Performance Modeling of Embedded Applications with Zero Architectural Knowledge

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

ABSTRACT

Performance estimation is a key step in the development of an embedded system. Normally, the performance evaluation is performed using a simulator or a performance mathematical model of the target architecture. However, both these approaches are usually based on the knowledge of the architectural details of the target.

In this paper we present a methodology for automatically building an analytical model to estimate the performance of an application on a generic processor without requiring any information about the processor architecture but the one provided by the GNU GCC Intermediate Representation. The proposed methodology exploits the linear regression technique based on an application analysis performed on the Register Transfer Level internal representation of the GNU GCC compiler. The benefits of working with this type of model and with this intermediate representation are three: we take into account most of the compiler optimizations, we implicitly consider some architectural characteristics of the target processor and we can easily estimate the performance of portions of the specification. We validate our approach by evaluating with cross-validation technique the accuracy and the generality of the performance models built for the ARM926EJ-S and the LEON3 processors.

Categories and Subject Descriptors

C.4 [**Computer System Organization**]: Performance of Systems

General Terms

Algorithms,Design,Performance

Keywords

Performance Estimation, profiling, GNU GCC

1. INTRODUCTION

Timings constraints are among the most critical aspects of the embedded systems design flow. Verifying these constraints only in the last phases of the design process is not possible: fixing the system in the last design process stages can require significant changes of the system itself delaying the final system production of an unsustainable amount of time. For this reason, system performance has to be evaluated from the early design steps.

The spreading of the MultiProcessor System on Chip (MPSoC) platforms in the embedded systems has increased the difficulty of estimating their performance. Difficulties arise not only from the presence in a system of multiple components, the interactions of which have to be taken into account, but also from their heterogeneity. This heterogeneity consists not only of the presence of different classes of components in the system (e.g., GPP, DSP, FPGA), but also of the availability of different components for a given class of processing elements. Since the hardware of an embedded system is not fixed, often the system designer has to select which components will be included into the final system and have to guarantee that these components will satisfy timing, power and economic constraints. Judging which are the more appropriate components is not a trivial task: in principle the designer has to evaluate the performance of each available candidate on the different portions of the application which are supposed to be mapped on it. Evaluation can be hardened by the fact that, since hardware and software are usually co-designed [23], multiple partitionings and mappings of the application may have to be considered and so the designer has to evaluate the performance of several arbitrary pieces of code.

The methods to early evaluate the performance of a processing element can be mainly divided into three categories: direct measures, estimations by simulation, estimations by use of mathematical models. Most of the time the first solution is not affordable since the real components are not available during these phases of the design and integrating the measurement on the real component in a Design Space Exploration framework can be a difficult task. For these reasons, techniques based on estimations have to be preferred. In the simulation based ones estimation is achieved by simulating the behavior of the component on a particular code by running a Cycle Accurate Simulator. Finally in the last category techniques, the estimation is obtained by exploiting mathematical models which correlate some numerical features of an application with its performance. In general, they are less accurate than the simulator based, but they are much faster.

Most estimation techniques present the same disadvantage: they require the designer to have some knowledge of the architecture of the target component (typically larger in case of simulation based techniques) to guarantee accurate estimation. This requirement, which could be rationale when the designer dealt with a single or few processing elements, can be not satisfiable anymore since the available processing element candidates are too many to be threated with such type of techniques.

In this paper, we present a performance estimation methodology focused on the estimation of application performance on arbitrary embedded processor. The proposed methodology does not require any knowledge of the target processor by the embedded system designer, but it exploits the information about the target processor provided by the GNU GCC compiler [12] internal representation. For this reason, the processor has to be targeted by the GNU GCC compiler. The performance models are built combining the linear regression technique with an analysis of operation sequences performed at the GNU GCC Register Transfer Language (RTL) level. RTL representation has two peculiar properties: the language itself is target-independent, but the descriptions of the applications written in this language differ according to the considered target processor. Thanks to the former characteristic, we do not need to specialize the analysis to support different processors, thanks to the latter we can still perform a target-dependent analysis without requiring direct knowledge of the architecture. Moreover, the operation sequences allow the analysis to catch also complex architectural aspects such as the pipeline.

To the best of our knowledge, all the other solutions in literature work at a somewhat higher (virtual instructions) or lower (assembly code) abstraction levels. So, the formers are not able to capture many significant architectural aspects and compiler optimizations, or require complex translations to gather at least some of them. With the latter, instead, it is necessary to know in detail the Instruction Set of the target architecture for generating the annotated specification. We resort to the use of a linear model because our main objective is to perform an unbiased analysis among different intermediate representations. We also show that our regression technique is quite accurate and comparable to non-linear models.

The remainder of this paper is organized as follows. Section 2 discusses the related work in the field of performance prediction, focusing mainly on embedded system design. Section 3 introduces the use of linear regression for performance estimation. Section 4 describes the proposed methodology, detailing the application analysis, the model building and the model utilization. Section 5 provides the experimental evaluation and finally Section 6 concludes the paper.

2. RELATED WORK

Several methodologies for building mathematical performance models of a processor have been proposed. Roughly, in most of these methodologies we can identify three different steps: identification of some numerical features significant of the application performance (e.g., number of times an operation appears in the code or is executed), extraction of these features from the applications with a static or a dynamic analysis (e.g., by profiling the application on the host), building of the performance model by correlating these features with the application performance. The parameters of these performance models can be built mainly in two different ways: by hand or by exploiting automatic methods.

The first method requires a deep knowledge of the architectural characteristics of the considered processor. For example Brandolese et *al.* [7] use the concept of “atoms” to describe basic elements of the source code. The key idea is that the performance of an application can be modeled as the sum of the estimated execution delays of each atom. The estimated execution delay of an atom is expressed as the sum of two contributions: a reference timing, that accounts for all the deterministic aspects in ideal conditions, and a statistical deviation that depends on the architectural aspects and on the compiler. The limits of this methodology reside in the complexity to get the reference times for the atoms and in particular their statistical deviations.

Also Beltrame et *al.* [5] propose a flexible approach to estimate the performance of applications on superscalar architectures based on combination of multiple contributions for each operation. In this approach the Cycles Per Instruction of each assembly instruction are modeled as the combination of three contributions: a fixed term for the nominal execution delay of the operation, a statistical term accounting for the stall overheads associated and a parallelism coefficient which models how much of the theoretical parallelism is exploited. The limit of this methodology consists in the data required for computing the parameters of the model. In fact, the starting and the ending time of each instruction of the execution trace of an application are needed.

Hwang et *al.* [14] work at higher granularity: they propose an estimation technique that takes in input C application processes and performs a basic block analysis. The basic blocks are annotated with execution delay estimated considering processor models. In particular these models describe the pipeline, the memory hierarchy and branch delays of the processor. This approach is extended to multiple processing elements by wrapping the annotated code with a SystemC wrapper used to model the communications. The generated Transaction Level Model is then compiled and executed natively on the host machine. In this way this solution takes into account multiprocessor issues, but it still requires a quite accurate model of the target processors and works only at the basic block granularity.

Performance models can also be built by exploiting automatic techniques: Meyr et *al.* [15] for example link the processor model generated from an architecture description language (at behavioral or cycle-accurate level) to SystemC based simulation. They integrate a fine-grained software instrumentation tool to obtain performance and memory access statistics. The approach is faster than using an ISS, but it requires the architectural description of the examined processor.

Some methodologies build the model by exploiting regression techniques instead of tuning parameters by hand or starting from an architectural model; however they may still require knowledge of the target platform or at least of its assembler. For example, Lajolo et *al.* [16] exploit the GCC compiler by performing a timing analysis on the assembly code and then regenerate the C code with the timings annotations related to the assembly instructions. Execution times of the assembly instructions on the target processor are obtained from the machine description files of GCC. The regenerated C code is then compiled on the host and run to obtain an estimation of the performance of the program. This approach is at a lower level than our RTL representation, but the regeneration of the original C code requires the knowledge of the Instruction Set for each desired target processor since the produced C source code has to be functionally equivalent to the assembly.

The approach of Oyamada et *al.* [19, 20] is also based on the instruction set of the target processor, but adopts another non-linear method, based on neural networks. Also in this case, for a given target processor, it is necessary to employ not only the trained neural network, but also the compiler and the methods to extract the dynamic assembly instructions count of the application. Moreover, nonlinear models may estimate more accurately software performance in some specific cases, but they make design space exploration more complex: they are usually not additive, and require code rewriting if the designer wants to explore only parts of the code.

Bontempi and Kruijtzer [6] use a non-linear method for the execution delay estimation of the whole program. They perform profiling through IPROF to gather statistics on the execution of the applications on a virtual processor (with a set of 42 virtual instructions). Being based on the profiling feature of the GNU GCC compiler and on its disassembler, IPROF directly collects statistics by running automatically instrumented assembly code. In this way it does not require any particular knowledge of the target processor, but it requires the availability of a processor with the same ISA of the target to run the profiling; this requirement is often not satisfiable. They exploit the lazy learning modeling technique: it builds an estimation function, which may be locally linear, based on the closeness of the application and the training set. This approach also suffers from the lack of architectural details in the estimation due to the use of virtual instructions.

Prediction techniques have also been developed for evaluating the trade-offs of different optimizations during compilation [25]. Two types of prediction models are presented. The firsts estimate how a particular optimization will change the characteristics of the intermediate code. The seconds predict how these changes will affect the performance.

Some techniques, which do not require any knowledge of the target processors, have already been proposed, but they can be adopted only in particular cases or produce not very accurate predictions. Indeed, since they work at high abstraction levels, they do not consider at all compiler optimizations nor architectural characteristics.

An example of this type of methodology is proposed by Suzuki et *al.* [21]. They estimate the execution time on the chosen target processor exploiting a simple additive performance model based on high-level C language statements. The cost of each statement is obtained by executing a set of benchmarks on an ISS and extracting an average cycle count. This work does not consider compiler and architectural features and can be only applied to applications with a very simple structure (no loops, no recursion), so it can not be adopted for estimation of most of the real applications.

Lavagno et *al.* [4] use virtual instructions instead of source instructions for estimation. The source code is compiled to an intermediate representation, with all the representative operations of a RISC processor. The code is then translated back to C with timing annotations that gather statistics on the execution of such virtual instructions, when compiled and run on any host. The cost of each virtual instruction on the target processor is obtained through two different techniques: by deriving the cycle counts from the processor manual or its ISS, or by exploiting linear regression technique. The authors of this work, however, highlight that this approach has some drawbacks: it does not consider all the possible compiler optimizations and use a reduced set of Virtual Instructions to model all the possible instructions of the target processor. To overcome these limitations, the authors resort again to an assembly-level estimation.

Also Giusto et al. [11] exploit a model based on virtual instruction set and linear regression. They, however, conclude that a linear approach based on virtual instructions is accurate only when the applications of the training set are very similar to the ones they want to estimate. Furthermore, approaches that adopt virtual instruction sets do not take in account the architectural characteristics of the processors, such as pipelining.

Generally, the recent literature on analytical models produced with zero knowledge of the target architecture has limited its scope to specific applications or to a limited set of benchmarks, with well defined characteristics. In this work, we try to broaden these approaches, dealing with very heterogeneous classes of applications, different levels of compiler optimizations and different target processors. The objective is to show how we construct a “good performance estimator”, sufficiently accurate to give reasonable suggestions at the initial stages of design space exploration, and sufficiently fast to easily evaluate a large number of design solutions without requiring any knowledge of the target processor.

3. LINEAR REGRESSION FOR PERFORMANCE ESTIMATION

The proposed methodology aims at automatically producing a performance model for estimating the execution time of an application on a generic embedded processor by exploiting the linear regression technique.

Regression analysis refers to mathematical techniques for the analysis of the relationships among data consisting of a dependent variable Y and one or more independent variables X_i. The dependent variable Y is modeled as a function of the independent variables X_i, the corresponding parameters β_i and an error term ϵ, treated as a random variable:

$$Y = f(\overline{X}, \overline{\beta}, \epsilon) \tag{1}$$

Linear regression is a form of regression analysis in which the model function is a linear combination of independent variables:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_k X_k + \epsilon \tag{2}$$

These techniques can be used for several purposes: prediction, inference, hypothesis and testing. In the case of the performance estimation problem, given a generic application, the dependent variable $Cycles$ corresponds to its execution time expressed in clock cycles. The independent variables x_i, instead, are a set of numeric features which well describe the performance characteristics of the specification. We detail this set F in Section 4.1. Since we are considering a linear performance model, we have:

$$Cycles = \beta_0 + \sum_{i \in F} \beta_i \cdot x_i \tag{3}$$

A regression technique is composed of two phases: the model building and the model application. For the first phase regression techniques require a set of benchmarks representative for the considered field. This set, used to develop and tune the regressive model, is called *training set*.

For each benchmark of the training set, the real execution delay in clock cycles has to be provided. It can be measured either by profiling the benchmark directly on the target processor or by running the application on a cycle accurate simulator.

The produced model is then used in the second phase of the methodology to estimate the performance of other applications, not present in the training set. The model takes in input the numerical features of the analyzed application and returns as output the estimated execution time.

Since both the model building and the model application phases use the numerical features, they share the application analysis step where these features are extracted.

4. PROPOSED METHODOLOGY

The following Sections detail the different parts of our methodology. Section 4.1 shows how the numeric features are extracted from an application. Section 4.2 describes how these data, obtained from the applications of the training set, are used to build the model. Finally Section 4.3 shows how the numeric features extracted from a new application and the performance model are used to predict the performance of this new application.

4.1 Application Analysis

One of the key aspects of the proposed methodology is which numerical features are extracted during the application analysis. To obtain a good estimation with a regression technique, it is necessary to extract really meaningful features that describe the performance behavior of the considered application. Nevertheless, in the contest of a fast and portable performance analysis, the significance is not the only characteristic to take into account when choosing the type of analysis to apply.

The characteristics required by the analysis can be summarized as follows:

1. it should take into account as much as possible the characteristics of the target processors, but without requiring the designer to have any knowledge of the architecture itself nor of its Instruction Set;
2. it should be easily extendible to other target processors;
3. it should consider the effects on the performance of the object code due to compile-time optimizations;
4. it should allow the estimation of the performance of arbitrary portions of the program and not only of the whole application;
5. it should take into account the dynamic behavior of the application: a static analysis may produce very poor results since it is very difficult to statically find correlation among source code, input data and performance.

In the following, we will discuss how our analysis, based on RTL operations sequences [8] of GNU GCC, well fits these requirements.

4.1.1 RTL sequences analysis

The GNU GCC compiler is divided into three main components:

- *front-end*: translates the source code in a language independent intermediate representation called GIMPLE [13];
- *middle-end*: performs the target independent optimizations;
- *back-end*: translates the GIMPLE representation into the RTL representation, performs the target dependent optimizations and then convert RTL representation into assembly language.

Figure 1 shows an example of the different representations used by the GNU GCC compiler during compilations targeting an ARM processor and a SPARC-compliant processor. The numbers at the beginning of the lines are used to show the correlation among operations in the different representations. The correlation among source code and GNU GCC internal representations has been computed automatically by a slightly modified version of the GNU GCC compiler, while the correspondence with the assembly operations has been added by hand. The Figure shows how an RTL description consists of a sequence of instructions of type `insn` or `jump_insn`. Each RTL instruction is composed of a combination of RTL operations: an RTL operation is mainly characterized by an operator (e.g., `plus`, `minus`), a data type (e.g., `SI` - Single Precision Integer), some operands (e.g., registers, results of other RTL operations) and annotations. For example the first ARM RTL instruction is composed of a `set` operation which writes in a register (`reg`) the result of a `PLUS` operation executed on a register (`reg`) and on a constant integer (`const_int`). In the rest of the methodology, we take into account only the first two characteristics of an RTL operation (i.e., we ignore the operands and annotations); moreover, for each RTL instruction we consider only its most significant RTL operation. In the example, these operations have been highlighted by using uppercase letters.

The different representations of this small piece of code give an idea of the advantage of using the RTL representation in this type of analysis with respect of others based on higher representations such as GIMPLE or Virtual Instructions. Consider in particular the sum operations which are annotated with indexes `1` and `3` in the source code and their corresponding assembly code. Even if they are apparently performance-equivalent operations (sum of a parameter and of a constant), the first is translated into just a single `add` ARM assembly instructions, the second into one `mov` and three `add` ARM operations. Also during the compiling process targeting SPARC the two sum operations are translated into different assembly operations: in this case, the first one is translated into a single `add` as in the ARM case while the second is translated into a sequence composed of `sethi`, `or` and `add`. An analysis performed at source code level or at GIMPLE level can not forecast these different translations. On the other hand, it is easy to see that the RTL representations provides information similar to the ones provided by assembler in terms of operations really executed. Indeed in the RTL produced during ARM-targeted compilation the two sum operations are represented as a single `plus` (which corresponds to the single `add`) and a sequence of a simple assignment (which corresponds to the `mov`) and of three `plus` respectively. In the RTL produced during SPARC-targeted compilation instead the two sum operations are translated as follows: the first one is translated into a single `plus` (which corresponds to the single `add`) while the second into a sequence of a simple assignment (which corresponds to `sethi`), a `ior` (corresponding to `or`) and a `plus` (corresponding to `add`).

Starting from the RTL description of the application, we consider all its two-length operations sequences instead of considering the single operations. Use of sequences is motivated by the trying of modeling the dependence among the execution time of an operation and the type of the operations which precede it in the computational flow. For example, the execution time required by a sum can be different if it is preceded by a multiply (the couple can be transformed into a Multiply and Accumulate operation), by a conditional jump (additional stalls caused by mis-prediction could be inserted into the pipeline) or by a generic operation. We partition the RTL sequences of the application into classes according to the double pair <*op:type-op:type*> which characterizes each of them. We profile the analyzed application on the host to retrieve information about how many times each execution path (i.e., sequence of Basic Blocks) is executed. Exploiting the correlation among source code operations and GIMPLE operations and then the correlation among GIMPLE operations and RTL operations, we infer how many times each RTL sequence is executed.

For each sequence class <*op:type-op:type*>, we sum the dynamic execution counters of all the RTL sequences of that class. At the end of the process, we obtain the counters that represent how many times the sequences of class <*op:type-operation:type*> have been executed. An example of the sequences extracted from the representation of Figure 1(c) is reported in the left part of Table 1: the first two columns list the identified sequences and their execution counters.

4.1.2 Evaluation of the approach

The RTL sequences based analysis meets the requirements previously listed for the following reasons:

1. RTL representations of the same application for different target processors are different, because during their generation the GNU GCC already considers architectural characteristics of the target processor; moreover the sequences allow to take into account effects on the performance of the interactions of operations executed in order (e.g., effects of the pipeline);
2. the Language is target independent: the analysis can easily be extended to any processors (old and new) supported by GNU GCC;

```
int main() {
   int a, b, c;


1:   b = b + 10;
2:   c = 3 * b;
3:   a = a + 10000000;
4:   return a - c;

}
```

(a) Source Code

```
;; main (){
  int a, b, c;
  int D.774;
<bb 2>:
1:   b_2 = b_1(D) + 10;
2:   c_3 = b_2 * 3;
3:   a_5 = a_4(D) + 10000000;
4:   D.774_6 = a_5 - c_3;
4:   return D.774_6;
}
```

(b) GIMPLE representation

```
1:   (insn 7 6 0 example.c:4 (set (reg/v:SI 134 [ b.4 ])
           (PLUS:SI (reg/v:SI 139 [ b ])
               (const_int 10 [0xa]))) -1 (nil))
2:   (insn 8 7 9 example.c:5 (set (reg:SI 140)
           (REG/v:SI 134 [ b.4 ])) -1 (nil))
2:   (insn 9 8 10 codes_example.c:5 (set (reg:SI 141)
           (ASHIFT:SI (reg:SI 140)
               (const_int 1 [0x1]))))
2:   (insn 10 9 11 example.c:5 (set (reg:SI 142)
           (PLUS:SI (reg:SI 141)
               (reg/v:SI 134 [ b.4 ]))) -1)
2:   (insn 11 10 0 example.c:5 (set (reg/v:SI 135 [ c ])
           (REG:SI 142)) -1 (nil))
3:   (insn 12 11 13 example.c:6 (set (reg:SI 144)
           (CONST_INT 9961472 [0x980000])) -1 (nil))
3:   (insn 13 12 14 example.c:6 (set (reg:SI 145)
           (PLUS:SI (reg:SI 144)
               (const_int 38400 [0x9600]))) -1 (nil))
3:   (insn 14 13 15 example.c:6 (set (reg:SI 143)
           (PLUS:SI (reg:SI 145)
               (const_int 128 [0x80]))))
3:   (insn 15 14 0 example.c:6 (set (reg/v:SI 133 [ a.5 ])
           (PLUS:SI (reg/v:SI 138 [ a ])
               (reg:SI 143))) -1 (nil))
4:   (insn 16 15 0 example.c:7 (set (reg:SI 136 [ D.746 ])
           (MINUS:SI (reg/v:SI 133 [ a.5 ])
               (reg/v:SI 135 [ c ]))) -1 (nil))
4:   (insn 17 16 18 example.c:7 (set (reg:SI 137 [ <result> ])
           (REG:SI 136 [ D.746 ])) -1 (nil))
4:   (JUMP_INSN 18 17 19 example.c:7 (set (pc)
           (label_ref 0)) -1 (nil))
```

(c) RTL representation during ARM-targeted compilation

```
1:   add      r1, r1, #10


2:   mov      r2, r1, asl #1

2:   add      r2, r2, r1


3:   mov      r3, #9961472

3:   add      r3, r3, #38400

3:   add      r3, r3, #128

3:   add      r0, r0, r3

4:   rsb      r0, r2, r0


4:   ldmfd    sp, {fp, sp, pc}
```

(d) ARM Assembler code

```
1:   (insn 7 6 0 example.c:4 (set (reg/v:SI 108 [ b.4 ])
           (PLUS:SI (reg/v:SI 113 [ b ])
                (const_int 10 [0xa]))) -1 (nil))
2:   (insn 8 7 9 example.c:5 (set (reg:SI 114)
           (REG/v:SI 108 [ b.4 ])) -1 (nil))
2:   (insn 9 8 10 example.c:5 (set (reg:SI 115)
           (ASHIFT:SI (reg:SI 114)
                (const_int 1 [0x1]))))
2:   (insn 10 9 11 example.c:5 (set (reg:SI 116)
           (PLUS:SI (reg:SI 115)
                (reg/v:SI 108 [ b.4 ]))) -1)
2:   (insn 11 10 0 example.c:5 (set (reg/v:SI 109 [ c ])
           (REG:SI 116)) -1 (nil))
3:   (insn 12 11 13 example.c:6 (set (reg:SI 118)
           (CONST_INT 9999360 [0x989400])) -1 (nil))
3:   (insn 13 12 14 example.c:6 (set (reg:SI 117)
           (IOR:SI (reg:SI 118)
                (const_int 640 [0x280]))) -1)
3:   (insn 14 13 0 example.c:6 (set (reg/v:SI 107 [ a.5 ])
           (PLUS:SI (reg/v:SI 112 [ a ])
                (reg:SI 117))) -1 (nil))
4:   (insn 15 14 0 example.c:7 (set (reg:SI 110 [ D.774 ])
           (MINUS:SI (reg/v:SI 107 [ a.5 ])
                (reg/v:SI 109 [ c ]))) -1 (nil))
4:   (insn 16 15 17 example.c:7 (set (reg:SI 111 [ <result> ])
           (REG:SI 110 [ D.774 ])) -1 (nil))
4:   (JUMP_INSN 17 16 18 example.c:7 (set (pc)
               (label_ref 0)) -1 (nil))
```

(e) RTL representation during SPARC-targeted compilation

```
1:   add      %i1, 10, %i1


2:   sll      %i1, 1, %g2

2:   add      %g2, %i1, %g2


3:   sethi    %hi(9999360), %g1

3:   or       %g1, 640, %g1

3:   add      %i0, %g1, %i0

4:   sub      %i0, %g2, %i0


4:   restore
4:   jmp      %o7+8
4:   nop
```

(f) SPARC Assembler code

Figure 1: Example of different representations of the function `main` obtained by compiling the source code without optimization. The number at the beginning of the lines in each representation shows the correlation among operations in that representation and operations in the original source code.

	Numerical Features				
	Initial Value	After Normalization	After Main introduction		After clustering
Sequence class			Absolute	Normalized	
ashift:SI-plus:SI	1	0.09	1	0.08	0.08
const_ int:-plus:SI	1	0.09	1	0.08	0.08
main:-plus:SI			1	0.08	0.08
plus:SI-minus:SI	1	0.09	1	0.08	0.25
plus:SI-plus:SI	2	0.18	2	0.17	
minus:SI-reg:SI	1	0.09	1	0.08	0.25
plus:SI-reg:SI	2	0.18	2	0.17	
reg:SI-ashift:SI	1	0.09	1	0.08	0.08
reg:SI-const_int:	1	0.09	1	0.08	0.08
reg:SI-jump_insn	1	0.09	1	0.08	0.08
Overall	11	1.00	12	1.00	1.00
Cycles	100	9.09	100	8.33	8.33

Table 1: Numerical features (for the ARM processor) extracted by a single execution of the function `main` presented in Figure 1(a) and effects of the pre-processing steps on it.

3. it is generated after the middle-end compilation flow: in this way all the target-independent optimizations have already been performed;
4. with a slightly modified version of the GNU GCC compiler it is possible to preserve the correlation among corresponding source code operations, GIMPLE operations and RTL operations; this allows, given a portion of the source code of the application, to identify the set of RTL operations generated from it; note that the changes applied to the compiler are independent from the target processor considered, so we can use the same GNU GCC patches to build performance model of different processors;
5. we can profile the application on the host machine by annotating its original source code; the profiling information can then be easily coupled with the static information contained in the RTL representation.

Nevertheless, this type of analysis still has some intrinsic limitations and can not model all the performance details of an application. The first limitation consists of the neglecting of the caches, whose effects are difficult to describe in a linear performance model. The second limitation depends upon the stage of the back-end flow where we extract the representation to be analyzed which is at the early stages of the back-end flow before most of the back-end optimizations have been applied. Transformations and optimizations performed in the back-end flow can change the structure of the application code, destroying the possibility of easily correlate the GIMPLE operations with RTL operations. This correlation has to be preserved for two reasons: to allow the estimation technique to give feedback to the designer on the performance of arbitrary pieces of code and to map source code level profiling information to the RTL representation. For this reason we do not use the RTL produced in the final stages of the compilation flow and we do not consider assembly level suitable for the proposed methodology. However, since we need to model in some way how a back-end optimization changes the relation between the RTL representation and performance, we use a different performance model for each combination of back-end optimizations.

4.2 Model building

Before executing the linear regression, we apply three steps of data pre-processing. The three steps are: *normalization*, *main introduction* and *clustering*. The first two steps aim at improving the accuracy of the model, while the third simplifies the performance model.

4.2.1 Normalization

One of the limitations in applying standard linear regression techniques to the performance estimation problem resides in the evaluation of the fitness of a model. The standard approach minimizes the Mean Squared Error MSE:

$$MSE = \frac{1}{n}\sum_{i=1}^{n} e_i^2 \qquad (4)$$

where e_i (residuals or estimated errors) are the difference between the observed data V_i and the predicted data $\overline{V_i}$:

$$e_i = V_i - \overline{V_i} \qquad (5)$$

.

For example, consider two observed data $V_i = 100$ and $V_j = 10,000$ and their predicted values $\overline{V_i} = 200$ and $\overline{V_j} = 10,100$. According to the MSE criterion, the error introduced in the two estimations is the same ($100^2 = 10,000$). However, the first estimation is double with respect to the real value, while the second estimation is wrong by only the 1%. This means that the *relative error* is 2 for the first prediction and 0.01 for the second. The fact that MSE does not take into account the relative error is critical in the case of performance estimation, because different applications may have very different execution delays, up to several orders of magnitude. Thus, a model built by only minimizing MSE would better fit long applications, while it would not correctly describe the shorter ones.

There also exist other linear regression techniques which consider different evaluation criteria, such as *Robust Regression* and *Generalized Least Squares*, but they still use criteria which are based on the *absolute* rather than the relative error.

In the proposed methodology, we overcome this limitation by changing the dependent variables $cycles$ through a normalization process. Instead of building a model for estimating the *overall execution cost* of an application, we apply the linear regression technique to infer a performance model for estimating the *average cost of a sequence*.

So, the new performance model which we are considering is characterized by:

- *input*: for each RTL sequence class, the fraction of the sequences of the application which belong to that class;

- *output*: the average number of cycles required by an RTL sequence of that application; range of this new dependent variable is less sensible than the original one.

All these data can be easily obtained by dividing both the RTL sequences classes counters and the measured execution delay of an application by the number of its RTL operations.

The third column of Table 1 shows the values of the features after the normalization on the example of Figure 1.

4.2.2 Main introduction

Generally, when we retrieve the execution delay of an application on a processor, using either simulators or direct instrumentation, we obtain a measure which is comprehensive of the start-up time of the application itself. This overhead can be approximated to be constant and independent from the application we are measuring, but its effect on the average cost of a sequence depends on the length of the application. Since we are using fractions instead of absolute values, the data passed to the regression do not store information about the overall dimension of the application. Thus, the relative influence of the start-up time on the average number of cycles of each sequence is not quantified.

This problem has been solved by introducing a fake operation at the beginning of the execution flow, belonging to a special class of operations called *main*. This operation represents the start-up of the application. The operation class *main* is treated like all the others, so the first sequence of the application will be an RTL sequence <*main:*, . . . >: its corresponding fraction will be larger for the short applications and smaller for the long.

The fifth column of Table 1 shows what happens to the computation of the fractions when the *main* operation is introduced.

4.2.3 Clustering

The last pre-processing step consists of reduction of the numerical features cardinality performed through a sequence clustering. Indeed, if we considered each single possible <*op:type-op:type*> as a different sequence class, we would have to consider millions of numerical features and so we would need millions of applications to build a performance model. Since not all these operations can be executed by a particular processor and not all these sequences can really occur in an application, the actual number of sequences is less but however it can still be quite huge.

In the practice, the number of the RTL operation classes and consequently the number of the RTL sequences can be drastically reduced by imposing an equivalence relation among <*op:type*> classes. This equivalence relation should describe which operations can be considered performance-equivalent. Since the proposed methodology is based on zero knowledge of the target architecture, building this relationship without this information could seem a very hard task. However, there are couple of operations which can be reasonably considered performance equivalent independently from the target architecture (e.g., plus and minus, less than and greater than, same operation on similar type of data) and so that can be inserted into the equivalence relation. Adding not performance equivalent operations to this relation can obviously decrease the accuracy of the produced model: the introduced error in particular will depend on their actual performance difference and on their frequency. On the other hand, the obtained benefits are a simpler estimation model and a reduction of the training set over-fitting.

The results of a possible clustering on the example of Figure 1 are shown in the last column of Table 1. In this case, the data regarding `plus` and `minus` have been aggregated.

Finally the Equation of the produced linear model is:

$$AC = M(\overline{N}) = \beta_0 + \sum_{i \in F} \left(\beta_i \cdot \frac{N_i}{\sum_{j \in F} N_j} \right) \quad (6)$$

where AC is the average number of cycles required for executing a sequence, β_0 is the intercept term, F is the set of sequences classes, β_i is the regression coefficient of the sequence class f_i, N_i is the counter of sequences of class f_i.

4.3 Model application

Given a generic application, we now want to use the performance model $M(\overline{N})$ to estimate its overall execution delay $Cycles$. We apply the RTL analysis described in Section 4.1 to obtain the sequence class counters $(\overline{N})$.

The overall execution delay $Cycles$ can be expressed as:

$$Cycles = AC \cdot \sum_{i \in F} N_i \quad (7)$$

.

By substituting AC with the value computed in Equation 6 we obtain:

$$Cycles = \beta_0 \cdot \sum_{i \in F} N_i + \sum_{i \in F} \beta_i \cdot N_i = \sum_{i \in F} ((\beta_i + \beta_0) \cdot N_i) \quad (8)$$

If we replace in Equation 8 $\beta_i + \beta_0$ with $\hat{\beta}_i = \beta_i + \beta_0$ we obtain:

$$Cycles = \sum_{i \in F} \left(\hat{\beta}_i \cdot N_i \right) \quad (9)$$

The performance model described by Equation 9 is derived by Equation 6 but allows to estimate directly the overall execution delay of the application. Furthermore, it allows to easily identify the contribution to the overall delay given by each RTL sequence. In fact, the contribution of a generic sequence belonging to class F_i is $\hat{\beta}_i$. The contribution of a generic part of the source code can be computed by retrieving the corresponding RTL sequences and by summing all their contributions.

5. EXPERIMENTAL EVALUATION

In this Section, we validate our methodology by estimating the performance of a set of benchmarks onto two processors based onto two different Instruction Set Architectures: an ARM926EJ-S processor and a LEON3 processor. We initially describe the experimental setup and then present the experimental results.

5.1 Experimental Setup

We integrated this methodology in PandA [2], a framework for the Hardware/Software codesign based on the GNU GCC compiler. We exploit three components of the framework: an RTL representation analyzer, a basic block profiler and a tool for measuring the application performance on the target processor. The analyzer and the basic block profiler are used both in the performance model building and application, while the last tool is used only during the training.

The RTL representation analyzer is implemented by applying the changes described in Section 4.1 to the GNU GCC, which is wrapped by the framework itself which directly controls which optimizations the compiler performs. Since the RTL description of the application produced during the compilation flow depends on the considered target, the used GNU GCC has to target the processor for which we want to build the performance model. In particular, since we validate our methodology on the estimation of the performance of an ARM processor and of a SPARC-compliant processor, we build both an ARM cross-compiler and a SPARC cross-compiler.

Regarding the profiler, the framework implements the Hierarchical Path Profiling technique [10] which works by instrumenting the source code produced starting from the GIMPLE internal representation. The instrumented code is then compiled for the host machine and executed on it. If it is not possible to execute the analyzed application on the host machine, the profiling information has to be provided with other methods, otherwise it is not possible estimate the application or use it in the training set.

From the path profiling information we extract the GIMPLE sequence counters and, by correlation, the RTL sequence counters presented in Section 4.1. Data are then pre-processed by applying all the steps described in Section 4.2. In particular, the main clustering introduced concerns the data types, which have been clustered into two: integer and floating point. In this way all operations characterized by a given operator have been clustered into only two classes: integer operations and floating point operations. This grouping surely introduces approximation in the produced models, but it allows to reduce significantly the number of possible sequences. The time required for the analysis of an application is mainly determined by the time required for performing its profiling. The instrumentation overhead introduced for the profiling ranges from 20% to 200% on a host linux machine with an Intel Xeon X5355 CPU (four cores, where each core runs at 2,33 GHz and accesses 4 MB of L2 cache shared with another core).

Considering the first target processor, the execution delays of the applications in the training set are obtained by direct execution on the target ARM926EJ-S processor, implemented on the Atmel DIOPSIS 940HF [1], running a Linux 2.6.24 kernel. The ARM926EJ-S processor implements the ARMv5TEJ instruction set with 5 stage pipeline, includes an enhanced 16 x 32-bit multiplier capable of single cycle MAC operations, and embeds separated 16 KB data and instruction caches. The delays are measured by adding instrumentation at the beginning and at the end of the source code of the application and just before and after system and library function calls, exploiting the `gettimeofday` c function. The time spent on the execution of system and library functions is subtracted from the application execution time since the proposed methodology doesn't aim at estimating the execution time of functions which source code is not available.

Obviously, the accuracy of these measurements directly influences the accuracy of the estimation model. In particular, the performance measures can be influenced by operating system activity such interrupt handling or execution of other applications. To try to remove such effects, we measure the execution delays of the training applications several times (at least 10). The data of the different runs are collected by the framework which tries to identify possible outliers. This is done by computing the mean value and the standard deviation and discarding the values that differ from the mean more than the deviation. The mean value of the remaining data is considered the real measure. Nevertheless, there can still be small measure errors in the collected data.

The second embedded processor used to validate our methodology is the LEON3 processor, a 32-bit microprocessor compliant with the SPARC V8 ISA. Based on LEON architecture originally designed by the European Space Agency, it is currently developed by Gaisler Research and licensed under GNU GPL. Real execution times of applications on LEON3 processor are obtained by exploiting TSIM, a cycle accurate simulator of the LEON3 processor. Also in this case, the execution times of system and library functions is subtracted from the application overall execution time.

To perform the linear regression, we use RapidMiner (formerly YALE [18]), an open-source java based tool for knowledge discovery and data mining.

To evaluate the quality of the models produced by our methodology and in particular to evaluate the gain provided by its two main contributions (i.e, the use of the RTL representation and the use of operations sequences), we compare their accuracy with the ones obtained starting from analyses performed on single operations and at higher level (GIMPLE representation).

GIMPLE representation is by design target independent and well fits in the class of Virtual Instructions representations proposed in [4, 11]. In fact, GIMPLE provides simple integer operations (e.g., `plus_expr` on `integer_type`), floating point operations (e.g., `plus_expr` on `real_type`), multiple and divide operations (e.g., `mult_expr` and `rdiv_expr`), subroutine call and return (e.g., `call_expr` and `return_expr`) and conditional and unconditional branches (e.g., `cond_expr` and `goto_expr`). We analyze the GIMPLE intermediate representation produced at the end of the middle-end optimization flow so that all the middle-end optimizations performed by the GNU GCC compiler are taken into account. From this representation and the profiling information, we extract the dynamic counters of the different classes of GIMPLE operations and of GIMPLE sequences which are then used as data for the linear regression after the same pre-processing described in Section 4.2.

We validate the proposed methodology on a very large set of C benchmarks. We extracted more than 600 benchmarks from six suites: DSP Stone [22], Splash 2 [24], Powerstone [17], OmpSCR [9], NAS Parallel Benchmark [3] and the GNU GCC testsuite [12]. The first three suites are typically used to validate Embedded System Design methodologies. DSP Stone, in particular, is used to measure the performance provided by DSPs coupled with their compilers. OmpSCR, instead, is a benchmark suite for evaluating the performance of shared memory multiprocessors platform in distributed systems; the NAS Benchmarks are designed to evaluate the performance of supercomputer systems. Finally, the GNU GCC testsuite is a set of tests used by the GCC developers to verify the correctness of the compiler. In building performance models for each processor, we consider two different set of optimizations corresponding to two optimizations levels of the GNU GCC: `-O0` (no-optimization) and `-O2` (optimize even more).

5.2 Experimental Results

To guarantee the generality of the presented results (i.e., to guarantee that this type of models can correctly estimate new applications), we apply the cross-validation technique provided by Rapidminer. Cross-validation is a technique for assessing the performance of a statistical analysis on an independent data set: in particular it is used to prove that the performance of the estimation does not depend on a particular choice of the training and the testing sets. Several methods for performing cross-validation exist: we use the *K-fold cross-validation* that acts as follow. The initial data set is randomly divided into k subsets (in our case, $k = 10$). The model building process is repeated k times and, at each iteration i, a performance model is built by analysing all the subsets except the i-th, which is used to test the model. At the end, the average error across all k trials is computed and a single model is built by combining the k models. The randomness and the repetition of the process guarantee that the results do not depend on a particular choice of the training and testing data.

Table 2 reports the results of the cross-validation of the models produced by using the analyses at the GIMPLE and at the RTL level based both on single operations and on sequences. The Table shows

		GIMPLE				RTL			
		Single Operations		Operations Sequences		Single Operations		Operations Sequences	
Processor	Opt. Level	ME	SD	ME	SD	ME	SD	ME	SD
ARM	O0	34.19%	51.54%	32.73%	39.69%	15.45%	30.15%	8.25%	7.08%
ARM	O2	33.50%	47.12%	34.92%	35.64%	24.64%	13.94%	18.72%	18.64%
LEON3	O0	15.57%	30.32%	9.09%	8.48%	13.60%	25.08%	8.03%	6.28%
LEON3	O2	14.16%	28.11%	14.68%	31.69%	9.91%	11.79%	6.74%	5.67%

Table 2: Cross-validation results on the accuracy of the performance models built starting from the different application analyses. *ME* is the Mean Error, *SD* is the Standard Deviation.

	Single Operation	Operations Sequence	Gain
GIMPLE	34.19%	32.73%	1.46%
RTL	15.45%	8.25%	7.20%
Gain	18.74%	24.48%	**25.94%**

(a) Accuracy gain on ARM performance model at O0 optimization level

	Single Operation	Operations Sequence	Gain
GIMPLE	33.50%	34.92%	−1.42%
RTL	24.64%	18.72%	5.92%
Gain	8.86%	16.20%	**14.78%**

(b) Accuracy gain on ARM performance model at O2 optimization level

	Single Operation	Operations Sequence	Gain
GIMPLE	15.57%	9.09%	6.48%
RTL	13.60%	8.03%	5.57%
Gain	1.97%	1.06%	**7.54%**

(c) Accuracy gain on LEON3 performance model at O0 optimization level

	Single Operation	Operations Sequence	Gain
GIMPLE	14.16%	14.68%	0.52%
RTL	9.91%	6.74%	3.17%
Gain	4.25%	7.94%	**7.42%**

(d) Accuracy gain on LEON3 performance model at O2 optimization level

Table 3: Gains obtained by exploiting RTL representation and operation sequences. Lowest rightest cells report the overall gain of the proposed methodology (analysis based on RTL sequence) with respect of analysis based on single Virtual Instruction (GIMPLE).

the mean and the standard deviation (in percentage) of the error obtained at the end of the process. These data are compared in the Table 3 which shows the contribution of the two characteristics of the proposed technique (i.e., use of RTL operations and use of sequence of operations) to its overall gain with respect to a technique based on analysis of Virtual Instructions. This Table shows how the models produced with the proposed methodology are almost twice more accurate than the models produced using GIMPLE operations in the worst situation (18,72% vs 33.50% for the ARM with O2 optimizations) and almost four times accurate in the best case (8.25% vs. 34.19% for the ARM with O0 optimizations). In particular the gain provided by the usage of RTL representation is more relevant in the estimation of the performance of the ARM processor since the GIMPLE representation worse approximates the final assembly code of the ARM processor (accuracy is 34.19% and 33.50%) than in the SPARC case (15.57% and 14.16%). Instead the use of the RTL representation, which is closer to the assembly than the GIMPLE, allows to better estimate its performance. Performances of the ARM, in particular when O2 optimization level is applied, are however harder to estimate than LEON3 also exploiting RTL representation because of its more complex architecture.

Considering the introduction of sequences in the analysis, it can be noticed how they do not provide a real advantage in performance estimation performed at higher level, since they are not able to correctly model all the assembly sequences. On the other hand, introducing the analysis of RTL sequences gives real benefits such as doubling the accuracy in the case of the performance model of the ARM with O0 optimization levels. For both the processors, the gain provided by the use of the RTL sequences is more significant at O0 optimization level than at O2 optimization level since back-end optimizations performance effects are however not so easy to predict using a linear models.

We now compare the results obtained by our methodology with some of the works presented in Section 2 that exploit analytical models. Our proposed methodology builds estimation models by exploiting linear regression and predicts quite accurately the performance of a set of heterogeneous applications, in contrast with what is stated in [11]. Our approach seems able to well describe applications from different application fields with the same models, and thus it requires less model specialization. Furthermore, by using a virtual instruction set, the models in [11] do not consider the effects of the optimizations. In our work, instead, we show that with the RTL representation, a pre-processing of the data before applying the linear regression, and an adequately sized training set, we are able to obtain a good estimation for very heterogeneous applications. The average error shown by the technique proposed in [21] is 6.03%, which is smaller than what obtained by our methodology, but it can only be used on applications without loops and recursion. The methodology proposed in [6] for estimating the performance of new applications on the same architecture produces average errors comparable to ours. The error ranges from 0.06% to 19.3% for the linear models and from 0.15% to 17.0% for non linear models. However, it seems that their methodology is focused on the performance estimation of known applications on unknown data, and not on the estimation of unknown applications. The number of analyzed benchmarks is also limited with respect to the number of the datasets: at most they use 6 benchmarks, with 15 different datasets for each. Furthermore, all the considered applications come from the same field, and have similar characteristics.

Results obtained by methodologies working at assembly level are better than our as expected, but this gain requires a detailed knowledge of the target architecture. The methodology presented in [16] works at the assembly level and seems to give better results. The average error obtained is smaller than 4%, but only three control intensive applications have been tested. Oyamada et *al.* [19,20] obtain an average error of 10.08% on a sufficiently hetereogenous set of benchmarks. The average error is measured by partitioning benchmarks in a training and in a test set and not with a cross-validation analysis. Their methodology combines the use of a non-linear model such as Neural Networks with assembly level information. Nevertheless, the use of a more detailed information and of non-linear models give a gain of only 8.64% with respect to the results obtained with our methodology in the worst case.

6. CONCLUSIONS

In this paper we presented a methodology for estimating the execution delay of C applications using the linear regression technique. The methodology combines basic block profiling information collected on the host machine with the RTL level information produced by the GNU GCC. The designer does not have to provide any information to build the performance model. All the needed information about the characteristics of the target is automatically and directly extracted from the RTL representation and indirectly by exploiting operation sequences. Moreover, the use of the RTL sequences allows to preserve correlation between the performance estimation and the source code. We applied the proposed methodology onto a set of heterogeneous C benchmarks and we tested its accuracy by using a cross-validation technique on two processors with different ISA. The results show that our approach produces better results than techniques based on target independent representations (i.e. GIMPLE), with an average error of 10.43% (vs. 24.35%). Nevertheless, by maintaining the correlation between the GIMPLE and the RTL representation, it remains sufficiently flexible to be adapted to any target processor supported by GCC without any modifications. Finally, by using different models for different levels of compiler optimization, the performance estimations correctly take into account the effects of a good range of optimizations.

Future works will mainly focus on exploiting numerical features based on longer operation sequences. Since the number of different sequences can grow exponentially with their length (e.g., extending the ARM rtl sequences from two to three operations increases the number of sequences from 141 to 1243), we have to improve the clustering phase to allow the building of performance models based on sequences longer than two using only some hundreds of benchmarks.

7. REFERENCES

[1] ATMEL DIOPSIS 940HF datasheet, available at http://www.atmel.com/products/Diopsis/.

[2] The PandA framework.

[3] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow. The nas parallel benchmarks 2.0. Technical report.

[4] J. Bammi, E. Harcourt, W. Kruitzer, L. Lavagno, and M. Lazarescu. Software performance estimation strategies in a system-level design tool. In *CODES 2000: the Eighth International Workshop on Hardware/Software Codesign.*, pages 82–86, 2000.

[5] G. Beltrame, C. Brandolese, W. Fornaciari, F. Salice, D. Sciuto, and V. Trianni. Modeling assembly instruction timing in superscalar architectures. In *ISSS '02: 15th international symposium on System Synthesis*, pages 132–137, 2002.

[6] G. Bontempi and W. Kruijtzer. A data analysis method for software performance prediction. In *DATE '02: Conference on Design, Automation and Test in Europe*, page 971, 2002.

[7] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. Source-level execution time estimation of c programs. pages 98–103, 2001.

[8] J. W. Davidson and D. B. Whalley. Quick compilers using peephole optimization. *Softw. Pract. Exper.*, 19(1):79–97, 1989.

[9] A. J. Dorta, C. Rodriguez, F. de Sande, and A. Gonzalez-Escribano. The openmp source code repository. In *PDP*, pages 244–250, 2005.

[10] F. Ferrandi, M. Lattuada, C. Pilato, and A. Tumeo. Performance estimation for task graphs combining sequential path profiling and control dependence regions. In *MEMOCODE'09: Proceedings of the 7th IEEE/ACM international conference on Formal Methods and Models for Codesign*, pages 131–140, Piscataway, NJ, USA, 2009. IEEE Press.

[11] P. Giusto, G. Martin, and E. Harcourt. Reliable estimation of execution time of embedded software. In *DATE '01: Conference on Design, Automation and Test in Europe*, pages 580–589, 2001.

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

[13] 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 *5th International Workshop on Languages and Compilers for Parallel Computing*, pages 406–420, 1993.

[14] Y. Hwang, S. Abdi, and D. Gajski. Cycle-approximate retargetable performance estimation at the transaction level. In *DATE '08: Conference on Design, Automation and Test in Europe*, pages 3–8, 2008.

[15] T. Kempf, K. Karuri, S. Wallentowitz, G. Ascheid, R. Leupers, and H. Meyr. A sw performance estimation framework for early system-level-design using fine-grained instrumentation. In *DATE '06: Conference on Design, Automation and Test in Europe*, pages 468–473, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[16] M. Lajolo, M. Lazarescu, and A. Sangiovanni-Vincentelli. A compilation-based software estimation scheme for hardware/software co-simulation. In *CODES '99: the Seventh International Workshop on Hardware/Software Codesign, 1999*, pages 85–89, 1999.

[17] A. Malik, B. Moyer, and D. Cermak. A low power unified cache architecture providing power and performance flexibility (poster session). In *ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design*, pages 241–243, New York, NY, USA, 2000. ACM.

[18] I. Mierswa, M. Wurst, R. Klinkenberg, M. Scholz, and T. Euler. Yale: Rapid prototyping for complex data mining tasks. In L. Ungar, M. Craven, D. Gunopulos, and T. Eliassi-Rad, editors, *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 935–940, New York, NY, USA, August 2006. ACM.

[19] M. Oyamada, F. R. Wagner, M. Bonaciu, W. Cesario, and A. Jerraya. Software performance estimation in mpsoc design. In *ASP-DAC '07: The 2007 Asia South Pacific Design Automation Conference*, pages 38–43, Washington, DC, USA, 2007. IEEE Computer Society.

[20] M. S. Oyamada, F. Zschornack, and F. R. Wagner. Applying neural networks to performance estimation of embedded software. *J. Syst. Archit.*, 54(1-2):224–240, 2008.

[21] K. Suzuki and A. Sangiovanni-Vincentelli. Efficient software performance estimation methods for hardware/software codesign. In *DAC '96: 33rd Design Automation Conference*, pages 605–610, Jun, 1996.

[22] V. živojnović, J. M. Velarde, C. Schläger, and H. Meyr. DSPSTONE: A DSP-oriented benchmarking methodology. In *ICSPAT '94: International Conference on Signal Processing and Technology*, 1994.

[23] W. Wolf. The future of multiprocessor systems-on-chips. In *DAC '04: The 41st annual Design Automation Conference*, pages 681–685, 2004.

[24] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *ISCA*, pages 24–36, 1995.

[25] M. Zhao, B. R. Childers, and M. L. Soffa. An approach toward profit-driven optimization. *ACM Trans. Archit. Code Optim.*, 3(3):231–262, 2006.