Analysis and Modeling of Energy Reducing Source Code Transformations

C. Brandolese, W. Fornaciari, F. Salice, D. Sciuto
Politecnico di Milano, Piazza L. da Vinci, 32 - 20133 Milano, Italy

Abstract

This paper presents a methodology and a set of models supporting energy-driven source-to-source transformations. The most promising code transformation techniques have been isolated and studied leading to accurate analytical and/or statistical models. Experimental results, obtained for some common embedded-system processors over a set of typical benchmarks, are presented, showing the viability of the proposed approach as a support tool for embedded software design.

1. Introduction

In a growing number of complex heterogeneous embedded systems the relevance of the software component is rapidly increasing. Issues such as development time, flexibility and reusability are, in fact, better addressed by software based solutions. Another trend that is significantly pushing designers to move as much functionality as possible toward software is the increased interest in platform-based designs. In such systems much of the architecture is fixed and can only be configured to match the design constraints. The greatest part of the application-specific functionality is thus naturally shifted from hardware dedicated components to software programs. In such a scenario it is clear that the importance of software is steadily increasing and poses new problems to designers. Though performance, in the sense of computational efficiency, is still the foremost requirement for most embedded systems, power consumption is gaining more and more attention. Optimization of the code is thus one of the key points and is currently addressed almost only by means of compilation techniques. It is still not uncommon for designers to manually code critical sections of the application directly in assembly. The recent technical literature proposes a different approach, based on source-to-source transformations aimed at improving code quality either directly or by enabling better compiler optimizations. Source code transformations are extremely complex to automate since they require a thorough semantic analysis of the code fragments to be optimized. This paper proposes a consistent and flexible methodology for the analysis of the effect of source-to-source transformations mostly aimed at allowing rapid and accurate design space exploration. The proposed approach is based on a wide set of models studied to decouple the processor-independent analysis from all technology specific aspects. Section 2 presents a brief overview of the most promising transformations and Section 3 outlines the conceptual design flow and describes the technology-related models adopted for energy gain estimation. Section 4 presents two case studies to better clarify the approach whose results are summarized in Section 5. Finally, Section 6 draws some conclusions and outlines possible extensions and improvements.

2. Transformations Overview

Source-to-source transformation presented in literature, can be grouped in to four main areas according to the code structures they operate on: *loops*, *data structures*, *procedures* and *control structures and operators*. It is worth noting that not all the transformations are interesting when operating at source-level since some of them can as well be performed at RT or assembly-level and are thus performed by modern compilers. The most promising transformations, either found in literature [2, 5] or studied in the present work, are summarized in the following. Particular attention must be devoted to loop transformations [1, 6, 7, 8] since most of the execution time of aprogram is spent in loops.

Loop unrolling replicates the body of a loop a given number of times U (the unrolling factor), and modifies the iteration step from 1 to U. This transformation impacts on energy in two ways. First, it reduces loop overhead by performing less compare and branch instructions. Second, it allows the compiler for a better optimization and register usage of the larger loop body.

1530-1591/04 $20.00 (c) 2004 IEEE

Loop distribution breaks a single loop into multiple loops with the same iteration range but each enclosing only a subset of the statements in the original loop. Distribution is used to create subloops with fewer dependencies, improve instruction cache and instruction TLB locality due to shorter loop bodies, reduce memory requirements by iterating over fewer arrays and improve register usage by decreasing register pressure.

Loop fusion performs the opposite operation of distribution by reduceing loop overhead, increasing instruction parallelism, improving register, data cache, TLB or page locality. It also improves the load balance of parallel loops.

Loop tiling improves memory locality, primarily the at cache level, by accessing matrices in $N \times M$ sized tiles rather than completely. It also improves processor, register, TLB, and page locality.

Software pipelining breaks the operations of a single loop iteration into S stages, and arranges the code in such a way that stage 1 is executed on the instructions originally belonging to iteration i, stage 2 on those of iteration $i-1$, etc. Startup code must be generated before the loop to initialize the pipeline for the first $S-1$ iterations and cleanup code must be generated after the loop to drain the pipeline for the last $S-1$ iterations.

The second class collects a number of data-structure and memory access transformations [3, 8].

Scratch-pad array introduction has the purpose of storing the most frequently accessed array elements in a smaller array (the scratch-pad) to improve spatial locality.

Multiple indirection elimination identifies common chains of indirections and stores the address into a temporary variable.

The third group gathers those transformations [3] impacting on procedures and functions.

Function inlining replaces the most frequently invoked function with the function body. Inline expansion increases the spatial locality and decreases the number of function calls. This transformation increases the number of unique references, which may result in more misses. However, a decrease in the miss rate may also occur, since, without inlining, the callee code might replace the caller code in the instruction cache.

Soft inlining is an intermediate solution between function calling and inlining. The transformation replaces calls and returns with jumps. This reduces the code size w.r.t. inlining and eliminates context switching overheads.

Code linking directives can be used to suitably reorder the objects of different functions to match as more as possible the dynamic call graph. This potentially leads to a reduction in instruction misses.

Most of the transformation in the last group are usually performed by compilers. Nevertheless, some of them can still be conveniently considered when operating at source-level [3, 4].

Conditional sub-expression reordering exploits shortcut evaluation of conditions usually performed by compilers. The transformation operates by reordering the sub-expressions according to their probability of being true (for OR conditions) or false (for AND conditions). This reduces the number of instructions executed.

Special cases pre-evaluation allows avoiding a function call (usually a mathematic library function) when the argument has a special value for which the result is known. This is done by defining suitable macros testing for the special cases and leads to a reduction of actual calls.

Special cases optimization replaces calls to generic library or user-defined functions with optimized versions, suitable for common special cases. As an example, power raising on integers can be coded more efficiently than it can be for real numbers.

3. Methodology

Transformations applied to source code might lead to very different results depending on a number of factors: the specific structure of the code, the target architecture, the parameters of the transformations etc. Furthermore, it is not unusual that a transformation applied on the source code as it is lead to poor or no energy reduction, while, when applied to a pre-transformed code its usefulness is greatly increased. Thus sequences of transformations should be considered, rather than single transformations. For this reason it is essential to explore different transformations and sequences of transformations in terms of their energy reduction efficiency. The exploration strategy should allow to easily modify the parameters of the transformation and of the target technology and to have a quick estimate of the expected benefits.

3.1. Conceptual Flow

Figure 3.1 shows the conceptual scheme of the estimation flow. The source code is processed and its rele-

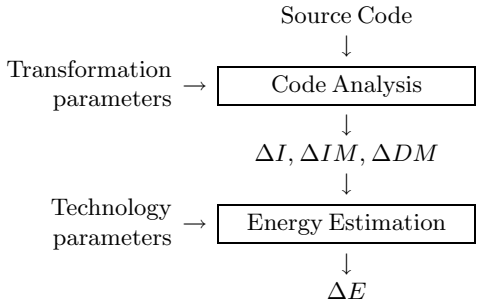

Figure 1: Methodology flow

vant characteristics are extracted by means of a lexical and syntactical analysis leading to the set of *code parameters*. Typical parameters are code size, loop body size, number of paths, number of loop iterations, etc. The designer then chooses the *transformations parameters* such as unroll factor, tiling size etc. and, finally, selects the target technology from a set of libraries. Such libraries are collections of *technology parameters* specifying architectural figures such as cache sizes, bus width etc. and electrical figures such as power supply voltages, average core currents, bus and memory capacitances etc. Based on all this data, the estimation models first provide the three dimensionless figures ΔI, ΔIM and ΔDM expressing the variations of number of instructions executed, of number of instruction misses and of number of data misses, respectively. These figures, though still rather abstract, already provide the designer with an indication of the potential benefits of a given transformation. To account for the target technology as well, the variations are fed to a set of models, depending on the *technology parameters*, leading to an estimate of the energy reduction ΔE deriving from the application of the considered transformation.

3.2. Technology Models

Experimental results have shown that the energy consumption of an embedded system based on a processor executing some programs can be approximated by considering three major contributions: the processor core and its on-chip caches, the system bus and the main memory. All these components can be modeled at different levels of accuracy by means of equations that involve two sets of parameters: those strictly related to the specific technology and those summarizing the properties and the behavior of the code being executed. In particular, as outlined in the description of the conceptual flow, the energy estimates can be based on three execution parameters only: the number of assembly instructions executed and the number of instruction and data cache misses. Though simple, the adopted models provide satisfactory results, especially when considering energy variations rather than absolute values. The technology parameters considered and used in the models are summarized in table 1. The

Symbol	Meaning
T_{ck}	CPU clock period
$\overline{CPI}$	Average CPI
$\overline{P}_{cpu}$	Average CPU power absorption
C_{tot}	Total capacitance on the bus
V_{sw}	Bus switching voltage
$\overline{A}_{sw}$	Average bus switching activity
W	Data bus width
B	Cache block size
S	Cache size
E_{dec}	Memory decode energy
E_{rw}	Memory read/write energy
E_{ref}	Memory refresh energy
V_m	Memory supply voltage
I_{ref}	Memory refresh current

Table 1: Technology parameters

form of the these equations, referred to relative energy variations, are described in the following using the symbols introduced. The processor energy is modeled as:

$$\Delta E_{cpu} = T_{ck} \cdot \overline{CPI} \cdot \overline{P}_{cpu} \cdot \Delta I \quad (1)$$

The contribution of system bus to energy variation is:

$$\Delta E_{bus} = \tfrac{1}{2} \cdot C_{tot} \cdot V_{sw}^2 \cdot W \cdot (N_{data} + N_{inst}) \quad (2)$$
$$N_{data} = \overline{A}_{sw,data} \cdot B_{data} \cdot \Delta DM \quad (3)$$
$$N_{inst} = \overline{A}_{sw,inst} \cdot B_{inst} \cdot \Delta IM \quad (4)$$

Finally, the energy memory model adopted is:

$$\Delta E_m = \Delta E_{m,data} + \Delta E_{m,inst} + \Delta E_{m,ref} \quad (5)$$
$$\Delta E_{m,data} = (E_{dec} + E_{rw} \cdot B_{data}) \cdot \Delta DM \quad (6)$$
$$\Delta E_{m,inst} = (E_{dec} + E_{rw} \cdot B_{inst}) \cdot \Delta IM \quad (7)$$
$$\Delta E_{m,ref} = T_{ck} \cdot V_m \cdot \overline{I}_{ref} \cdot \overline{CPI} \cdot \Delta I \quad (8)$$

4. Case Studies

In this section, two case studies are reported: Loop unrolling and Loop fusion. For each transformation, the source code parameters and the model equations are reported and discussed.

4.1. Loop Unrolling

Loop unrolling is a parametric transformation whose results in terms of energy reduction are influenced by the *unrolling factor* U, i.e. the number of times the

loop body is replicated to build the modified loop. The parameter U, thus, completely defines the transformation. The effects of loop unrolling clearly depend also on the characteristics of the source code being transformed. Such properties are captured by the set of *source code parameters* reported in Table 2.

Symbol	Meaning
LI, LS	Loop instructions, size (bytes)
LBI, LBS	Loop body instructions, size (bytes)
N	Loop iterations

Table 2: Source code parameters for loop unrolling

The number of instructions of the original loop is:

$$I_o = N \cdot LI \tag{9}$$

The transformed loop executes $N_t = \lfloor N/U \rfloor$ iterations and $LI_t = LI + (U-1) \cdot LBI$ instructions per iteration. Therefore, the total number of instructions executed by the transformed loop is:

$$I_t = N_t \cdot LI_t = \lfloor N/U \rfloor \cdot [LI + (U-1) \cdot LBI] \tag{10}$$

The instructions gain obtained with unrolling is thus:

$$\Delta I = \frac{\lfloor N/U \rfloor \cdot [LI + (U-1) \cdot LBI] - I_o}{I_o} \tag{11}$$

The transformation has also effects on the number of instruction cache misses due to the increased dimension of the loop body. A more accurate analysis leads to the results—summarized below—that show a non-linear dependence of the number of misses on the relative values of the loop size LS and the instruction cache size S_{inst}[1]. Three significant cases have been identified:

- $LS \leq S_{inst}$ — In this case there are no capacity misses since the entire loop code can be loaded into the cache. Hence, there are only cold misses, during the first iteration. The number of instruction cache misses is thus:

$$IM = \lceil LS/B_{inst} \rceil \tag{12}$$

- $S_{inst} < LS < 2 \cdot S_{inst}$ — In this case capacity misses also take place. The number of cold misses is the same as in the previous case, but, in addition, for every successive iterations, there are $2 \cdot \lceil (LS \,\%\, S_{inst})/B_{inst} \rceil$ capacity misses. Therefore, the total number of misses is:

$$IM = \lceil LS/B_{inst} \rceil + 2 \cdot (N-1) \cdot \left\lceil \frac{LS \,\%\, S_{inst}}{B_{inst}} \right\rceil \tag{13}$$

- $LS \geq 2 \cdot S_{inst}$ — The number of misses in every iteration equals the number of cold misses, i.e.:

$$IM = N \cdot \lceil LS/B_{inst} \rceil \tag{14}$$

For all these cases, the relevant figure is the variation of the number of instruction cache misses $\Delta IM = IM_t - IM_o$. Such difference depends on the variation of number of instructions due to the transformation:

$$\Delta LS = LS_t - LS_o = (U-1) \cdot LBS \tag{15}$$

and must be calculated for all the $3^2 = 9$ cases. It is worth noting that since the transformed code will always be larger than the original one, only 6 out of the 9 cases are significant. For the sake of conciseness, only the two limiting cases are described in the following.

- $(LS_o \leq ICS) \wedge (LS_t \leq ICS)$ — In this case both the original and the transformed code completely fit into the cache and thus only cold misses take place. The variation, recalling Equation (12) is:

$$\Delta IM = \lceil \frac{LS_o}{B_{inst}} \rceil - \lceil \frac{LS_t}{B_{inst}} \rceil \approx \left\lceil \frac{(U-1) \cdot LBS}{B_{inst}} \right\rceil$$

- $(LS_o \geq 2 \cdot ICS) \wedge (LS_t \geq 2 \cdot ICS)$ — In this other limiting case, both codes are larger than the double of the cache size and thus each instruction fetch causes a miss. Recalling Equation (14), the instruction miss variation is:

$$\Delta IM = N_t \cdot \left\lceil \frac{LS + (U-1) \cdot LBS}{IBS} \right\rceil - N_o \cdot \left\lceil \frac{LS}{IBS} \right\rceil$$

In a similar manner and referring to Equations (12)–(14), the variations for the other four cases can be calculated. The last effect to be considered is the variation of data cache misses. Since the transformation does not modify the data access pattern of the code, the term ΔDM can be assumed to be 0, at least at a first approximation. The three contributions ΔI, ΔIM and ΔDM can now be fed to the technology models to derive the overall energy saving.

4.2. Loop Fusion

This transformation has the purpose of combining into a new single loop the bodies of different subsequent loops. Some constraint must be satisfied, in particular the loops to be fused need to have the same iteration range and the statements in their bodies must be independent. The only *transformation parameter* characterizing loop fusion is the number of loop to be merged FN. The *source code parameters* that influence the effect of this transformation are all those considered for

1 The loop size and number of instructions are linearly related assuming a fixed instruction size.

loop unrolling (Table 2) plus the number and size of control instructions, defined as:

$$LCI = LI - LBI \quad (16)$$
$$LCS = LS - LBS \quad (17)$$

In the following the subscript $k \in [1, NF]$ is used to indicate a specific loop among those to be fused. An additional useful parameter is the average number of control instructions over all the considered loops:

$$\overline{LCI} = 1/NF \cdot \sum_{k=1}^{NF} LCI_k \quad (18)$$

Using the symbols just introduced, the number of instructions in the original and transformed codes are:

$$I_o = N \cdot \sum_{k=1}^{NF} (LBI_k + LCI_k) \quad (19)$$
$$I_t = \overline{LCI} + N \cdot \sum_{k=1}^{NF} LBI_k \quad (20)$$

The variation ΔI is thus given by:

$$\Delta I = (NF - 1) \cdot \overline{LCI} / \sum_{k=1}^{NF} LI_k \quad (21)$$

assuming that $\overline{LCI} = LCI_k \quad \forall k$. To study the effect of loop fusion with respect to instruction misses, the same cases considered for loop unrolling and expressed by Equations (12)–(14) turn out to be applicable. Nevertheless, when considering the original code composed of NF loops, the number of instruction misses must be estimated for each single loop according to the three mentioned equations and then summed over all loops. On the other hand, the estimates for the transformed code can be obtained by simply substituting LS with the overall transformed code size LS_t, defined as:

$$LS_t = \overline{LCS} + \sum_{k=1}^{NF} LBS_k \quad (22)$$

According to equations 12)–(14) and referring to the original code sizes $LS_{o,k}$ and the transformed code size LS_t, the number of instruction misses of the original loops $IM_{o,k}$ and the transformed one IM_t can be derived. The resulting overall variation is thus:

$$\Delta IM = IM_t - \sum_{k=1}^{NF} IM_{o,k} \quad (23)$$

It is worth noting that the number of possible cases derived from the limiting conditions on the cache size is, in general, $3 \cdot 3^{NF}$. Similar considerations, not reported here for the sake of conciseness, apply to the estimation of data cache misses. Since in most cases the different loops operate on different arrays, data misses tend to be increased, the best-case condition being that all data fit into the cache in which case the number of misses will approximately be invariant.

5. Experimental results

This section reports the results obtained with the proposed models in the estimation of ΔI and ΔIM for two transformations. Similar results have been obtained for other transformations listed in Section 2.

5.1. Loop Unrolling Results

The transformation has been applied on a test code and the number of instruction and instruction misses have been measured both on the original and transformed code and compared with the corresponding estimated figures. Figure 2 shows the predictions of ΔI and of ΔIM for different instruction cache sizes. The proposed models show a very good accuracy and are thus suited for energy gain estimation.

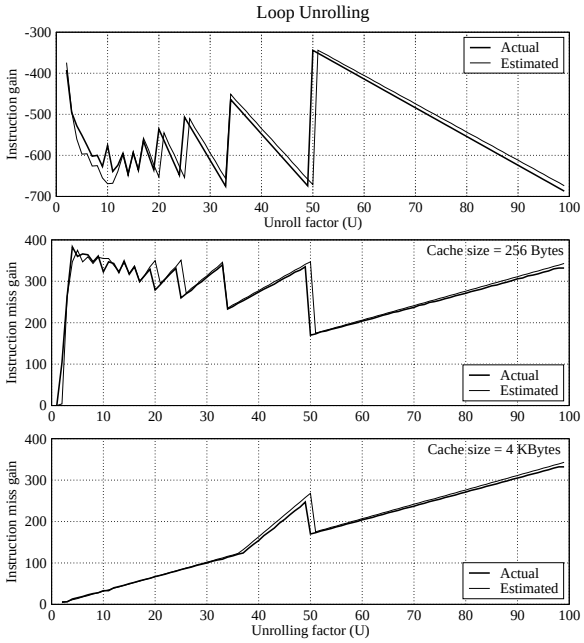

Figure 2: Loop unrolling: ΔI, ΔIM

5.2. Loop Fusion Results

A similar procedure has been applied for loop fusion with $NF = 2$ and the results for instruction misses are shown in Figure 3, where the x axis is an index related to the loop body size ratio. Again the accuracy obtained is more than satisfactory. It is worth noting that similar results have been obtained for other transformations as well. The results presented here have been selected since they refer to the most critical cases, i.e. the class of loop transformations.

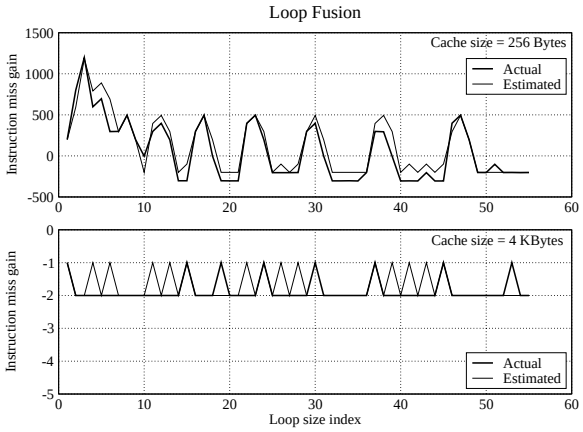

Figure 3: Loop fusion: ΔIM

5.3. Energy Estimation Results

The estimates of ΔI, ΔIM and ΔDM, combined with the energy models (see Section 3.2) adopted to account for the technology-dependent parameters, lead to a new set of results showing the accuracy of the complete methodology in terms of energy reduction estimation. The complete models for different transformations have been tested on a set of SPEC95 benchmarks in order to quantify the energy improvement estimation error. The actual energy gain has been obtained by simulating both the original and the transformed code and then compared with the estimated gain derived from the models presented in this paper. Experiments have been performed for four architectures based on different processors(strongARM, PowerPC, microSPARC and MIPS) using third-party power profiling tools(SimplePower, sim-outorder, SpixTools). Each benchmark has been analyzed varying both the instruction cache size (S_{inst}) and the input data and *all* compatible transformations have been applied in a proper sequence using the predicted optimal values for their parameters (unroll factor, tile size, etc.). Table 3 collects the relative eeror between the estimated gain ΔE_{est} and the actual value ΔE_{act} derived from simulation. The results confirm that the models are reliable since they can correctly predict both energy reductions and undesirable energy increases. In conclusion, the average estimation error has shown to be around than 3%.

6. Conclusions

This paper presented a methodology and a set of related models for the estimation of the energy consumption variation deriving from the application of source-to-source code transformations. The main motivation of such a work is that source code transformations are very difficult to automate and a tedious work to be performed by hand. This enforces the need for an estimation methodology that allows evaluating the potential benefits of a code transformation without actually modifying the code. The proposed methodology is based in a first, technology independent phase and a second technology dependent step. The results reported show the accuracy of the models and the viability of the approach.

	FIB		FIR		WAVE	
S_{inst}	$\epsilon_{\%}$	$\sigma^2_{\%}$	$\epsilon_{\%}$	$\sigma^2_{\%}$	$\epsilon_{\%}$	$\sigma^2_{\%}$
256	+4.16	3.96	n/a	n/a	-1.63	1.20
512	+7.18	4.02	-3.67	4.48	-1.82	1.15
1K	+3.31	1.49	-2.11	4.95	-3.93	1.51
2K	-1.42	2.15	+1.03	7.68	-0.53	1.59
4K	-2.08	1.91	-11.24	7.57	+0.03	1.60
Average	3.63	2.71	4.51	6.17	1.59	1.41

Table 3: Gain estimation relative errors

References

[1] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high performance computing. *Technical Report N. UCB/CSD-93-781, University of California at Berkeley*, 1993.

[2] L. Benini and G. De Micheli. System-level power optimization: Techniques and tools. *Transactions on Design Automation of Electronic Systems*, 5:115–192, 2000.

[3] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. The impact of source code transformations on software power and energy consumption. *Journal of Circuits, Systems and Computers*, 11(5):477–502, 2002.

[4] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. Library functions timing characterization for source-level analysis. *Conference on Design Automation and Testing in Europe*, pages 1132–1133, March 2003.

[5] F. Catthoor, H. De Man, and C. Hulkarni. Code transformations for low power caching in embedded multimedia processors. *Proc. of IPPS/SPDP*, pages 292–297, 1998.

[6] M. S. Lam. Software pipelining: An effective scheduling technique for vliw machines. *SIGPLAN Conference on Programming Language Design and Implementation*, pages 318–328, 1988.

[7] M. S. Lam, E. E. Rothberg, and M. E. Wolfe. The cache performance and optimization of blocked algorithms. *Conference on Architectural Support for Programming Languages an Operating Systems*, pages 63–74, 1991.

[8] M. J. Wolfe. More iteration space tiling. *ACM Proceedings of Supercomputing*, pages 655–664, 1989.