As a solution, the authors have shown that by evaluating a small percentage of the transformation space, almost randomly selected, their approach can find excellent optimizations across a range of architectures, outperforming static techniques significantly. This approach is highly attractive in situations that require high performance, such as embedded systems, in which the compilation time can be amortized across the number of products shipped, or scientific code that is run many times, like weather prediction models, or in the case of vendor supplied library routines.

This paper extends the literature on iterative compilation, considering the following two problems: (i) the effect of several important loop transformations on energy consumption

*All authors are with Eindhoven University of Technology, PO Box 513, 5600 MB, Eindhoven, The Netherlands. Contact: s.v.gheorghita@tue.nl



(a) Pentium II (b) UltraSparc I

Fig. 1. Execution Time of Matrix-Matrix Multiplication for Different Unroll Factors and Tile Sizes

and (ii) the possibility of using iterative compilation for finding the best compiled code for energy. Furthermore, we are interested in answering the question whether an improvement in performance always implies a reduction in energy consumption. Our results show that this is often, but, maybe surprisingly, not always the case.

The paper is organized as follows. Section II presents related work in non-static compilation techniques and in compilation for low energy. Section III presents an analysis of the effects of five important loop transformations on energy consumption. The experimental environment and the benchmarks that are used are presented in Section IV. The last two sections, V and VI, present the results, their evaluation, and the conclusions.

II. Related Work

Besides the iterative compilation approach introduced in [15, 16], several non-static ways to improve the performance of compiled code have been proposed. For example, [1, 9, 23] use profiling and runtime information to find which transformations are best for a given application and platform. Besides these attempts to use search algorithms in optimizations, there are others including Massalin's Superoptimizer [18] and Keith Cooper's adaptive compiler [11]. Superoptimizer tries to find the optimal instruction selection using an exhaustive search. This technique produces good results, but it is too time expensive. Granlund and Kenner adapted Massalin's ideas and created a faster Superoptimizer that generates code compatible with gcc assembly [13]. Keith Cooper's adaptive compiler uses genetic algorithms and in all tested cases it improved execution time up to 20% and it reduced code size up to 13%.

Reduction of energy consumption by generating better code has been a research issue for the last ten years [21]. Besides the work that has been done looking for new optimizations to reduce energy consumption, several approaches tried to analyze the impact of source code transformations on energy. Madhavi Valluri and Lizy John [22] studied the effect of compiler optimizations on power dissipation and energy, using a group of benchmarks compiled with gcc, having only one compiler optimization enabled. In [14], the authors present an experimental evaluation of several state-of-the-art compiler optimizations on energy consumption, considering both the processor core and the memory system. These two papers analyze the energy effect for an isolated loop transformation using the compiler default values for its parameters. In contrast, our work is concerned with the combined effect on energy consumption of a sequence of loop transformations considering a large set of values for their parameters.

III. Loop Transformation Effects on Energy Consumption

Loop transformations are source-to-source restructuring transformations that modify either the body or the control structure of loops, in order to change the iteration space of a loop nest. These changes may improve instruction and data cache performance, instruction level parallelism (ILP), and iteration level parallelism. They are very effective because a gain inside a loop iteration may result in a big improvement considering the number of iterations. While loop transformations often aim to reduce the number of cycles needed to run an application, they might not decrease consumed energy as well, as they can create very complex loop bounds and array subscript expressions that may increase the energy used by the processor for computation. The expected effects of five of these transformations on system energy are analyzed in the following paragraphs. The chosen transformations, loop unrolling, unroll-and-jam, tiling, loop fission and loop fusion, are the most-used transformations in state-of-the-art compilers, like gcc.

In [3], the authors identify three major constituents consuming significant system energy: (i) computation units, (ii) memory units, and (iii) inter-processor communication units. As we analyze the effects of loop transformations on a single processor architecture, only the first two **energy components** are interesting for us. Computation energy includes all the operations done on the processor, except the energy used by the registers. In contrast with many papers that present large

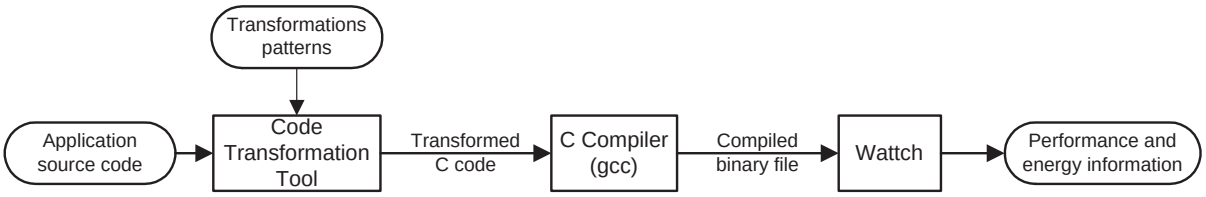


Fig. 2. Compilation Framework

memory energy reductions without looking at register energy [19], we consider the registers together with the caches as part of the memory system because we want to make a fair analysis of the division between the energy components. For example, a loop transformation can change the source code in such a way that data can fit in registers, instead of the cache. If the registers are considered to be part of the computation, it would result in an increase of computation energy and a reduction of memory system energy, which would not be correct.

Loop unrolling [20] replicates the loop body a number of times u (= unroll factor) and multiplies the iteration step with u. This transformation is used in particular to improve instruction level parallelism, exposing more instructions to the hardware at a single point in time. Even though it reduces computation energy due to less loop control structures and array offset computations and a better instruction scheduling, it may increase computation energy because of the newly added instructions (e.g. array subscript expressions). Temporal data locality may be improved due to the freedom offered to the scheduler, allowing the use of registers instead of the data cache, reducing memory system energy. If the chosen unroll factor is too big, more instruction cache misses are produced and also the *loop buffer* [2], if present in the processor, becomes less efficient. So temporal instruction locality decreases, and memory system energy increases.

Most compilers unroll the innermost loop of a nesting. Straightforward outer loop unrolling is not very efficient because it replicates the instances of inner loops. To avoid the additional loop control overhead when outer loops are unrolled, a compiler may fuse the copies of inner loops back together, generating the same loop nest as in the original code, but with a larger inner loop body and a smaller number of outer loop iterations. This sequence of transformations is called **unroll-and-jam** [7] and its effect on energy consumption does not differ from the ordinary loop unrolling transformation.

Loop tiling [10, 17] (or blocking) divides the loop iteration space in small tiles and transforms the loop nest to iterate over them. It improves the temporal data locality of the program, reducing the number of data cache misses together with the memory system energy. Computation energy is also decreased as a smaller number of cycles is lost for solving data cache misses[1]. On the contrary, the newly added loop control structures increase computation energy.

Loop fission [20] (or loop distribution) reduces the size of a loop body, splitting it into two or more separate loops. While this transformation decreases energy consumption in the memory system, by reducing the number of instruction cache misses and data cache conflicts, and allowing the use of the loop buffer when the new loop bodies are small enough to fit in, it may increase memory system energy by destroying temporal locality (within one loop body data can often be stored in registers, but between loops the cache, or even background memory, has to be used). Computation energy is increased due to the new loop control structures, but the smaller number of cycles resulting from a better scheduling may also reduce it.

Loop fusion [20] (or loop merging), the opposite of loop fission, combines one or more loops with the same bounds into a single loop. It may create more temporal data locality that decreases energy used by the memory system. It may also add new instruction cache misses or hinder the use of the loop buffer, which increases memory system energy. As part of the loop control structures are eliminated and a better scheduling may be done, computation energy is reduced.

IV. Experimental Environment and Benchmarks

Figure 2 shows the compilation framework used in our experiments. It consists of a code transformation tool, a compiler, and an architecture simulator. In contrast with previous work done in iterative compilation [15, 16], which was concerned with applications written in Fortran, our study is focused on ANSI C applications, as C is the main programming language for embedded systems. In each experiment, a sequence of source-to-source transformations (together with their specific parameters) was applied to the application source, using CTT [4], a fully programmable code transformation tool developed at Delft University of Technology. The resulting C code was compiled with gcc and the Wattch 1.0 toolset [5] was used to evaluate the energy consumption and the performance of the binary code. Wattch is a framework that contains a power analyzer based on the Simplescalar [6] architectural simulator. Energy estimation is based on a suite of parameterizable power models of all the hardware structures of the processor and on the use of these structures. As Wattch supports three conditional clocking styles for disabling parts or all of the hardware to reduce power consumption when they are not used, we have chosen the one we found the most realistic. It assumes that if only a portion of a unit's ports is used, the power will scale linearly according to the number of ports being used. If a unit is unused, 10% of its maximum power is taken into account. In our studies, we used an architecture similar to the Alpha 21064 processor. In order to perform a fair comparison in both energy consumption and performance, a four-way 1024 KB unified data and instruction

[1]During cycles in which the processor is stalled because of cache misses, it consumes a percentage of the maximum energy.

```
for (y=1; y<IMAGE_HEIGHT-1; y++)
 for (x=1; x<IMAGE_WIDTH-1; x++)
 {
   tmp = 128 + image[y][x]*12;
   tmp += image[y-1][x-1]*-1 + image[y-1][x]*-2;
   tmp += image[y-1][x+1]*-1 + image[y+1][x]*-2;
   tmp += image[y+1][x-1]*-1 + image[y][x-1]*-2;
   tmp += image[y+1][x+1]*-1 + image[y][x+1]*-2;
   if (tmp > 255)
     target[y][x] = 255;
   else if (target[y][x] < 0)
     target[y][x] = 0;
   else
     target[y][x] = tmp;
 }
```

Fig. 3. Convolution Filter

TABLE I
THE EXPERIMENTS DONE ON KERNELS

Benchmark	Data L1-cache parameters	gcc optim. level	Matrix size	Charts
SOR	2K, 2-ways	O6	512	Fig. 4, a–b
	4K, 2-ways	O6	512	–
MxM	16K, 2-ways	O6	100	–
	16K, 2-ways	O6	128	Fig. 4, c–d
MxV	16K, 2-ways	O6	512	Fig. 5, a–b
	16K, 2-ways	O6	600	Fig. 5, c–d
	16K, 2-ways	O2	512	–
	16K, 2-ways	O2	600	–
LUD	16K, 2-ways	O6	100	Fig. 6, a–b
	16K, 2-ways	O6	128	Fig. 6, c–d

L2-cache was used in all experiments, as Wattch does not model external DRAM. In our case, this is not a problem, as our experiments show that the number of L2-cache misses for this configuration depends on the application and hardly on loop transformation parameters. Hence we can leave out the almost constant contribution of external memory for energy consumption and cycles. The data L1-cache was a parameter of the experiment.

In order to evaluate the potential of iterative compilation for reducing energy consumption, we first focussed our attention to four small kernels of scientific computation and multimedia applications, that exhibit a wide variety of memory access patterns: Matrix-Matrix Multiplication (MxM), Matrix-Vector Multiplication (MxV), LU Decomposition (LUD) and Successive Over Relaxation (SOR). On these benchmarks, we have applied two loop transformations: loop tiling [10, 17] and unroll-and-jam [7] (except for SOR, where loop unrolling was applied instead of unroll-and-jam, because of the structure of the application). From the application performance point of view, these transformations seem to be highly interdependent and their compound result gives rise to a highly irregular optimization space [16]. Combining the best tile size with the best unroll factor does not necessarily give the best overall transformation.

The performance of the benchmarks was collected for all square tiles with sizes between 1 and 100 and all unroll factors between 1 and 32. An exception is the MxM benchmark, for which the maximum unroll factor was 24 due to the very long simulation time and the bad performance obtained for higher values of the unroll factor. We use only square tiles because the obtained speedup is as good as the one obtained using rectangular tiles. However, the time needed for iterative compilation is a factor of 5 smaller for square tiles than for rectangular tiles [16]. This is due to the huge reduction of the search space and the higher percentage of good solutions in the remaining search space.

As the above kernels contain only one loop nest, loop fusion and loop fission cannot be applied to them. To test the effect of these transformations, we combine a sequence of four filters used in the image processing field (convolution 3x3, histogram equalization, center-of-gravity computation and image binarization) into one program. Every filter consists of a loop nest that iterates over all the image pixels and performs some computation on them. Except for the first filter (convolution, presented in Fig. 3), the computation for a pixel depends only on the computation of the previous filters for the same pixel and not on the computation for other pixels (e.g. the third filter can start its computation for image pixel x,y if the second filter finished with that pixel). Considering their properties, these loop nests can be merged using loop fusion. All the possibilities of merging these four filters are considered (the first loop nest merged with the second one and the last two alone; the first and the fourth alone and the second merged with the third, etc). In the end, after loop fusion, loop unrolling is applied. All unroll factors from 1 to 32 are considered. As loop fission is the opposite of loop fusion, no separate experiments for fission are needed.

The benchmarks' data input sizes for all experiments were chosen to be big enough to produce many data cache misses. Also, they cover both pathological (when array sizes are multiples of big powers of two) and non-pathological cases.

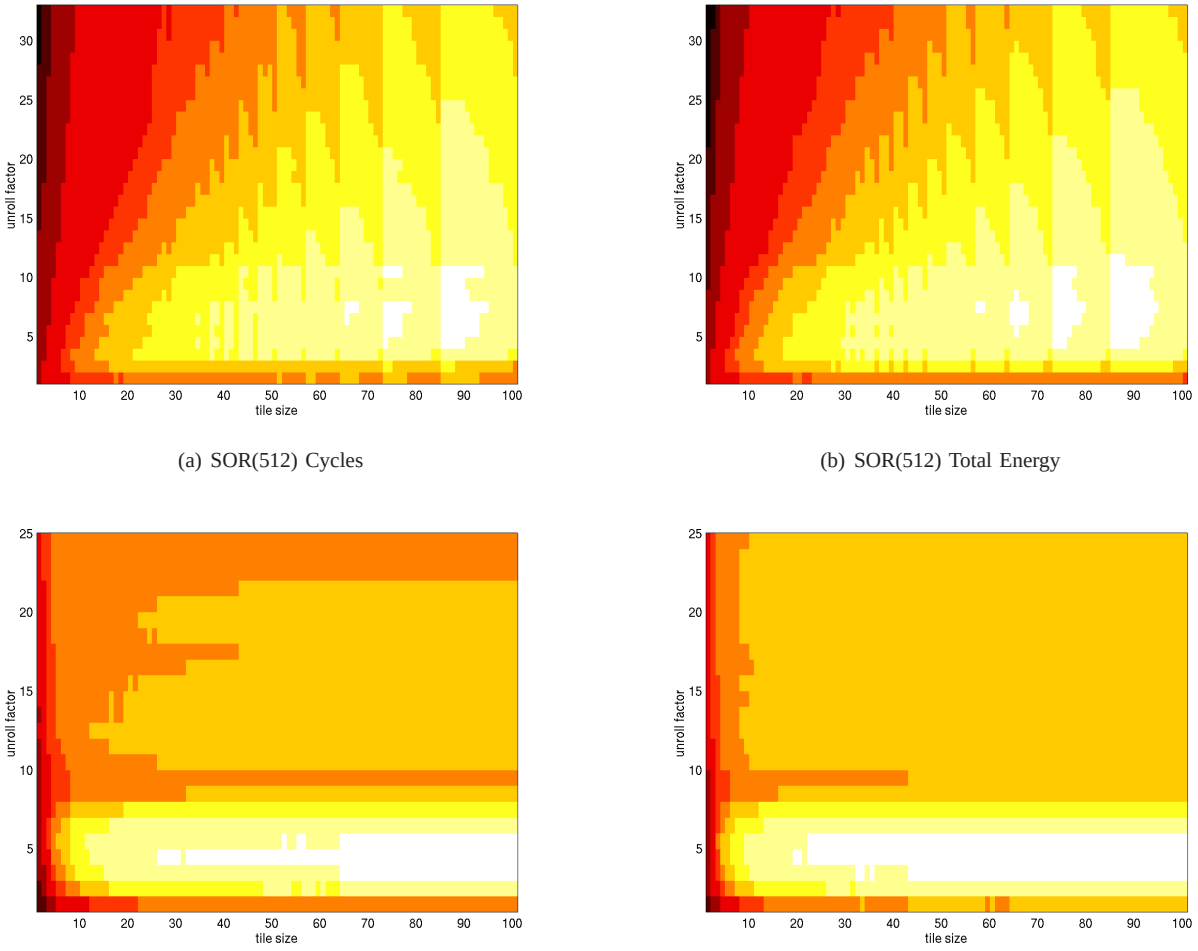
V. RESULTS AND EVALUATION

A. Kernel Evaluation for Loop Tiling and Unrolling

Table I presents the list with experiments that we have done. Each entry shows a benchmark, the input data size and the environment used (data L1-cache parameters, gcc optimization level). Only a representative subset of the results is presented in Fig. 4-6 (see the last column of Table I). For each experiment, two charts were generated, one for performance and one for energy consumption. In all charts, the horizontal axis represents the variation of the tile size and the vertical axis represents the variation of the unroll factor. White points in a chart represent the best solutions in the transformation space. The darker a point is, the worse the solution gets (see Table II).

Charts (a–b) in Fig. 4 show the evaluation of the SOR benchmark, considering a data L1-cache of 2K and a matrix input size of 512 (the number of matrix elements is 512x512). Comparing the results, the transformation space for total energy is very similar to the one for performance. The results for the second experiment done on the SOR benchmark are very similar to the first one.

From the two experiments for the MxM benchmark, charts (c–d) in Fig. 4 show the evaluation for matrix size 128. This size was chosen, because its transformation spaces are more irregular than the ones for matrix size 100. We extensively



(a) SOR(512) Cycles

(b) SOR(512) Total Energy

(c) MxM(128) Cycles

(d) MxM(128) Total Energy

Fig. 4. SOR and MxM Kernels Evaluation

TABLE II
CHARTS COLOR THRESHOLDS

Color	Worse than the best solution
	Between 0% – 3%
	Between 3% – 10%
	Between 10% – 17%
	Between 17% – 25%
	Between 25% – 35%
	Between 35% – 50%
	Between 50% – 100%
	Between 100% – 200%
	Between 200% – 300%
	More than 300%

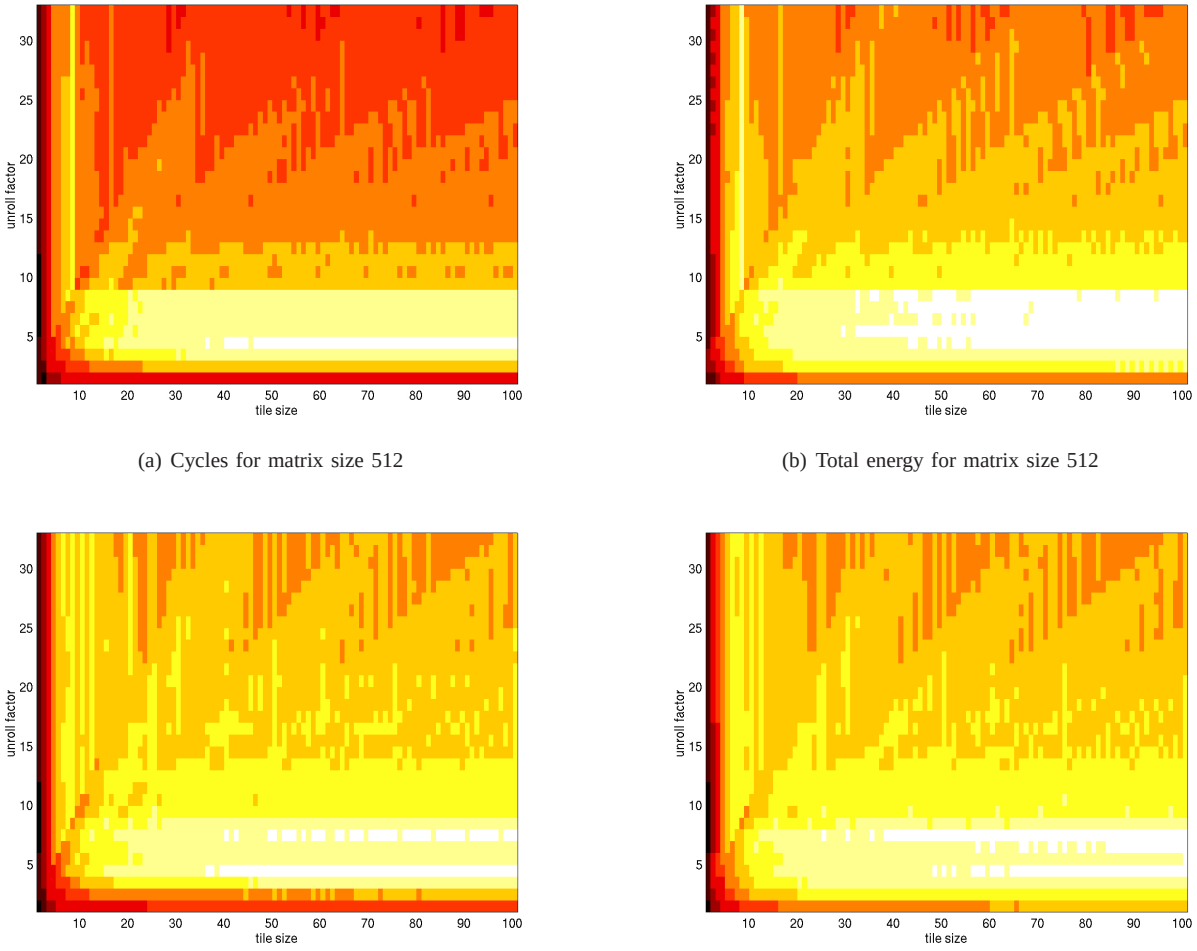
looked for an explanation, but could not find it. Its transformation spaces differ from the ones for SOR, although the area covering the best solutions is the same. The similarity between the energy and the performance transformation spaces is observed for this benchmark, as well.

Our experiments with the MxV benchmark show that when gcc is used with the O6 optimization level, the performance and energy transformation spaces are more regular than when O2 is used, eliminating most of the 'special' cases[2]. In Fig. 5, we present the experiments that use gcc O6, because it is more challenging to improve the best results obtained by a back-end compiler. The charts, which show the evaluation for matrix input sizes 512 and 600, evidence the fact that the transformation space does not coincide for different data input sizes. In contrast to the results for MxM, for this benchmark, the transformation spaces are slightly more regular for the pathological case, when the input size is a power of two ($512=2^9$).

Figure 6 shows the results of both experiments done for the LUD benchmark. The left upper triangles of the charts, that show bad performance, appear because the unroll factor cannot be bigger than the tile size[3] (for example, from the generated code with unroll factor 10 and tile size 2, only the first 2 copies of the loop body are executed, the rest

[2]When gcc O2 optimization is used, a bad performance is obtained for tile size 16, compared to 15 or 17. The array padding optimization, enabled in a higher optimization level of gcc, solves this problem.

[3]This observation applies also to the previous experiments, but it is not so obvious in the presented charts.



(a) Cycles for matrix size 512

(b) Total energy for matrix size 512

(c) Cycles for matrix size 600

(d) Total energy for matrix size 600

Fig. 5. MxV Kernel Evaluation

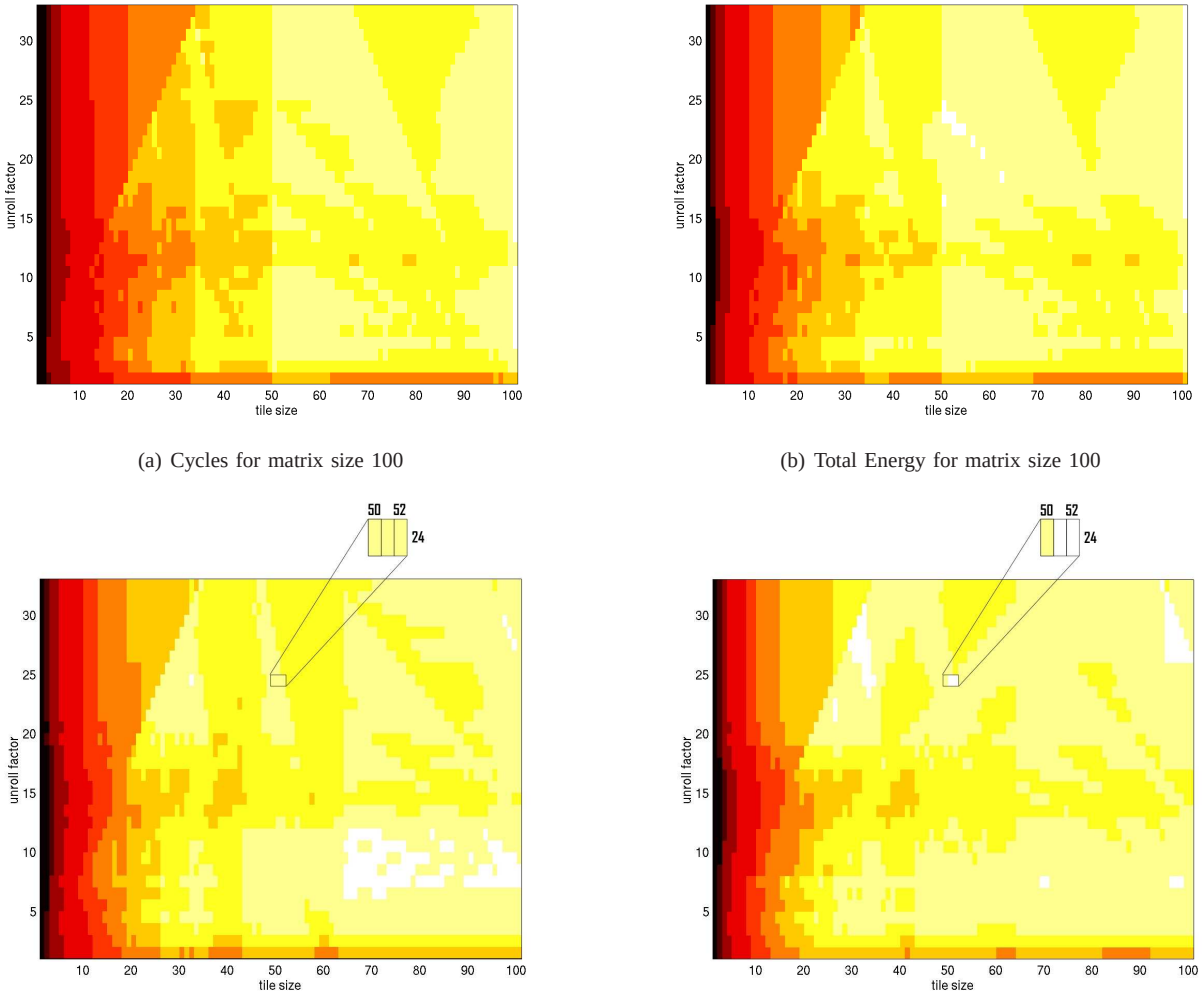
of them introducing only overhead for testing conditions). This benchmark accentuates the earlier conclusion that the transformation spaces do not coincide for different input sizes. From the charts, it can also be observed that, even though the transformation spaces are similar for total energy and performance, one cannot be included in the other. This means that if the best solution for performance is found, it will not always be the best one for energy, or vice versa.

In order to evaluate this observation further, we analyzed two points of the transformations spaces for the LUD 128 experiment. The points are (50,24) and (52,24) and they are highlighted in charts (c–d) from Fig. 6. Even though both points are between 3% and 10% worse than the best solution in the cycles chart, the first one is between 3% and 10% and the second one is between 0% and 3% in the energy chart. This means that the average energy per cycle is not the same for these two points. We found out that the difference is coming from the energy used by branch prediction. This is due to different numbers of conditions in the two input codes. The number of conditions is determined by the tile size and unroll factor. Based on this observation, for this experiment, we can conclude that from two solutions that run in the same number of cycles, the best one for energy is the one with the smaller number of conditional branches. There may be other situations in which similar effects may occur (e.g. accesses to the cache instead of the registers in architectures where both accesses take the same number of cycles [8]).

Analyzing all the obtained transformation spaces, we see that, in contrast with the rest of the benchmarks, the ones for the MxV benchmark have more local minima that are not also global minima. The number of local and global minima must be considered when a search algorithm, used to quickly find the best solution in the transformation space, is chosen.

B. Benchmark Evaluation for Loop Fusion

The sequence of image filters that we have selected as the benchmark for evaluation of collective effects of loop fusion and fission together with loop unrolling, was executed using 16K data L1-cache and 2K instruction L1-cache. We have chosen this small size instruction cache, because both loop fusion and loop unrolling increase the size of the loop body, which can produce instruction cache misses. We wanted to measure the effect of these cache misses. We chose small val-



(a) Cycles for matrix size 100

(b) Total Energy for matrix size 100

(c) Cycles for matrix size 128

(d) Total Energy for matrix size 128

Fig. 6. LUD Kernel Evaluation

ues for the loop transformation parameters, because otherwise the simulation time may be very long.

Figure 7 shows the charts for performance and energy consumption, for matrix input size 512. The horizontal axis gives the loop fusion level (e.g. 1,2-3,4 means that the second and third loop nests are merged, and the first and fourth remain separate). The vertical axis represents the variation of the unroll factor. The charts use the color representation for indicating quality of solutions defined in Section V-A. The white colored points represent the best solutions and the darker a point is, the worse the solution gets (see Table II).

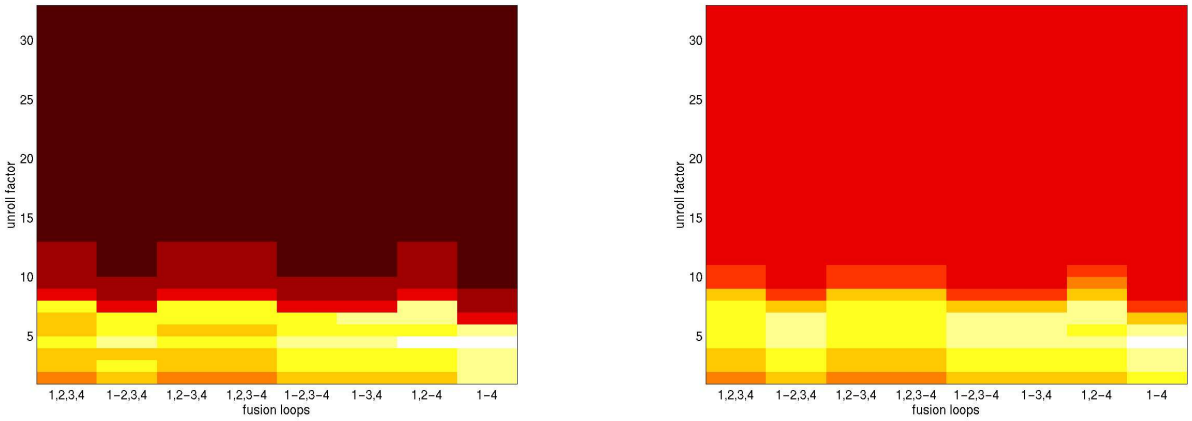
As in the previous experiments, in these cases, transformation spaces for total energy are very similar to the ones for performance, as well. We note that for these experiments, the total energy of the solutions varies between 100% and 200% of the best solution, and the performance is within 400% of the best performance achieved. Thus if the best solution is missed, the energy penalty will be lower than the performance penalty. There are no big differences between these transformations spaces and the ones obtained for matrix size 600.

C. Energy Distribution

This section analyzes the distribution of total energy over the memory and computation components. We are also looking at the way the energy decreases, i.e., whether both energy components decrease simultaneously or whether the consumption is transferred from one component to another. Based on the information received from Wattch (used technology: 0.35), we divided the total energy in three parts: memory system energy, computation energy and total clock energy[4].

Figure 8 presents the energy distribution for the LUD benchmark executed on a matrix of size 128 (Fig. 6(d) shows the total energy for this benchmark). We limit our discussion only to this experiment because the others have similar energy distributions. The charts' horizontal axes represent the variation of the tile size and the vertical axes represent the variation of the unroll factor. For each point in the transformation space, the percentage of the total energy used by each of the three

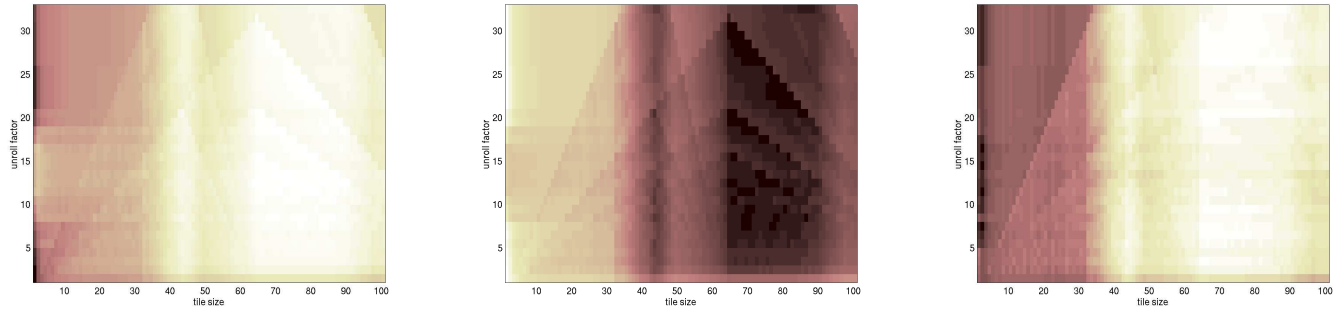
[4]For the Alpha 21064 architecture, the percentage of the total clock energy is very high, up to 34%, because it includes all clock capacitances, including the clock nodes within individual units.



(a) Cycles for matrix size 512

(b) Total energy for matrix size 512

Fig. 7. Benchmark Evaluation for Loop Fusion



(a) Computation Energy (Min,Max) = (36.2,44.0)

(b) Memory Energy (Min,Max) = (20.3,30.3)

(c) Clock Energy (Min,Max) = (33.4,36.6)

Fig. 8. LUD128 Energy Distribution (Percentages from the Total Energy)

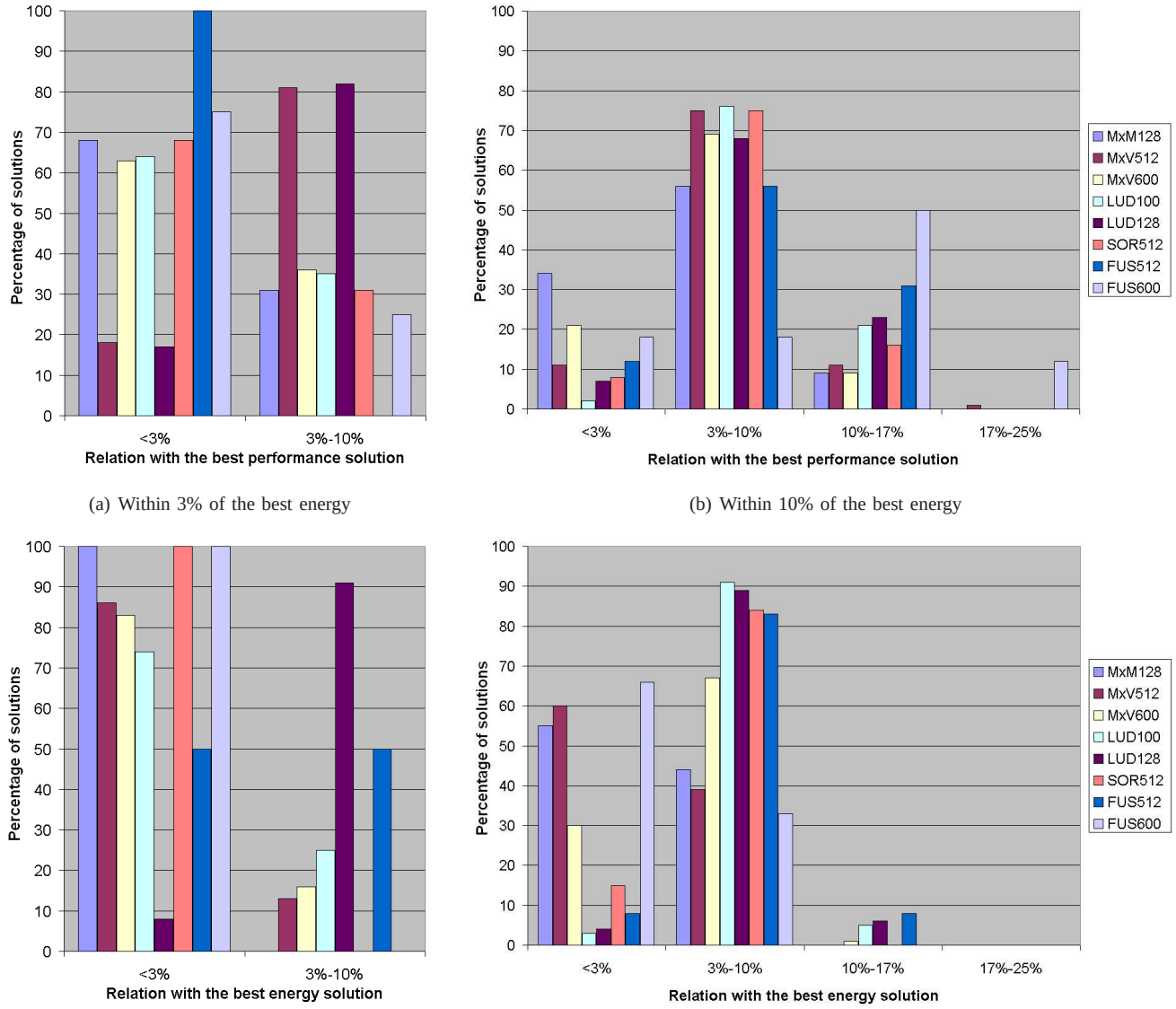
sources (memory, computation and clock) is computed. All charts have different scales for the color of the points. For every one, the white color represent the smallest percentage and the black color the highest percentage. The darker a point is, the larger the percentage of total energy that is used. The minimum and maximum percentage are printed under the charts (e.g. for memory computation, the white points represent 36.2% of the total energy and the black points 44.0%).

The energy used in the memory system is in all the points of the transformation space smaller than the energy used for computation. Analyzing the distribution of the total energy, we observe that in points with high energy consumption (see Fig. 6(d)), a big percentage of it (42%–44%) is used by computation and a small percentage in the memory system (20%–22%). As solutions become better, the percentage of the total energy used in the memory system becomes bigger (up to 30%), even though the absolute amount of memory energy decreases with a few percent. At the same time, the percentage of energy used for computation decreases (down to 36%), because of the reduction in the number of cycles. There are two possible explanations for this effect: the transformation shifts workload from computation to memory or both energy components decrease, but the computation energy decreases faster than memory energy. Based on our analysis, we conclude that in all our experiments, the second explanation is the correct one, namely that the energy reduction is more efficient for computation energy, than for memory system energy. The percentage of energy used by the clock shows a small variation, but it follows the same trend as computation energy, for the same reason, namely reduction of the number of cycles.

D. Evaluation Summary

In the introduction, we set out two major goals, namely to see how loop transformations affect energy consumption and to investigate whether iterative compilation is a useful approach for compiling for low energy.

With respect to the first goal, we can conclude that loop transformations do have a similar effect on energy consumption and performance. However, as for example Fig. 6 shows, the transformation spaces on energy and performance do not coincide 100% and they are also not included one in another. Charts (a) and (c) in Fig. 9 show the distribution of good solutions that are within 3% of the best energy resp. performance in the performance resp. energy transformation space



(a) Within 3% of the best energy

(b) Within 10% of the best energy

(c) Within 3% of the best performance

(d) Within 10% of the best performance

Fig. 9. The Correspondence between the Energy and Performance Transformation Spaces

(e.g. chart (a), MxM128 benchmark: the chance that a good solution for energy is also a good solution for performance is 68%, whereas the chance that it is between 3% and 10% worse than the best performance solution is 32%). Charts (b) and (d) in Fig. 9 make the comparison for solutions that are within 10% of the best solution. We observe that there are generally more good performance solutions that point to good energy solutions than vice versa. Considering the results, we conclude that to obtain the best solutions for energy and performance, a combined estimator factor, like energy-delay product [12], must be used.

With respect to the second goal, some of the experiments show very irregular transformation spaces, with many local minima. Such a space will be hard to capture in an analytical model, and it will be hard to devise a search algorithm without random elements that will not often get stuck in local minima. Therefore, these experiments suggest that iterative compilation is not only a good approach for performance optimization, but also for energy optimization.

VI. Conclusion

In this paper, we have analyzed, both theoretically and experimentally, the effect of several important loop transformations on energy consumption. For all analyzed benchmarks, the transformation space for energy and the one for performance look very similar. However, despite this similarity, the problem cannot be reduced to finding the best solution for performance, as it will not be always the best solution for energy, or vice versa. Based on this and on the irregularity of the transformation spaces, we conclude that the iterative compilation method is a good approach for compiling for energy, or for a combined optimization criterion, like energy-delay product.

In the future, we plan to extend our work to multiprocessor systems. In this case, the problem is more complex, because

there are more factors that affect the energy consumption (e.g. the mapping of portions of the source on different processors, the amount of communication between processors).

We are also interested in studying the possibility of using iterative compilation ideas together with the energy-delay product concept, in order to reduce the energy consumption for a given performance level. We would like to consider an approach that starting from a fast version of an application, will decrease the energy based on reducing its performance down to the requested limits.

ACKNOWLEDGMENT

We would like to thank Peter Knijnenburg for his valuable input during early stages of this work.

REFERENCES

[1] J. Auslander, M. Philipose, C. Chambers, S.J. Eggers, and B.N. Bershad, "Fast, effective dynamic compilation," in *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 1996, pp. 149–159.

[2] R.S. Bajwa, M. Hiraki, H. Kojima, D.J. Gorny, K. Nitta, A. Shridhar, K. Seki, and K. Sasaki, "Instruction buffering to reduce power in processors for signal processing," *IEEE Transactions on Very Large Scale Integration (VLSI)*, vol. 5, no. 4, pp. 417–424, December 1997.

[3] L. Benini and G. de Micheli, "System-level power optimization: techniques and tools," *ACM Transactions on Design Automation of Electronic Systems*, vol. 5, no. 2, pp. 115–192, April 2000.

[4] M. Boekhold, I. Karkowski, H. Corporaal, and A. Cilio, "A programmable ANSI C transformation engine," in *Proc. of the 8th International Conference on Compiler Construction*. Springer Verlag, 1999, pp. 292–295.

[5] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *Proc. of the 27th International Symposium of Computer Architecture*, 2000, pp. 83–94.

[6] D. Burger and T.M. Austin, "The simplescalar tool set, version 2.0," *ACM SIGARCH Computer Architecture News*, vol. 25, no. 3, pp. 13–25, June 1997.

[7] S. Carr, "Combining optimization for cache and instruction-level parallelism," in *Proc. of PACT*. IEEE, 1996, pp. 238–247.

[8] L.N. Chakrapani, P. Korkmaz, V.J. Mooney, K.V. Palem, K. Puttaswamy, and W.F. Wong, "The emerging power crisis in embedded processors: what can a poor compiler do?" in *Proc. of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. ACM Press, 2001, pp. 176–180.

[9] R. Cohn and P. Lowney, "Feedback directed optimization in Compaq's compilation tools for Alpha," in *Proc. of the 2nd Workshop on Feedback Directed Optimization*, November 1999, pp. 3–12.

[10] S. Coleman and K.S. McKinley, "Tile size selection using cache organization and data layout," in *Proc. of SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 1995, pp. 279–290.

[11] K.D. Cooper, D. Subramanian, and L. Torczon, "Adaptive optimizing compilers for the 21st century," *Journal of Supercomputing*, vol. 23, no. 1, pp. 7–22, August 2002.

[12] R. Gonzalez and M. Horowitz, "Energy dissipation in general purpose microprocessors," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 9, pp. 1277–1284, September 1996.

[13] T. Granlund and R. Kenner, "Eliminating branches using a superoptimizer and the GNU C compiler," in *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 1992, pp. 341–352.

[14] M.T. Kandemir, N. Vijaykrishnan, M.J. Irwin, and W. Ye, "Influence of compiler optimizations on system power," in *Proc. of Design Automation Conference*. ACM Press, 2000, pp. 304–307.

[15] P.M.W. Knijnenburg, T. Kisuki, and M.F.P. O'Boyle, "Iterative compilation," in *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation - SAMOS*, ser. Springer Lecture Notes in Computer Science, vol. 2268. Springer Verlag, 2002, pp. 171–187.

[16] P.M.W. Knijnenburg, T. Kisuki, and M.F.P. O'Boyle, "Combined selection of tile sizes and unroll factors using iterative compilation," *Journal of Supercomputing*, vol. 24, no. 1, pp. 43–67, 2003.

[17] M.D. Lam, E.E. Rothberg, and M.E. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proc. of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, 1991, pp. 63–74.

[18] H. Massalin, "Superoptimizer: a look at the smallest program," in *Proc. of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*. IEEE Computer Society Press, 1986, pp. 122–126.

[19] M. Miranda, C. Ghez, C. Kulkarni, F. Catthoor, and D. Verkest, "Systematic speed-power memory data-layout exploration for cache controlled embedded multimedia applications," in *Proc. of the 14th International Symposium on Systems Synthesis*. ACM Press, 2001, pp. 107–112.

[20] S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc, August 1997.

[21] V. Tiwari, S. Malik, and A. Wolfe, "Compilation techniques for low energy: An overview," in *IEEE Low-Power Electronics Symposium*, 1994, pp. 38–39.

[22] M. Valluri and L. John, "Is compiling for performance == compiling for power?" in *The 5th Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT-5)*, Monterrey, Mexico, January 2001.

[23] M. Voss and R. Eigenmann, "ADAPT: Automated de-coupled adaptive program transformation," in *Proc. of the International Conference on Parallel Processing*, 2000, pp. 163–170.