View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Archivio istituzionale della ricerca - Politecnico di Milano

Combining target-independent analysis with dynamic profiling to build the performance model of a DSP

Marco Lattuada, Fabrizio Ferrandi

Politecnico di Milano - Dipartimento di Elettronica ed Informazione
Via Ponzio, 34/5 – 20133 – Milano (Italy)
{lattuada,ferrandi}@elet.polimi.it

Abstract

Fast and accurate performance estimation is a key aspect of heterogeneous embedded systems design flow, since cycle-accurate simulators, when exist, are usually too slow to be used during design space exploration. Performance estimation techniques are usually based on combination of estimation of the single processing elements which compose the system. Architectural characteristics of Digital Signal Processors (DSP), such as the presence of Single Instruction Multiple Data operations or of special hardware units to control loop executions, introduce peculiar aspects in the performance estimation problem.

In this paper we present a methodology to estimate the performance of a function on a given dataset on a DSP. Estimation is performed combining the host profiling data with the function GNU GCC GIMPLE representation. Starting from the results of this analysis, we build a performance model of a DSP by exploiting the Linear Regression Technique. Use of GIMPLE representation allows to take directly into account the target-independent optimizations performed by the DSP compiler. We validate our approach by building a performance model of the MagicV DSP and by testing the model on a set of significative benchmarks.

1. Introduction

In the last years there has been a considerable growing in the performance request in the embedded systems. To satisfy this growth, multiprocessor heterogeneous systems have become the de-facto standard in the embedded systems increasing the complexity of the design flow. Indeed, the use of this type of architectures introduces new problems in the design flow such as component selection and applications partitioning and mapping [1]. Nevertheless, the time-to-market of embedded systems has not been extended to allow to treat accurately these new problems. On the contrary, a faster design flow is required by manufacturers to make products available on the market as soon as possible. This acceleration of the design process imposes to use very fast heuristics to speed-up the design space exploration potentially impoverishing the quality of the design results. There are some critical analyses which in any case can not be too approximated. One of them is the analysis of the application overall performance since meeting the performance constraints is a mandatory objective for the design flow. Evaluation of the performance during the different design steps of an application by directly measuring it onto the target architecture is not always possible: final target platform, some of its components, or part of its compiler tool chain can be not available during applications development. For example, final platform can be not yet ready because still in design while compiler tool chain use can be limited by license or host compatibility problems. To address these problems, we need to estimate the execution time of the overall application instead of measuring it. Moreover, considering the current requests of an embedded system design flow, we have to provide a fast estimation technique but which provides results enough accurate to correctly drive the design exploration without cutting good solutions from the design space.

To perform the estimation of the overall application we need to estimate its single task on the different processing elements, since most of the methodologies [2, 3] for estimating the overall performance of a system are based on combination of single component performance estimation techniques. The methodology presented in this paper focuses on the performance estimation of *Digital Signal Processors (DSP)*. DSPs are microprocessors specialized for real time computing of digital signals, so they are mainly focused on data intensive applications and this purpose reflects on their architecture [4]. For example, they implement Single Instruction Multiple Data operations, special arithmetic operations such as Multiply and Accumulate and special hardware units to control loop executions.

Independently by the architectural characteristics of a DSP, two are the possible scenarios of use of a DSP in an embedded platform. In the first scenario, the DSP is controlled by a General Purpose Processors which uses it as a co-processor [5, 6]. In the second scenario, DSP is the master component of the architecture and executes directly all the application or at least it controls the computation flow [7]. The estimation methodology presented in this paper is

focused on the first scenario since is much more frequent in the Multi Processor System on Chip architectures [8]. The same methodology can be adopted also in the second scenario, but it requires that the target platform (or its simulator) allows to measure the performance of the single functions.

There are two possible strategies to predict the execution time of a function on a DSP: use a simulator or build a mathematical model. The first method can not always be adopted because a cycle-accurate simulator of the target DSP can not be available. Moreover, simulation time can be too long making using of the simulator unsuitable during the tuning of the platform and of the applications.

To speed up simulation, Stolberg et al. [2] propose a methodology which integrates simulation and analysis. In this methodology the application is decomposed in core tasks whose execution time is supposed to be known. These data are then combined with information extracted from analysis of the application to produce the estimation of the whole application. The main disadvantage of this technique is that it requires the execution time of each core present in the application.

An hybrid approach which combines simulation and analysis is also proposed by Gao et al. [3]. Simulation is used to estimate execution time of library functions and the first run of a particular control path (sequence of instructions executed from the starting of a function to its ending) on the DSP. Execution times of the General Purpose Processors are estimated using a linear model built by hand, while successive runs of paths on the DSP are estimated using the data measured during the first runs. This technique assumes that different executions of the same code on different data are performance equivalent. This assumption holds only if the considered DSP has not a data cache. Moreover, it requires to measure directly the execution time of each executed paths, so input data which activate each path have to be computed.

The second strategy consists of the creation of a mathematical performance model of the target DSP. There are two possible ways to build such type of models: by hand or by using automatic techniques. The first way is applicable when the target DSP architecture is well known. For example, So et al. [9] propose a method for estimating the benefits in the simultaneous execution of different threads on a DSP. The mathematical model they propose is based on the scheduling of the assembly operations of the threads under analysis and requires a deep knowledge of the DSP architecture.

Use of GNU GCC [10] Intermediate Representation for DSP performance analysis has been proposed in [11]. Pegatoquet et al. extract the RTL representation from a modified version of the GNU GCC, but they do not consider it suitable for a DSP performance analysis: the used load-store model does not always describe correctly the memory access model of a DSP. For this reason the authors translate this representation in Directed Acyclic Graphs and estimate the application performance starting from them. Since the RTL representation is target dependent, new translation rules for each considered target DSP have to be added by the designer.

The automatic techniques build performance models by using statistical data modeling techniques. In [12], Cavazos et al. describe a methodology based on Neural Networks to predict the effect on the application performance when a combination of optimizations is applied. For each new function considered, it requires to run four different transformed versions of the application on the real platform or on a cycle-accurate simulator.

The methodology proposed in this paper belongs to the last class of estimation techniques presented. It automatically builds a performance model of a DSP by exploiting linear regression technique. The linear models, although among the simplest performance models, can model quite accurately the performances of the miscellaneous architectures of the DSP and, thanks to the intelligibility of their parameters, allow the designer to easily compare the performance models of different DSPs [13]. The model is built using a set of numeric features extracted from the GIMPLE representation [14] of the application combined with host profiling information. GIMPLE is a language used as intermediate representation by the GNU GCC. Use of the GIMPLE representation extracted at the end of the middle-end flow allows to take into account the effects of the target-independent optimizations performed by the DSP compilers. Indeed, even if GNU GCC compiler targets mostly RISC processors, most of the optimizations which it applies during the target-independent optimization flow (e.g.: copy propagation, common subexpression elimination, dead code elimination) are also applied by most of the DSP compilers [12]. An analysis performed on a lower level target dependent representation would provide more accurate results. However, most of the times this type of analysis can not be applied since DSP compilers are usually not open source nor allow easy access to their internal intermediate representation.

The main contributions of this paper can be summarized as follows:

- it proposes a methodology for fast and automatic building of performance estimation models for DSPs without a deep knowledge of architectural details;
- it proposes a methodology which can directly predict the effect of some of the optimizations performed by the DSP compiler.

The paper is organized as follows: Section 2 describes the methodology, Section 3 shows the experimental results of the application of the methodology while Section 4 draws some conclusions and outlines some future works.

2. Methodology

The flow of the proposed methodology is composed by the following phases:

1. *Performance Model Building*: during this phase the Performance Model represented by a linear function f is built; the phase is composed by three main steps:
   (a) *Function Analysis* (Section 2.1): a set of functions is analyzed to extract a set of numeric features which characterize them from the performance point of view;
   (b) *Preprocessing* (Section 2.3): these numeric features are preprocessed;
   (c) *Linear Regression Analysis* (Section 2.2): the preprocessed data are used as input of the the Linear Regression technique to build the model;
2. *Performance Model Application*: given a new function, the built model is used to estimate its execution time; the steps which compose this phase are:
   (a) *Function Analysis* (Section 2.1);
   (b) *Preprocessing* (Section 2.3);
   (c) *Estimation computation* (Section 2.4): the data produced in previous steps are used as input of function f to compute the estimation.

The following Sections describe in more details the phases which have just been presented.

2.1 Function Analysis

One of the key aspects of the proposed methodology is how to analyze a function to extract information representative of its performance aspects. Indeed, the significance of this information will reflect on the quality of the produced estimation. Moreover, the definition of this analysis has to take into account that the same procedure will be also used during Performance Model Application, so it must have not particular requirements nor take too much time.

First of all, the analysis should consider the dynamic aspects (e.g., the branch probabilities, the number of loop iterations). Most of the times this type of information can not be easily provided with good accuracy by a pure static analysis, but it can be produced by running an instrumented version of the function on the host machine.

The other important aspect which characterizes a function analysis is the abstraction level at which it is performed. Possible abstraction levels go from source code level to assembly level. The last level is the most precise, but it is not suitable for the aims of this methodology because of its requirements. Assembly level analysis requires not only the availability of a compiler for the target architecture, but also of a simulator which satisfies these two conditions: it should provide accurate profiling information to be used in the analysis (e.g., operations or paths counters) and its simulation speed should be comparable with the execution of the application on the host processor.

On the opposite, the analysis performed at the source code level does not require target compiler and simulator,

```
1  int function(int a, int b, int * c,
2     int * d)
3  {
4     int counter, mult, i;
5     for(i = 0; i < 10; i++) {
6         mult = a * b;
7         counter += c[i];
8         counter -= d[i];
9     }
1      return counter + mult;
11 }
```

Figure 1. Example of function to be analyzed

but produces numeric features which are less representative of the performance of the considered function since it does not take into account the optimizations performed by the target compiler at all. The performance models built starting from this type of information can be not enough accurate to be used without the risk of driving design space exploration towards bad solutions. This can lead to a poor quality design or to the increasing of the design time.

We choose to perform analysis at an intermediate level: the GIMPLE representation [14]. In particular we use the intermediate representation extracted at the end of the middle-end flow, during which all the selected target-independent optimizations have been applied. Extracting the intermediate representation at this point allows to produce numeric features which can better predict the impact of the target independent optimizations performed by the DSP compiler on the function.

Let us consider the example presented in Figure 1 and suppose that the DSP compiler applies the loop-invariant code move optimization (move outside the loop operations whose result does not depend on the particular iteration) during the compilation. In this case only a single multiply will be executed during each execution of the function. In Table 1 results of analysis performed at the source and at the GIMPLE level are reported. The source level analysis detects the execution of 10 multiplications. On the other hand, the analysis performed at GIMPLE level correctly detects the execution of a single multiplication.

In the GIMPLE flow the optimizations selected should be all the ones that also the DSP compiler applies. If we have no information about the optimization flow of the DSP compiler, we can try different optimizations sets such as the ones of the GNU GCC optimization levels.

The extracted numeric features consist mainly of dynamic counters of GIMPLE operations. Since these data can depend on the dataset on which the function is executed, the analysis has to be repeated for each given dataset. The analysis proceeds as follows: each GIMPLE operation is associated with the corresponding C operation from which it has been generated. Combining this mapping and the profiling information obtained running the benchmark on the host,

Table 1. The numeric features extracted from the example of Figure 1

	Source code	GIMPLE	
Feature	Original value	Original value	Preprocessed value
for	10	10	0.188
read:int	20	20	0.377
minus-plus:int	21	21	0.396
mult:int	10	1	0.019
return	1	1	0.019

we associate to each operation its dynamic frequency. The operations are then clustered according their operator (e.g., plus, mult) and their data type (e.g., integer or floating). Further clustering can be performed on operations which can be considered equivalent from the performance point of view. In the example, *plus* and *minus* operations have been clustered. Example of the numeric features extracted are shown in Table 1.

The other numeric features concern the presence and the type of loop constructs in the function code. In the GIMPLE language there is not an explicit representation of the loop conditional constructs since they are replaced by combinations of labels, conditional and unconditional jumps. Starting from these operations and from the Control Flow Graph of the function, loops present in the source code are identified. A special operation is associated with each identified loop and it is counted dynamically as other operations exploiting host profiling. These special operations can be of two types: *while*, which identifies the iteration of a generic loop, and *for*, which identifies the iteration of a for loop. This information can be very significant in the estimation since many DSP optimizations are based on the presence of loops and in particular on the presence of for loops.

2.2 Linear Regression Analysis

Regression analysis identifies a set of mathematical techniques to analyze numerical data for modeling the relationships among a variable Y (*dependent variable*) and one or more variables X_i (*independent variables*). In particular we use the Linear Regression Technique which describes this relationship using a linear function f of the independent variables X_i, of the corresponding parameters β_i and of an error term ϵ, treated as a random variable:

$$Y = f(\overline{X}, \overline{\beta}, \epsilon) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_k X_k + \epsilon \quad (1)$$

This analysis can be used for several purposes: in the proposed methodology we exploit it to predict the DSP functions execution times. Note that we are assuming that the execution time of a function can be approximated with a linear combination of some numerical characteristics of the function itself.

To compute the parameters β_i, a set of data, called *training set*, representative of the prediction problem, has to be provided as input of the technique. In particular, for each element of the set, the values of the independent and of the dependent variables have to be provided.

When Regression Analysis is exploited to perform estimation, the dependent variable Y of Equation 1 corresponds to the variable that we want to estimate. In our estimation problem, the function execution time would seem to be a good candidate to be the dependent variable Y. Given this choice, the variables X have to be some numeric features extracted from the function analysis and representative of its overall performance. A possible choice of the variables X is the counters of the different types of operations.

2.3 Preprocessing

The use of function execution time as dependent variable Y as initially proposed in the previous Section has to be avoided since it introduces a bias in the model because of the very large range (several orders of magnitude) of this variable. How a too wide range of the dependent variable can affect the goodness of the model can be easily seen on an example. Let us consider two observed data $V_1 = 50$ and $V_2 = 2000$ and their predicted values $\overline{V_1} = 100$ and $\overline{V_2} = 2,100$ and let us analyze which is the estimation error introduced. The standard linear regression technique minimizes the *Mean Squared Error* MSE:

$$MSE = \frac{1}{n}\sum_{i=1}^{n} e_i^2 = \frac{1}{n}\sum_{i=1}^{n} \left(V_i - \overline{V_i}\right)^2 \quad (2)$$

where V_i is the observed data and $\overline{V_i}$ is the predicted data. The square error in first case is smaller than in the second ($2,500$ vs. $10,000$), so the technique will consider much more significant the error introduced in the second estimation. On the opposite, the goodness of a performance estimation model is usually evaluated considering its *Mean Relative Error* [3, 11, 12]. According to this criterion, the estimation error produced in the first case is much more significant (relative error is 2) than the one of the second case (0.05). This consideration can be generalized: the regression technique is biased to better predict the longest functions in spite of shortest ones.

To reduce the range of the dependent variable and so to reduce this bias problem, we have to preprocess the data. Preprocessing of the data is quite common in Regression Analysis. Which transformations (e.g., Standardization [15], Logarithmic Transformation [16]) are applied depends on the characteristics of the problem. In our case, for each function, all the data are divided by the overall sum of dynamic counters of GIMPLE operations. This transformation not only reduces the range of the depedent variable, but also preserves the linear relationship between independent variables and it. In this way we have a prediction problem with these characteristics: the dependent variable Y is the average execution time per operation and the independent variables X_i are the numeric features computed during Function Analysis

divided by their overall sum. In this process, the loop special operations previously introduced are considered as normal GIMPLE operations.

The result of the application of the Linear Regression Technique to the preprocessed numeric features is the computation of the values of the parameters β_i. At the end of this step, given the numeric features X_i and the overall sum of GIMPLE operation S we obtain the performance model for predicting the Average Execution Time per-operation AC:

$$AC = f(\overline{N}) = B_0 + B_1 N_1 + B_2 N_2 + ... + B_k N_K \quad (3)$$

where B_i are constant coefficients and $N_i = \frac{X_i}{S}$

2.4 Estimation computation

Last phase of the methodology flow is the estimation computation. The function, whose execution time we want to predict, is analyzed as it has been described in Section 2.1 and the results are preprocessed as it is described in the previous Section.

Since the aim of the methodology is the estimation of the function execution time ($Time$), the dependent variable AC has to be multiplied by the number of GIMPLE operations S. So, we have:

$$\begin{aligned} Time =& AC \cdot S \\ =& (B_0 + B_1 N_1 + B_2 N_2 + ... + B_k N_K) \cdot S \quad (4) \\ Time =& \left(B_0 + B_1 \frac{X_1}{S} + B_2 \frac{X_2}{S} + ... + B_N \frac{X_N}{S} \right) \cdot S \\ =& (B_0 \cdot S) + \sum_{i \in F} B_i \cdot X_i \quad (5) \end{aligned}$$

Rewriting Equation 4 into Equation 5 has two consequences: the preprocessing is not more needed (X_i variables are used instead of N_i) and the overall number of operations S is treated as a numeric feature (with coefficient B_0).

3. Experimental Evaluation

We validate the proposed methodology building the performance model of the MagicV DSP processor. MagicV is a 1 GFLOP floating point DSP with 40-bit precision developed by Atmel [17] integrated with an ARM926EJ-S on the Atmel DIOPSIS 940HF chip [6]. The operating system installed on the system is Linux 2.6.24. We initially describe our experimental setup and then we present the experimental results.

3.1 Experimental Setup

The proposed methodology has been integrated in PandA [18], a Hardware/Software codesign framework based on the GNU GCC compiler developed at Politecnico di Milano. It exploits three existing components of the framework: the GIMPLE analyzer, the edge profiler and the automatic code instrumenter.

GIMPLE representation is extracted from the GNU GCC and it is analyzed to produce the numeric features as described in Section 2.1. The framework is able to control which optimizations the GNU GCC applies before producing the GIMPLE used by the methodology. The profiling information are obtained by running on the host machine the function code instrumented using a modified version of the Edge Path Profiling [19].

During the model building phase, the real execution times of the functions are measured directly on the board. These measures are performed automatically by generating instrumented code that exploits a free running hardware timer of the chip. The modified source code is then compiled with the Chess compiler from Target Compiler [20]. It should be pointed out that the optimization performed by this DSP compiler are not under the designer's control. Each executable is run ten times to mitigate the measures perturbation introduced by other process executed by the operating system. Then, the framework collects all the measures of the execution time of a function and tries to identify the outliers data (perturbed runs) by considering values that differ more than the standard deviation from the mean. These outliers are removed from the set and the real measure is computed as the average of the remaining values.

To perform the linear regression, we use RapidMiner (formerly YALE [21]), an open-source java based tool for knowledge discovery and data mining.

3.2 Experimental Results

We validate the methodology using two benchmarks suites for embedded systems: the DSP Stone [22] and the Powerstone [23] suites. We analyze the benchmarks to identify all the functions which can be executed on the DSP: we withdraw some of them because of the restrictions imposed by the DIOPSIS tool chain. The extracted functions are listed in the left part of Table 2.

Since we have no information about which optimizations the Chess compiler applies, we build different performance models guessing the optimizations performed. The considered performance models differ for the target independent optimizations applied to the GIMPLE representation. In particular we choose the optimizations sets corresponding to the GNU GCC optimization level: O0 (*do not optimize*), O1 (*optimize*), O2 (*optimize even more*).

We verify the accuracy of the produced models through the *cross-validation* technique on the benchmark set. Table 3 shows the average error and the standard deviation for each model.

The most accurate model is based on the numeric features extracted using the O1 set of optimizations (Mean Relative error is 23.05%). This model is almost twice more accurate than the model produced without any optimizations (Mean Relative error of O0 model is 40.72%). On the other hand, the model built using the O2 optimizations set is less accurate. This can be motivated by the fact that the O2 optimizations set models less accurately the set of optimizations performed by Chess. From this consideration and from the fact that O2 optimizations set is a superset of O1, we can

Table 2. The estimation error of the model O1

Benchmark	Function	Real Exec. Time (μs)	Estim. Error %
DSP Stone Fixed Point			
adpcm	adapt_quant	$2.193 \cdot 10^5$	11.1
adpcm	adpt_predict_1	$7.255 \cdot 10^5$	12.1
adpcm	coding_adjustment	$2.371 \cdot 10^5$	12.6
adpcm	encoder	$3.368 \cdot 10^6$	9.4
complex_multiply	main	$1.905 \cdot 10^4$	7.3
complex_update	main	$1.899 \cdot 10^4$	38.3
convolution	main	$1.890 \cdot 10^4$	38.8
dot_product	main	$1.736 \cdot 10^4$	16.2
fft_16	fft_bit_reduct	$5.118 \cdot 10^4$	29.9
fft_16	fft_inpsca	$4.562 \cdot 10^4$	47.5
fft_1024	fft_bit_reduct	$1.680 \cdot 10^6$	39.6
fft_1024	fft_inpsca	$1.277 \cdot 10^6$	39.0
fir	main	$2.193 \cdot 10^4$	3.0
fir2dim	main	$5.085 \cdot 10^4$	5.1
iir_biquad_N_sections	main	$2.174 \cdot 10^4$	4.6
iir_biquad_one_section	main	$1.958 \cdot 10^4$	29.6
lms	main	$2.157 \cdot 10^4$	24.2
mat1x3	main	$1.634 \cdot 10^4$	44.2
matrix1	main	$6.521 \cdot 10^4$	9.9
matrix2	main	$6.411 \cdot 10^4$	3.3
n_complex_updates	main	$2.408 \cdot 10^4$	36.5
n_real_updates	main	$2.009 \cdot 10^4$	16.6
real_update	main	$5.104 \cdot 10^4$	12.4
	Average	$2.229 \cdot 10^5$	21.4
DSP Stone Floating Point			
complex_multiply	main	$1.716 \cdot 10^4$	22.1
complex_update	main	$1.908 \cdot 10^4$	45.6
convolution	main	$2.075 \cdot 10^4$	2.8
dot_product	main	$1.752 \cdot 10^4$	27.3
fir	main	$2.371 \cdot 10^4$	27.2
fir2dim	main	$4.398 \cdot 10^4$	7.8
iir_biquad_N_sections	main	$2.337 \cdot 10^4$	1.2
iir_biquad_one_section	main	$1.848 \cdot 10^4$	43.6
lms	main	$2.344 \cdot 10^4$	40.2
mat1x3	main	$1.773 \cdot 10^4$	38.5
matrix1	main	$6.683 \cdot 10^4$	11.7
matrix2	main	$6.411 \cdot 10^4$	11.0
n_complex_updates	main	$2.155 \cdot 10^4$	16.6
n_real_updates	main	$2.080 \cdot 10^4$	16.0
real_update	main	$1.732 \cdot 10^4$	20.9
	Average	$2.772 \cdot 10^4$	22.2
Power Stone			
blit	blit	$4.081 \cdot 10^5$	14.6
bcnt	main	$2.393 \cdot 10^4$	2.5
compress	cl_hash	$2.010 \cdot 10^5$	0.1
engine	engine	$6.893 \cdot 10^6$	0.0
engine	interpolate	$7.884 \cdot 10^6$	0.0
g3fax	rowout	$1.887 \cdot 10^6$	3.5
pocsag	find_syndromes	$3.605 \cdot 10^5$	7.1
pocsag	num_proc	$1.245 \cdot 10^5$	10.5
pocsag	alpha_proc	$2.460 \cdot 10^5$	13.8
pocsag	normalized_locator	$2.784 \cdot 10^5$	7.4
v42	putcode	$1.282 \cdot 10^7$	0.2
	Average	$2.829 \cdot 10^6$	5.4

Table 3. Cross-validation accuracy of the built models

	Relative Error	
Model	Mean	Standard Deviation
O0	40.72%	15.60%
O1	23.05%	15.94%
O2	34.34%	18.65%

infer that some of the O2 exclusive optimizations are not performed by Chess compiler

In the rest of experiments we concentrate on the O1 optimizations set model, which proved to be the most accurate. Moreover, we do not apply the cross-validation process to deeply analyze the accuracy of the predictions. Detailed results are reported in Table 2. The mean relative error (18.0%) in this case is smaller since we are now using all the benchmarks set.

The maximum error introduced in prediction is on functions of the *fft* benchmarks and in particular on the function *fft_inspca* with 47.5%. These functions are characterized by a large transfer of data performed at the beginning and at the end of each function execution. The methodology does not extract this type of information during the function analysis, so the performance model does not correctly estimate them. A second aspect which the analysis does not catch is the high number of memory accesses of these functions.

Considering the average error on the whole suites, the error introduced on the Power Stone benchmarks is quite small (5.4% with the maximum of 14.6 on the *blit* function) with respect to the DSP Stone benchmarks (21.4% and 22.2%). The different average error is due to the fact that Power Stone benchmarks have longer and more complex functions. The effects of a single optimization can be relatively much more significant on a small function than on a large one, because in the first case it can modify the whole function while it is quite difficult that a single optimization modifies completely a complex or big function. As a consequence, the error of not correctly estimating the effect of an optimization is relatively much more significant for small functions.

Finally, we compare the results of our methodology with the ones produced by the works described in Section 1. The technique presented in [12] can estimate the performance effects of one or more transformations on a code with an error of 7.3%, but requires at least four runs of the application on target platform. For this reason, if the target platform is not available during the application design, this technique can not be adopted. Moreover, even if a cycle-accurate simulator is available, it can be too slow to be used during the design space exploration where several estimations can be required. On the other hand, the methodology we propose does not need neither to use a simulator nor the target platform during estimation. Use of a simulator during estimation is required also by the Cross Reply technique presented in [3] to measure the execution time of the first runs of the different parts of the function. Moreover, the authors do not report the results of the estimation of the execution time of the single DSP tasks.

Techniques presented in [11] and [9] both produce very accurate results (error is about 2.0% for the first method and few cycles for the second). Unfortunately these techniques are applicable only to DSPs of which all architectural details are known, since estimation is produced by directly schedul-

ing the assembly operations of the function on the real architecture. This limitation can impact very heavily on the results of the design of the systems, since it restricts the component selection to a limited set of the available DSPs.

Average prediction error of the methodology proposed by Stolberg et al. [2] is of 9.2%, but their methodology requires to have the execution time of all the kernels of the analyzed applications. For this reason, it can not be applied, if the execution time of a kernel is not known or if a kernel is modified during the design flow.

4. Conclusions

In this paper we presented a methodology for building the performance model of a DSP by combining GIMPLE analysis, profiling information and linear regression techniques. The advantages of using this methodology in a design flow are: it does not limit the choice of the components of the target architecture, since the designer does not need a deep knowledge of the DSP and of its tool chain, and it is enough fast and accurate to be efficiently used during design space exploration to drive the designer towards good quality solutions.

The proposed methodology has been validated on a real DSP processor, the MagicV processor from ATMEL, on two different benchmark suites. The results show how our approach is able to take into account some of the optimization performed by the DSP compiler and produces a model with an average relative error of 18.0%.

Future works will improve two aspect of the proposed methodology: modelization of memory accesses and optimizations selection. To overcome the first problem, we will improve the GIMPLE analysis to retrieve information which better models the memory accesses from the performance point of view. The second improvement will consist of adding an automatic technique to select which optimizations have to be applied before the GIMPLE analysis.

References

[1] P. Chandraiah and R. Doemer. Designer-controlled generation of parallel and flexible heterogeneous mpsoc specification. In *DAC '07*, pages 787–790, New York, NY, USA, 2007. ACM.

[2] H. J. Stolberg, M. Bereković, and P. Pirsch. A platform-independent methodology for performance estimation of multimedia signal processing applications. *J. VLSI Signal Process. Syst.*, 41(2):139–151, 2005.

[3] L. Gao, K. Karuri, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr. Multiprocessor performance estimation using hybrid simulation. In *DAC '08*, pages 325–330, New York, NY, USA, 2008. ACM.

[4] E. J. Tan and W. B. Heinzelman. Dsp architectures: past, present and futures. *SIGARCH Comput. Archit. News*, 31(3):6–19, 2003.

[5] J. C., K. Cyr, S. de Gregorio, J.-P. Giacalone, J. Webb, and Y. Masse. Open multimedia application platform: enabling multimedia applications in third generation wireless terminals through a combined risc/dsp architecture. *ICASSP '01*, 2:1009–1012, 2001.

[6] MagicV VLIW DSP and Diopsis, http://www.atmel.com/products/diopsis/.

[7] T. J. Lin, H. Y. Lin, C. M. Chao, C. W. Liu, and C. W. Jen. A compact dsp core with static floating-point unit & its microcode generation. In *GLSVLSI '04*, pages 57–60, New York, NY, USA, 2004. ACM.

[8] W. Wolf, A.A. Jerraya, and G. Martin. Multiprocessor system-on-chip (mpsoc) technology. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(10):1701–1713, Oct. 2008.

[9] W. So and A. G. Dean. Reaching fast code faster: using modeling for efficient software thread integration on a vliw dsp. In *CASES '06*, pages 13–23, New York, NY, USA, 2006. ACM.

[10] GNU Compiler Collection. GCC, version 4.3, http://gcc.gnu.org/.

[11] A. Pegatoquet, E. Gresset, M. Auguin, and L. Bianco. Rapid development of optimized dsp code from a high level description through software estimations. In *DAC '99*, pages 823–826, New York, NY, USA, 1999. ACM.

[12] J. Cavazos, C. Dubach, F. Agakov, E. Bonilla, M. F. P. O'Boyle, G. Fursin, and O. Temam. Automatic performance model construction for the fast software exploration of new hardware designs. In *CASES '06*, pages 24–34, New York, NY, USA, 2006. ACM.

[13] Rafael Saavedra-Barrera, Alan J. Smith, and Eugene Miya. Machine characterization based on an abstract high-level language machine. *SIGMETRICS Perform. Eval. Rev.*, 18(3):24, 1990.

[14] L. J. Hendren, C. Donawa, M. Emami, G. R. Gao, Justiani, and B. Sridharan. Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations. In *LCPC '93*, pages 406–420, 1993.

[15] M. P. Allen. *Regression analysis with standardized variables*, chapter 10, pages 46–50. Springer US, 1997.

[16] S. Weisberg. *Applied Linear Regression*. Probability and Statistics. Wiley, March 2005.

[17] Atmel corporation. http://www.atmel.com.

[18] PandA framework, http://trac.ws.dei.polimi.it/panda.

[19] T. Ball and J. R. Larus. Efficient path profiling. In *MICRO-29*, pages 46–57, 1996.

[20] Target Compiler Technologies. CHESS/CHECKERS, http://www.retarget.com.

[21] I. Mierswa, M. Wurst, R. Klinkenberg, M. Scholz, and T. Euler. Yale: Rapid prototyping for complex data mining tasks. In Lyle Ungar, Mark Craven, Dimitrios Gunopulos, and Tina Eliassi-Rad, editors, *KDD '06*, pages 935–940, New York, NY, USA, August 2006. ACM.

[22] V. živojnović, Juan M. Velarde, C. Schläger, and H. Meyr. DSPSTONE: A DSP-oriented benchmarking methodology. In *ICSPAT '94*, 1994.

[23] A. Malik, B. Moyer, and D. Cermak. A low power unified cache architecture providing power and performance flexibility (poster session). In *ISLPED '00*, pages 241–243, New York, NY, USA, 2000. ACM.