Using Iterative Compilation to Reduce Energy Consumption

Stefan Valentin Gheorghita, Henk Corporaal and Twan Basten

Eindhoven University of Technology, PO Box 513, 5600 MB, Eindhoven, The Netherlands.
{s.v.gheorghita,h.corporaal,a.a.basten}@tue.nl

Keywords: Iterative Compilation, Program Optimization, Energy Consumption, Program Transformation.

Abstract

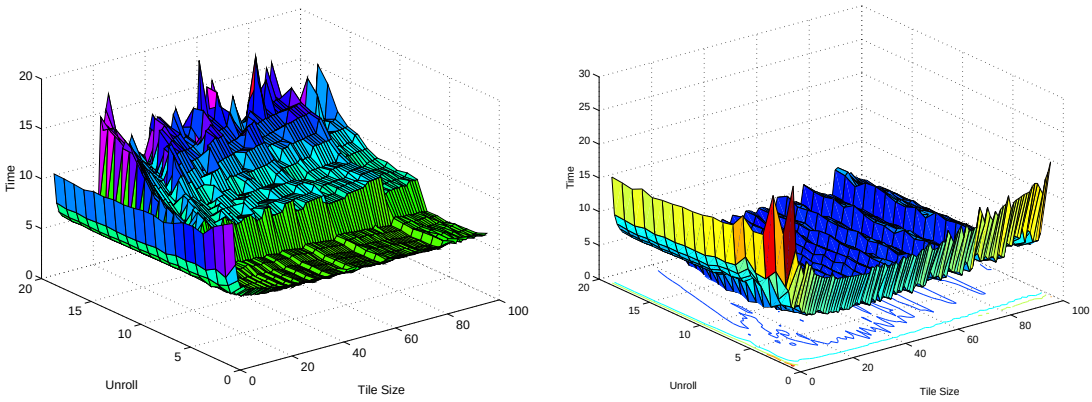
The rapid range of architectural changes in processors puts compiler technology under an enormous stress. This is emphasized by new demands added to compilers, like reducing static code size, energy consumption or power dissipation. Iterative compilation has been proposed as an approach to find the best 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 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 and for the combination of energy and performance. From analyzed benchmarks, we conclude that performance improvement is coming together with decreasing energy consumption. Iterative compilation seems therefore a promising approach to the compilation for energy problem, but a larger set of loop transformations and their combinations needs to be studied for a definitive conclusion.

1 Introduction

Each year, computing technology starts being used in new areas. The number of computers in use is growing and the increasing demand for greater performance in all areas of computing leads to an exponential growth of hardware performance, and also of hardware diversity. The variety of processors, with different instruction sets, performance parameters and memory hierarchies, is increasing, especially in the embedded systems area where the backward compatibility is 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 became important in the late 1990s because of the limited size of embedded systems memory and also of the importance of transferring applications over the internet. The rise of mobile embedded systems (like mobile phones, PDAs, digital cameras) directs also the interest of compilers 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 the cache hierarchy or to reduce power. Most transformations require loop restructuring. Although a large number of transformations have been recognized as being potentially beneficent [1], 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 [1]. This sequence is largely based on observed behavior of a small number of benchmarks. The values of the 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 much difficulty 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 [2, 3] and loop unrolling [4], figure 1 from [5] shows the execution time of Matrix-Matrix Multiplication for several tile sizes and unroll factors on two different platforms. It can



(a) Pentium II (b) UltraSparc I

Figure 1: Execution Time Matrix-Matrix Multiplication for Unrolling and Tiling.

be observed that a small deviation from good tile sizes or unroll factors can cause a huge increase of the execution time and even a slowing down compared with 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 [5, 6] Knijnenburg, Kisuki and O'Boyle have proposed the concept of *iterative compilation*, where many variants of the source programs 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. The authors have shown that by randomly evaluating a small percentage of the transformation space, 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 iterative compilation study, considering the following two problems: (i) the effect of loop transformations on energy consumption and (ii) the possibility of using iterative compilation in order to find the best compiled code for energy and energy-delay product[1] [7]. It is organized as follows. Section 2 presents the related work in dynamic compilation techniques and in compilation for energy. Section 3 presents a theoretical analysis of the effects of a few important loop transformations on energy consumption. The experimental environment and the used benchmarks are presented in section 4. The last two sections, 5 and 6, present some results and the conclusions drawn from the experiments done.

2 Related work

Besides the iterative compilation approach introduced in [5, 6], several ways to dynamically improve compiled code were proposed. For example, [8, 9, 10] use profiling and runtime information to find which transformations are best for a given application and platform. Other attempts to use search algorithms in optimizations include Massalin's Superoptimizer [11], which tries to find the optimal instruction selection using an exhaustive search, and Keith Cooper's adaptive compiler which uses genetic algorithms trying to reduce the application code size and running time [12]. Massalin's technique produces good results, but it was too time expensive. Granlund and Kenner adapted Massalin's ideas and created a faster superoptimizer that generates gcc assembly compatible code [13]. In all tested cases, Cooper's adaptive compiler improved the running time with up to 20% and the reduced the code size with up to 13%.

The reduction of energy consumption by generating better software code has been a research issue for the last ten years [14]. 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 [15] studied the effect of compiler optimizations on power dissipation and energy, using a group of benchmarks com-

[1] The product between total energy and number of cycles. It is used when the minimum energy at a given performance level, or more performance for the same energy is required. In this case, both energy and performance must be considered simultaneously.

piled with gcc having only one compiler optimization enabled. In [16], 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 individual loop transformation using the compiler default value for its parameters. In contrast, our work is concerned on the combined effect on energy consumption for a sequence of loop transformations considering a large set of values for their parameters.

3 Loop transformation effects on energy consumption

Loop transformations are source-to-source restructuring transformations which modify either the body or the control structure of the loop, in order to change the iteration space of the loop nest. These changes may improve the instruction and data cache performances, the instruction level parallelism (ILP), and the iteration level parallelism. They are very effective because a small gain inside a loop iteration may result in a big improvement considering the number of iterations. While loop transformations always aim to reduce the number of cycles taken by an application to run, they might not decrease the 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 effects of a few of these transformations on system energy are analyzed in the following paragraphs. The chosen transformations, loop unrolling, unroll-and-jam and loop tiling, are the most used transformations in current commercial compilers.

Loop unrolling [1] replicates the loop body for 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 if it reduces computation energy, due to less loop control structures and array offset computations and a better instruction scheduling, it may increase the computation energy because of the newly added instructions (eg. array subscript expressions). Temporal data locality may be improved, allowing the usage of registers instead of the data cache, which reduces the memory system energy. If the chosen unroll factor is too big, more instruction cache misses are produced and also, the *loop buffer* [17], if present in the processor, becomes less efficient. So instructions temporal locality decreases, and memory system energy consumption increases. Taking into account all these effects, we expect that for a good unroll factor that improves performance, this transformation always reduces the energy used.

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** [4] and its effect on energy consumption does not differ from the ordinary loop unrolling transformation.

Loop tiling [2, 3] (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[2]. Contrariwise, the newly added loop control structures increase computation energy. Combining all these consequences, this transformation is expected to always reduce the system's energy consumption when increasing the performance.

4 Experimental environment and benchmarks

Figure 2 shows the compilation framework used in our experiments. It consists of a code transformation engine, a compiler and an architecture simulator. In contrast with previous work done in iterative compilation [5, 6] 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 on the application source, using CTT [18], a fully programmable code transformation engine developed at Delft University of Technology. The resulting C code was compiled with gcc and the Wattch 1.0 toolset [19] was used to evaluate the energy consumption and the performance of the resulting binary code. Wattch is a framework that contains a power analyzer based on the Simplescalar [20] architectural simulator. The energy estimation is based on a suite of parameterizable power models of all the hardware structures of the processor and on the usage of these structures. As Wattch supports three conditional clocking styles for disabling parts of or all of the hardware to reduce the power consumption when they are not used, we chose the one we found most realistic. It assumes that if only a portion of a unit's ports is used, the power is

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

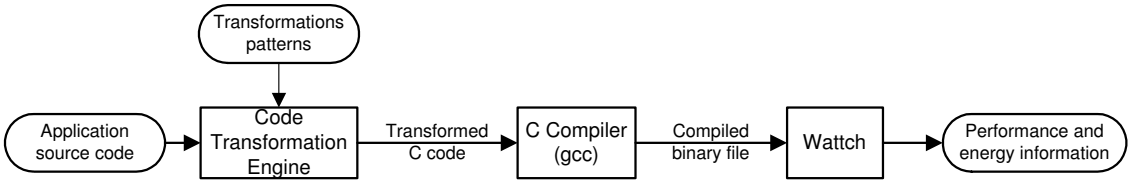


Figure 2: Compilation Framework.

scaled 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 two architecture configurations similarly with the Alpha 21064 and Alpha 21264 processors. In order to perform a fair comparison in both energy consumption and performance, a four-way 1024 KB unified data and instruction level-2 cache[3] was used in all experiments. The data level-1 cache was a parameter of the experiment.

In order to evaluate the iterative compilation potential for reducing energy consumption, we restricted our attention to three small kernels that exhibit a wide variety of memory access patterns: Matrix-Matrix Multiplication (MxM), Matrix-Vector Multiplication (MxV) and Successive Over Relaxation (SOR). On these benchmarks we applied two loop transformations: loop tiling [2, 3] and unroll-and-jam [4] (except SOR, where loop unrolling was applied instead of unroll-and-jam, due to 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 [6]. Combining the best tile size with the best unroll factor does not necessarily give the best overall transformation. The close interaction between tiling and unrolling can be seen in figure 1, which shows that a small deviation from good tile sizes or unroll factors can cause a huge increase in execution time.

The benchmarks' performances were collected for all square tiles with sizes between one and 100 and all unroll factors between one and 32. An exception was 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 almost as good as the one obtained using rectangular tiles. However, the time needed for iterative compilation is a factor of 8 smaller for square tiles than for rectangular tiles [6]. The benchmarks' data input sizes were chosen to be big enough to produce many data cache misses. Also, they cover both pathological (when arrays' sizes are multiples of big powers of two) and non pathological cases.

[3]DRAM memory power is not modelled in the Wattch toolset, and we used the level-2 cache to simulate the system's background memory

Color	Threshold (Compared with the best solution)
White	Less than 103%
	Between 103% - 110%
	Between 110% - 117%
	Between 117% - 125%
	Between 125% - 135%
	Between 135% - 150%
	Between 150% - 200%
Black	More than 200%

Table 2: Chart color/grayscale thresholds.

5 Results and evaluation

Table 1 presents the list with the experiments that we have done. Every row shows the benchmark's name, the input data size and the environment used (architecture type, data level-1 cache parameters, gcc optimizations level). Due to space constraints, only a representative subset of the results of these experiments (see the last column of the table) are presented in figure 3. For each experiment, three charts were generated, for performance, energy consumption and the energy-delay product optimization criterion. In all these charts, the horizontal axis represents the variation of the tile size and the vertical axis represents the variation of the unroll factor. White colored points from a chart represent the best solutions in the transformation space with respect to the chart's optimization criteria. The darker a point is, the worse the solution is. Table 2 shows the colors/thresholds.

Charts (a-c) from figure 3 show the evaluation of the SOR benchmark, on an Alpha 21064 architecture with a data level-1 cache of 2K, for a matrix input size of 512 elements. Comparing the results, the transformation space for total energy is very similar to the one for performance. Based on a detailed analysis of the code and of the collected information, we observed that the irregularity of the transformation space is caused only by the variation in the amount of computation, although the data cache is very small compared to the matrix size. The energy used by the entire memory system is 10% of the total energy and it

Benchmark	Architecture	Data L1-cache parameters	gcc optimization level	Matrix size	Charts
SOR	Alpha 21064	2K, two way, 32B block size	O6	512	a-c
	Alpha 21064	4K, two way, 32B block size	O6	512	–
MxM	Alpha 21064	16K, four way, 32B block size	O6	100	–
	Alpha 21064	16K, four way, 32B block size	O6	128	d-f
MxV	Alpha 21064	16K, four way, 32B block size	O6	512	g-i
	Alpha 21064	16K, four way, 32B block size	O6	600	j-l
	Alpha 21064	16K, four way, 32B block size	O2	512	–
	Alpha 21064	16K, four way, 32B block size	O2	600	–
	Alpha 21264	64K, two way, 32B block size	O2	512	–
	Alpha 21264	64K, two way, 32B block size	O2	600	–

Table 1: The experiments done.

varies within 10%, which means that the cache energy may not increase the total energy with more that 1%. The results for the second experiment done on the SOR benchmark are very similar with the ones previously presented.

From the two experiments done on an Alpha 21064 platform for the MxM benchmark, charts (d-f) from figure 3 show the evaluation for a matrix size of 128. This size was chosen, because its transformation spaces are more irregular than the ones for a matrix size with 100 elements. Its transformation spaces differ from the ones for SOR, although the area covering the best solutions is the same. In this case, the irregularity of the transformation space is generated by both computation and memory accesses. The observation about the similarity between the total energy and the performance transformation space, is true for this benchmark, too.

From the experiments on the MxV benchmark, it resulted that when gcc is used with the O6 optimization level, instead of O2, the performance and total energy transformation spaces are more regular, eliminating most of the "special" cases[4] from them. In charts (g–l) from figure 3 we present only the experiments that use gcc O6, because it is more challenging to improve the best results obtained by a backend compiler. The charts, which show the benchmark evaluation on Alpha 21064 for matrix input sizes of 512 and 600 elements, evidence the fact that the transformation spaces do not coincide for different data input sizes. The space is more regular for the pathological case, when the input size is a power of two ($512=2^9$). In contrast with the transformation spaces for the previous benchmarks, these have more local minima which are not global minima (especially for cycles and energy -delay product for data input size 600). The number of local and global minima must be considered when the search algorithm used to find the best solution in the transformation space is chosen.

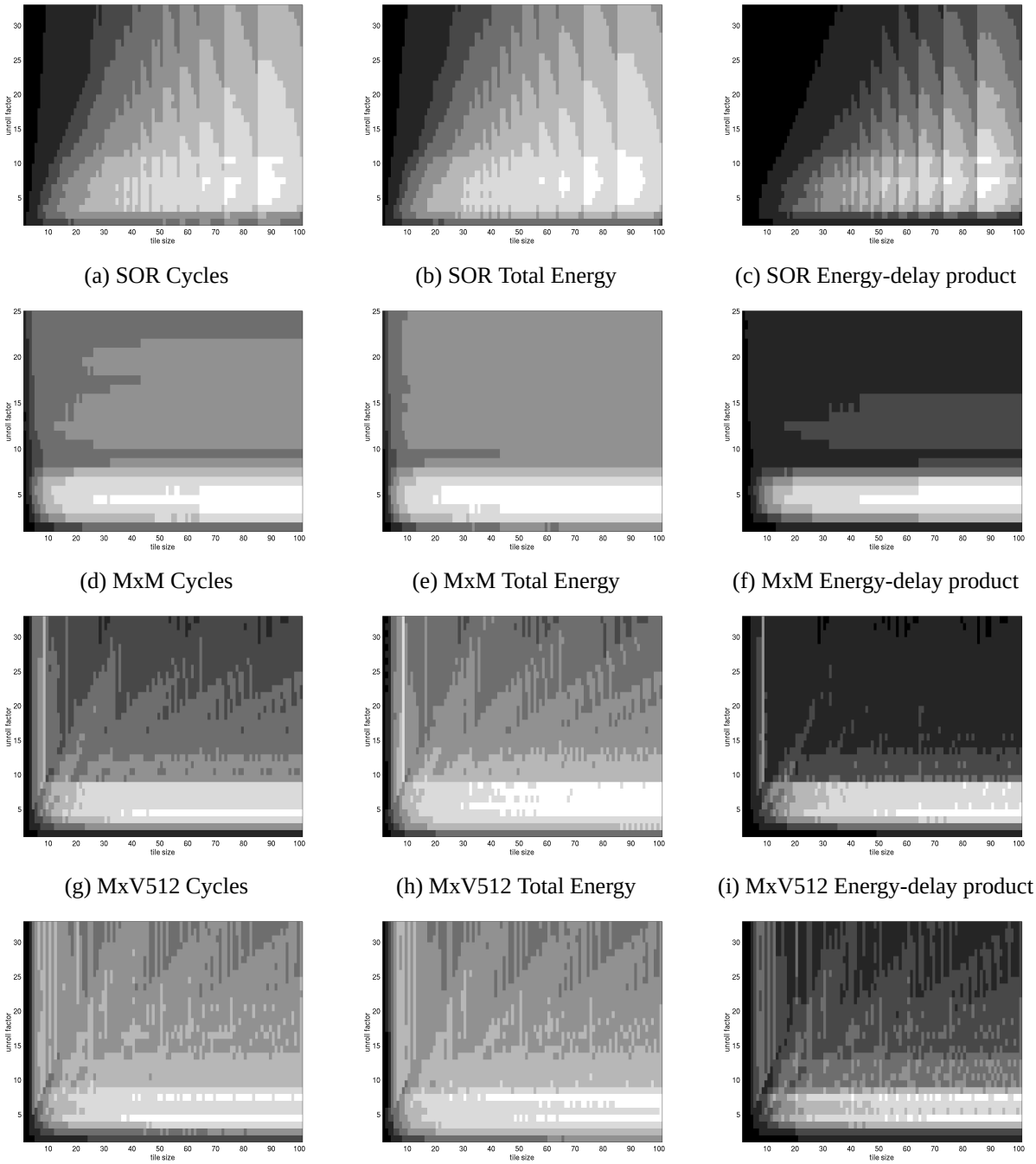
In the introduction, we set out two major goals, namely to see how loop transformations affect energy consumption and to investigate whether iterative compilation is a useful approach to the compiling for energy problem. With respect to the first goal, we can conclude that loop transformations do have an effect on energy consumption, in line with their effect on performance, confirming the theoretical analysis we made in a previous section. With respect to the second goal, particularly the last set of experiments shows a transformation space with many local minima. Such a space will be hard to capture in an analytical model. Therefore, these experiments suggest that iterative compilation is a useful approach to the compilation for energy problem.

6 Conclusion

In this paper, we analyzed, both theoretically and experimentally, the effect of several important loop transformations on energy consumption. For all analyzed benchmarks, the transformation spaces for energy and the one for performance look very similar. This may come from the behavior of the used transformations, loop unroll and tiling, which seem to always decrease the energy consumption together with performance improvement. Based on the analysis of the obtained results, we observe that, as the iterative compilation method is a good approach for compiling for performance, it is useful for compiling for energy, too. For a definite conclusion, a larger set of loop transformations and their combinations needs to be studied.

In future work, we will also look at 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

[4]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.



(a) SOR Cycles (b) SOR Total Energy (c) SOR Energy-delay product

(d) MxM Cycles (e) MxM Total Energy (f) MxM Energy-delay product

(g) MxV512 Cycles (h) MxV512 Total Energy (i) MxV512 Energy-delay product

(j) MxV600 Cycles (k) MxV600 Total Energy (l) MxV600 Energy-delay product

Figure 3: Benchmarks evaluation on an Alpha 21064 architecture: (a-c) SOR for matrix input size 512; (d-f) MxM for matrix input size 128; MxV for matrix input sizes 512 (g-i) and 600 (j-l)

the energy based on reducing its performance down to the requested limits.

References

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

[2] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proc. of the 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, (Santa Clara, CA, US), pp. 63–74, April 1991.

[3] S. Coleman and K. S. McKinley, "Tile size selection using cache organization and data layout," in *Proc. of SIGPLAN Conf. on Programming Language Design and Implementation*, (La Jolla, CA, US), pp. 279–290, ACM Press, 1995.

[4] S. Carr, "Combining optimization for cache and instruction-level parallelism," in *Proc. of PACT*, (Boston, MA, US), pp. 238–247, IEEE, October 1996.

[5] 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*, vol. 2268 of *LNCS*, pp. 171–187, Springer Verlag, 2002.

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

[7] R. Gonzalez and M. Horowitz, "Energy dissipation in general purpose microprocessors," *IEEE J. of Solid-State Circuits*, vol. 31, pp. 1277–1284, September 1996.

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

[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*, pp. 3–12, November 1999.

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

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

[12] K. D. Cooper, D. Subramanian, and L. Torczon, "Adaptive optimizing compilers for the 21st century," *J. of Supercomputing*, vol. 23, pp. 7–22, August 2002.

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

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

[15] 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.

[16] M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye, "Influence of compiler optimizations on system power," in *Proc. of Design Automation Conf.*, (Los Angeles, CA, US), pp. 304–307, ACM Press, 2000.

[17] 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, pp. 417–424, December 1997.

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

[19] 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*, (Vancouver, British Columbia, Canada), pp. 83–94, June 2000.

[20] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," *ACM SIGARCH Computer Architecture News*, vol. 25, pp. 13–25, June 1997.