Ultra Low-Power Electronics and Design

Edited by

Enrico Macii
Politecnico di Torino,
Italy

KLUWER ACADEMIC PUBLISHERS
BOSTON / DORDRECHT / LONDON

A C.I.P. Catalogue record for this book is available from the Library of Congress.

ISBN 1-4020-8075-1 (HB)
ISBN 1-4020-8076-X (e-book)

Published by Kluwer Academic Publishers,
P.O. Box 17, 3300 AA Dordrecht, The Netherlands.

Sold and distributed in North, Central and South America
by Kluwer Academic Publishers,
101 Philip Drive, Norwell, MA 02061, U.S.A.

In all other countries, sold and distributed
by Kluwer Academic Publishers,
P.O. Box 322, 3300 AH Dordrecht, The Netherlands.

Printed on acid-free paper

Printed in the Netherlands.

Contents

Chapter 9

SOURCE-LEVEL MODELS FOR SOFTWARE POWER OPTIMIZATION

Carlo Brandolese, William Fornaciari and Fabio Salice
Politecnico di Milano

Abstract This chapter presents a methodology and a set of models supporting energy-driven source-to-source transformations. The most promising code transformation techniques have been identified 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 discussed, showing the value of the proposed approach as a support tool for embedded software design.

Keywords: Software optimization, Power optimization, Source-level modeling

9.1 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 many 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 chapter proposes a sound 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.

9.2 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*, *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 [1, 2] or studied in the present work, are summarized in the following. Particular attention must be devoted to loop transformations [3–6] since most of the execution time of a program 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. The transformation impacts on energy in two ways: on one hand, it reduces loop overhead by performing less compare and branch instructions; on the other hand, it allows the compiler for better optimization and register usage in the larger loop body.

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 sub-loops 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 action of distribution, i.e. merging, by reducing loop overhead, increasing instruction parallelism, improving register, data cache, TLB or page locality. It also improves the load balance of parallel loops.

Loop interchange exchanges the position of two loops in a loop nest, generally moving one of the outer loops to the innermost position. It is one of the most valuable transformations and can improve performance in

many ways: it enables and improves vectorization, increases data access locality and increases the number of loop-invariant expressions in the inner loop.

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.

Loop unswitch is applied when a loop contains a branch with a loop-invariant test condition. The loop is then replicated inside each branch of the conditional, saving the overhead of conditional branching inside the loop, reducing the code size of the loop body, and possibly enabling the parallelization of one or both branches.

The second class collects a number of data-structure and memory access transformations [7, 6].

Local to global array promotion allows compilers to use simpler addressing modes since global arrays address does not depend on the stack pointer.

Scratch-pad array introduction has the goal 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 [7] 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 [7, 8].

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.

9.3 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, leads to poor or no energy reduction, while, when applied to a pre-transformed code its effectiveness is greatly increased. Thus sequences of transformations should be considered, rather than single transformations. For this reason it is crucial 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 thus leading to a quick estimate of the expected benefits.

9.3.1 Conceptual Flow

Figure 9.1 shows the conceptual scheme of the estimation flow. The source code is processed and its relevant 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.

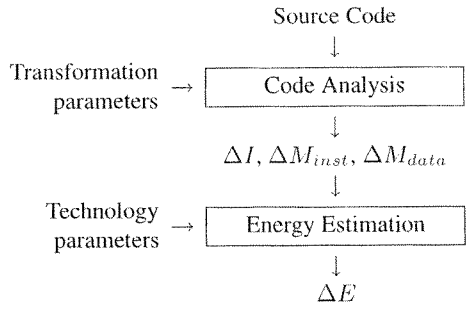

Figure 9.1. Phases of the methodology flow

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, ΔM_{inst} and ΔM_{data} expressing the variations of number of instructions executed, of number of instruction cache misses and of number of data cache 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.

9.3.2 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 above 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 adopted for the CPU, the cache, the bus and the main memory are summarized in Table 9.1.

Table 9.1. Technology parameters

Symbol	Meaning
T_{ck}	CPU clock period
$\overline{CPI}$	Average CPI[1]
$\overline{P}_{cpu}$	Average CPU power
C_{tot}	Total capacitance on the bus
V_{sw}	Bus switching voltage
$\overline{A}_{sw}$	Average bus switching activity
W	Bus width

Symbol	Meaning
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
$\overline{I}_{ref}$	Average memory refresh current

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

$$\Delta E_{cpu} = T_{ck}\overline{P}_{cpu}\overline{CPI}\Delta I \tag{3.1}$$

The contribution of system bus to energy variation ΔE_{bus} is:

$$\Delta E_{bus} = \tfrac{1}{2}C_{tot}V_{sw}^2(\Delta N_{bus,addr} + \Delta N_{bus,data} + \Delta N_{inst}) \tag{3.2}$$

where:

$$\Delta N_{bus,addr} = \overline{A}_{sw,addr}W_{addr}(\Delta M_{data} + \Delta M_{inst}) \tag{3.3}$$

$$\Delta N_{bus,data} = \overline{A}_{sw,data}W_{data}B_{data}\Delta M_{data} \tag{3.4}$$

$$\Delta N_{bus,inst} = \overline{A}_{sw,inst}W_{data}B_{inst}\Delta M_{inst} \tag{3.5}$$

Finally, the adopted memory model expresses the energy variation ΔE_m as:

$$\Delta E_m = \Delta E_{m,data} + \Delta E_{m,inst} + \Delta E_{m,ref} \tag{3.6}$$

where:

$$\Delta E_{m,data} = (E_{dec} + E_{rw}B_{data})\Delta M_{data} \tag{3.7}$$

$$\Delta E_{m,inst} = (E_{dec} + E_{rw}B_{inst})\Delta M_{inst} \tag{3.8}$$

$$\Delta E_{m,ref} = T_{ck}V_m\overline{I}_{ref}\overline{CPI}\Delta I \tag{3.9}$$

9.4 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.

9.4.1 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 9.2.

Table 9.2. Source code parameters for loop unrolling

Symbol	Meaning
LI	Number of loop instructions
LS	Size of loop instructions (bytes)
LBI	Number of loop-body instructions
LBS	Size of loop-body instructions (bytes)
N	Loop iterations

The number of instructions of the original loop is:

$$I_o = N \cdot LI \tag{4.1}$$

The transformed loop executes $N_t = \lfloor N/U \rfloor$ times and:

$$LI_t = LI + (U-1)LBI \tag{4.2}$$

instructions per iteration. Therefore, the total number of instructions executed by the transformed loop is:

$$I_t = N_t \cdot LI_t = \left\lfloor \frac{N}{U} \right\rfloor \cdot [LI + (U-1)LBI] \tag{4.3}$$

The instructions gain obtained with unrolling is thus:

$$\Delta I = \left\lfloor \frac{N}{U} \right\rfloor \cdot [LI + (U-1)LBI] - I_o \tag{4.4}$$

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 in the following—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}[2]. 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:

$$M_{inst} = \left\lceil \frac{LS}{B_{inst}} \right\rceil \tag{4.5}$$

- $S_{inst} < LS < 2S_{inst}$
  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 additional iterations, there are $2\lceil (LS \mod S_{inst})/B_{inst} \rceil$ capacity misses. Therefore, the total number of misses is:

$$M_{inst} = \left\lceil \frac{LS}{B_{inst}} \right\rceil + 2(N-1) \left\lceil \frac{LS \mod S_{inst}}{B_{inst}} \right\rceil \tag{4.6}$$

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

$$M_{inst} = N \left\lceil \frac{LS}{B_{inst}} \right\rceil \tag{4.7}$$

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

$$\Delta LS = LS_t - LS_o = (U-1)LBS \tag{4.8}$$

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 boundary 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 (4.5), is:

$$\Delta M_{inst} = \left\lceil \frac{LS_o}{B_{inst}} \right\rceil - \left\lceil \frac{LS_t}{B_{inst}} \right\rceil \approx \left\lceil \frac{(U-1)LBS}{B_{inst}} \right\rceil \tag{4.9}$$

- $(LS_o \geq 2ICS) \wedge (LS_t \geq 2ICS)$
  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 (4.7), the instruction miss variation is:

$$\Delta M_{inst} = N_t \left\lceil \frac{LS + (U-1)LBS}{IBS} \right\rceil - N_o \left\lceil \frac{LS}{IBS} \right\rceil \tag{4.10}$$

[2] The loop size and number of instructions are linearly related assuming a fixed instruction size.

In a similar manner and referring to Equations (4.5)–(4.7), 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 ΔM_{data} can be assumed to be 0, at least at a first approximation. A first validation can be performed at this level comparing the dimensionless estimated figures ΔI and ΔM_{inst} with those derived from simulation. Figure 9.2 shows the results for the variation of number of instruction executed. It is worth noting that ΔI does not depend on the cache size but only on the structure of the code and the effectiveness of the optimizations that the compiler can exploit on the modified loop.

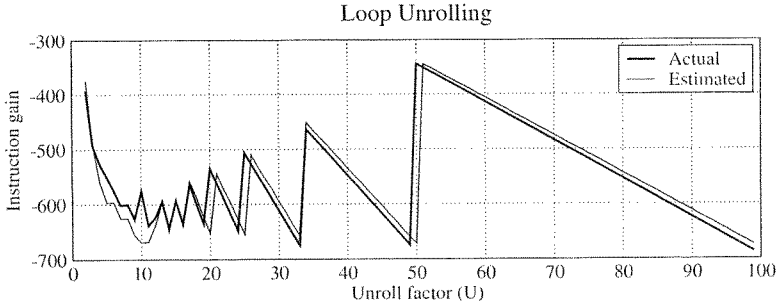

Figure 9.2. Loop unrolling: ΔI

As far as the variation of instruction cache misses, different scenarios have been considered by varying the cache size from 256 to 4096 bytes. Table 9.3 summarizes the results obtained by averaging the estimation error over the interval $U = [2; 100]$ and Figure 9.3 shows the two boundary cases.

Table 9.3. Loop unrolling: ΔM_{inst} average error and standard deviation

$S_{inst}(bytes)$	$\overline{\epsilon}_{\%}$	$\overline{\sigma}_{\%}$
256	-1.881	8.026
512	-2.557	7.101
1024	-2.531	6.910
2048	-2.750	9.252
4096	-1.691	5.065

The two contributions ΔI and ΔM_{inst} (remembering that $\Delta M_{data} = 0$) can now be fed to the technology models to derive the overall energy saving. Table 9.4 reports the average error and the corresponding standard deviation in terms of energy gain for the five cache-size scenarios just considered.

These results show that the model tends to underestimate the potential gain deriving from loop unrolling. A possible reason is that unrolling a loop leads to a

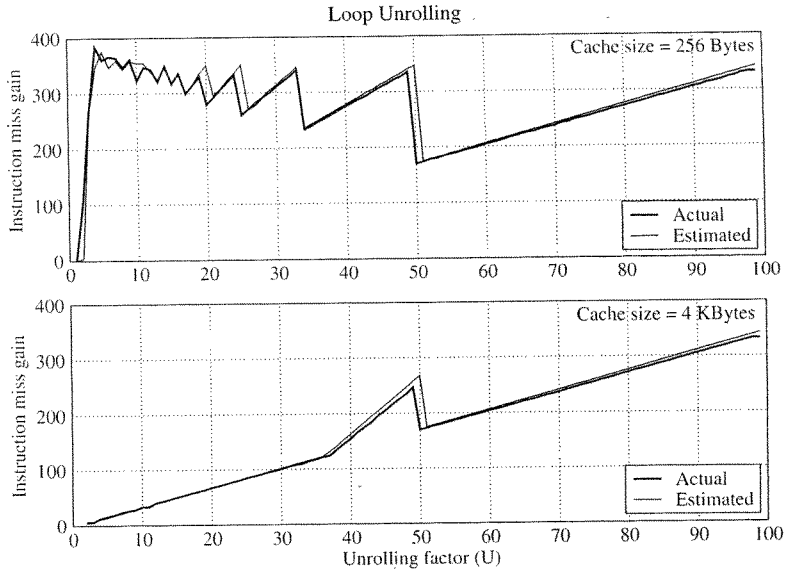

Figure 9.3. Loop unrolling: ΔM_{inst}

Table 9.4. Loop unrolling: ΔE average error and standard deviation

$S_{inst}(bytes)$	$\overline{\epsilon}_{\%}$	$\overline{\sigma}_{\%}$
256	-1.754	9.144
512	-4.552	7.322
1024	-7.663	6.966
2048	-6.203	5.777
4096	-4.409	3.011

longer loop body, i.e. a larger basic block where the compiler can better perform optimizations. Despite the light biasing of the model, the overall average error is, in absolute value, approximately 4.9% and this can be considered more than satisfactory when operating at source code level.

9.4.2 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 merged 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 NF of loops to be merged. The *source code parameters* that influence the effect of this transformation are

all those considered for loop unrolling (see Table 9.2) plus the number and size of control instructions, defined as:

$$LCI = LI - LBI \tag{4.11}$$
$$LCS = LS - LBS \tag{4.12}$$

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:

$$\overline{LCI} = \frac{1}{NF} \sum_{k=1}^{NF} LCI_k \tag{4.13}$$

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

$$I_o = N \sum_{k=1}^{NF} (LBI_k + LCI_k) \tag{4.14}$$
$$I_t = N(\overline{LCI} + \sum_{k=1}^{NF} LBI_k) \tag{4.15}$$

The variation ΔI is thus given by:

$$\begin{aligned} \Delta I &= N(\overline{LCI} + \sum_{k=1}^{NF} LBI_k - \sum_{k=1}^{NF} (LBI_k + LCI_k) = \\ &= N(\overline{LCI} - \sum_{k=1}^{NF} LCI_k) \end{aligned} \tag{4.16}$$

Assuming that $\overline{LCI} = LCI_k \ \ \forall k$ yields:

$$\sum_{k=1}^{NF} LCI_k = \sum_{k=1}^{NF} \overline{LCI} = NF \cdot \overline{LCI} \tag{4.17}$$

and thus Equation (4.16) can be rewritten as:

$$\Delta I = N(\overline{LCI} - \sum_{k=1}^{NF} LCI_k) = N(1 - NF)\overline{LCI} \tag{4.18}$$

To study the effect of loop fusion with respect to instruction misses, the same cases considered for loop unrolling and expressed by Equations (4.5)–(4.7) 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:

$$LS_t = \overline{LCS} + \sum_{k=1}^{NF} LBS_k \tag{4.19}$$

According to Equations (4.5)–(4.7) 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:

$$\Delta M_{inst} = IM_t - \sum_{k=1}^{NF} IM_{o,k} \tag{4.20}$$

It is worth noting that the number of possible cases derived from the limiting conditions on the cache size is, in general, 3^{NF+1}. Similar considerations 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. A validation procedure similar to that used for loop unrolling has been applied for loop fusion also, considering the simplest and most common case where $NF = 2$. To analyze the behavior of the transformation, loops with different body sizes have been considered and the results for instruction misses are shown in Figure 9.5, where the x axis is an index related to the loop body size ratio. For the same combinations of loop body sizes and for an instruction cache size varying from 256 to 4096 bytes, the gain in terms of instruction misses have also been estimated and compared with actual results, leading to the data collected in Table 9.5 and the graphs of Figure 9.5 relative to the two limiting cases.

Again the accuracy obtained is more than satisfactory since the average absolute error is approximately 2.1% with very low standard deviation. Combining dimensionless figures with the energy models of the different component of the considered system led to the energy estimates. Such estimates show a very limited error, as reported in Table 9.6, and are not biased. It is though worth noting that the reported results refer to loops manipulating very small arrays for which the hypothesis of being fully contained in the data cache may be assumed to hold. This translates into the models by assuming $\Delta M_{inst} = 0$.

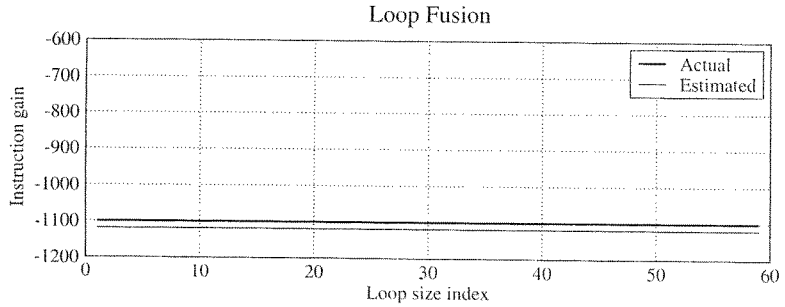

Figure 9.4. Loop fusion: ΔI

Table 9.5. Loop fusion: ΔM_{inst}

$S_{inst}(bytes)$	$\overline{\epsilon}_{\%}$	$\overline{\sigma}_{\%}$
256	+2.423	2.701
512	+3.004	2.804
1024	-3.150	4.253
2048	+0.153	1.672
4096	-0.258	1.419

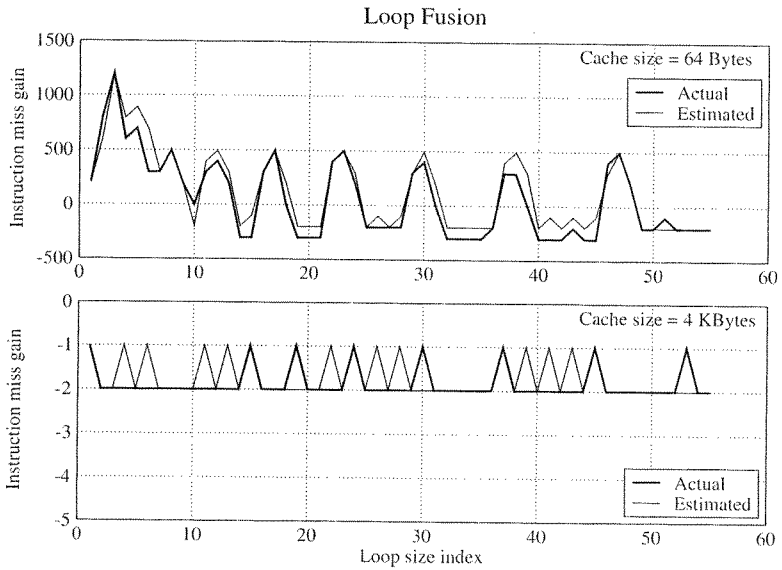

Figure 9.5. Loop fusion: ΔM_{inst}

More complex cases show higher errors but preliminary experimental results suggest that a 10–15% error is a reasonable and conservative upper bound.

Table 9.6. Loop fusion: ΔE

$S_{inst}(bytes)$	$\overline{\epsilon}_{\%}$	$\overline{\sigma}_{\%}$
256	+1.945	3.882
512	+0.177	3.469
1024	-0.194	3.916
2048	+1.592	2.425
4096	+0.168	0.017

9.5 EXPERIMENTAL RESULTS

The estimates of ΔI, ΔM_{inst} and ΔM_{data}, combined with the energy models (see Section 9.3.2) 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 (ΔE) estimation. The models for 5 transformations have been tested on a set of SPEC95 benchmarks in order to quantify the energy gain 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. Experiments have been performed on four architectures based on different processors and operating systems using third-party timing and/or power profiling tools (see Table 9.7).

Table 9.7. Operating environments for validation

Processor	Operating system	Simulation engine
Intel strongARM	Linux RedHat 9.0	SimpleScalar 3.0 / SimPAnalyzer
IBM PowerPC 405	Linux RedHat 9.0	SimpleScalar 3.0
Sun microSPARC II EP	Solaris 8	SpixTools
MIPS Tech. MIPS-32	Linux RedHat 9.0	SimpleScalar 3.0

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 9.8 collects the relative error 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%.

Table 9.8. Energy gain estimation relative errors

	FIB		FIR		WAVE-1		WAVE-2		IIR	
S_{inst}	$\overline{\epsilon}_{\%}$	$\overline{\sigma}_{\%}$	$\overline{\epsilon}_{\%}$	$\overline{\sigma}_{\%}$	$\overline{\epsilon}_{\%}$	$\overline{\sigma}_{\%}$	$\overline{\epsilon}_{\%}$	$\overline{\sigma}_{\%}$	$\overline{\epsilon}_{\%}$	$\overline{\sigma}_{\%}$
256	+4.16	3.90	n/a	n/a	-1.97	2.81	+4.29	3.63	-1.63	1.20
512	+7.18	4.02	-3.67	4.48	-1.83	2.67	+4.63	3.52	-1.82	1.15
1024	+3.31	1.49	-2.11	4.95	-2.87	3.51	+4.81	0.79	-3.93	1.51
2048	-1.42	2.15	+1.03	7.68	-2.37	3.71	+4.20	0.57	-0.53	1.59
4096	-2.08	1.91	+11.25	7.57	-1.86	3.71	+3.74	0.20	+0.03	16.00
Average	3.63	2.69	4.51	6.17	2.18	3.28	4.33	1.74	1.58	4.29

9.6 CONCLUSIONS

The presented work has addressed the problem of the fast estimation of the effects induced by a set of specific source code transformations by using a structured methodological approach based on technology-independent models. In particular, the presented analysis flow, by providing an appropriated set of both technological and transformation parameters, allows the designer to an *a priori* evaluation of the impact of a specific transformation and/or the effect of a sequence of interdependent transformations. Two specific transformations have been accurately described: loop unrolling and loop fusion. As far as loop unrolling is concerned, it has been shown that the proposed model can be considered more than satisfactory since the average error between the estimated gain and the simulated gain is, approximately, 4,9% with a low standard deviation. Concerning loop fusion, the model has produced estimates—for a wide set of technological options—displaying an average absolute error of 2,1% with an high level of reliability. Both the methodology and the models has been validated on a set of benchmarks showing an overall average error of the estimated energy gain around 3%. This result is more than satisfactory and confirms that the models of the different transformation are sufficiently accurate and the methodology, though subject to further improvements, is promising.

References

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

[2] 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.

[3] 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.

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

[5] 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.

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

[7] 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.

[8] 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.