Multi-Metric and Multi-Entity Characterization of Applications for Early System Design Exploration

Lukai Cai, Andreas Gerstlauer, and Daniel Gajski
Center for Embedded Computer Systems, University of California, Irvine
{lcai, gerstl, gajski}@cecs.uci.edu

Abstract— At system level, intensively analyzing the system application will produce a variety of useful characteristics and provide designers valuable exploration indications. In this paper, we present such an analysis approach based on the instrumentation-based profiling. The proposed approach analyzes complex system application and generates multi-metric and multi-entity characteristics. Experimental results show the applicability of the approach for efficient early design space exploration.

I. Introduction

At system level, system behavior and the interaction between behavior blocks must be completely understood and explored. This initial behavior-based analysis allows for better heuristics for application-architecture mapping. A simple example is to find out the behavior block which contains most of computation for later optimization.

Behavior-based analysis demands a new approach to derive the characteristics of the application during the simulation. Because system design involve processes, channels, and variables in the system behavior as well as PEs, buses, and memories in the architecture platform, the required approach should analyze all the application entities including process, channel, variable, and port (**multi-entities**). It should also intensively derive static and dynamic characteristics for operation, traffic, and storage metrics (**multi-metrics**).

Traditionally, estimation approaches are based on either a purely static analysis or a purely dynamic simulationr [7, 9, 4, 8]. These approaches are applied after design space exploration. In contrast to them, there is only a limited number of approaches that aim to analyze system behavior. Traditional software profiling tools [3, 6] usually provide target/host machine-dependent characteristics. Even though such profilers can produce some characteristics of system behavior, they only support operation-related data such as function call statistics.

Based on instrumentation-based profiling approach [2], in this paper, we target for the system behavior before design space exploration, and automatically generate multi-entities and multi-metrics characteristics. The compuated characteristics can be used extensively for behavior analysis and earily design space exploration before exploration.

The rest of this paper is organized as follows. In Section II, we introduce the specification characteristics. In Section III, we describe the usage of the generated characteristics for early design space exploration . In Section IV, experimental results that show the applicability of the approach to the space exploration are presented. Finally, the paper concludes with a summary in Section V.

II. Specification Characteristics

Our previous work [2] introduced the design flow of the adopted instrumentation-based profiling. We first instrument and simulate the system behavior to collect execution counts that capture the dynamic behavior of the application at the basic block level N_{BB}. We then compute specification characteristics $r_{i,d}$, $i \in I$, $d \in D$ by statically analyzing the code together with the collected counters N_{BB}. Specification characteristics are computed hierarchically for each behavior, port, variable, and channel in the specification. I is the set of possible item types defined by the characteristics' category and D is the set of data types found in the code.

Specification characteristics are classified into three categories: operation, traffic, and storage. In each category static and dynamic metrics are computed. Static characteristics are derived directly from the code of the specification model whereas dynamic characteristics depend on data collected during simulation. In general, static and dynamic specification metrics $R = \sum_i \sum_d r_{i,d}$ in each category are computed by summation of corresponding characteristics r over a subset of item and data types.

A. Operation Characterization

Operation characteristics signify the complexity of the computation in the specification. They are attached to behaviors as the computational units of the system.

Static operation characteristics are defined as the number of operations in the code of each behavior. They represent the **code complexity** which is related to code size or implementation complexity of the control unit in general.

The static operation characteristics of a leaf behavior equals to the summation of the characteristics of all its

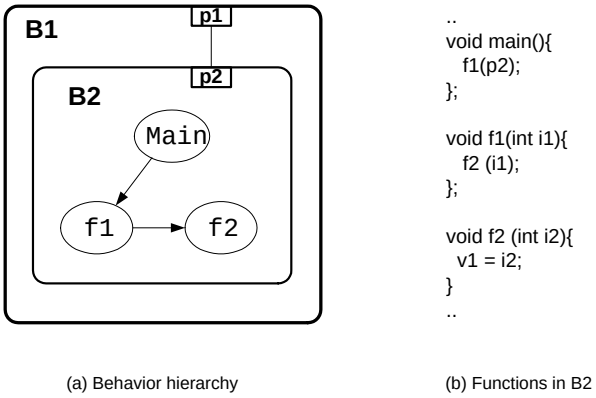

Fig. 1. Example of dynamic traffic computation

local functions. The static operation characteristics of a hierarchical behavior equals to the summation of the characteristics of all its child/instantiated behaviors.

Dynamic operation characteristics are defined as the number of operations executed by each behavior during simulation. Dynamic operations represent the **computational complexity** in the system which is related to performance issues.

In general, the dynamic operation characteristics of a leaf behavior equals to the characteristics of its main function. The characteristics of a hierarchical behavior equals to the summation of the characteristics of its child behaviors. More detailed algorithms of dyanamic operation computation, including for hierarchical behavior instantiation and recursive function calls, are described in [1].

B. Traffic Characterization

Traffic characteristics signify the complexity of the communication in the specification as the amount and type of data exchanged, providing separate input and output traffic characteristics via corresponding item types. As behaviors communicate through variables and channels connected to their ports, traffic characteristics are attached to behavior ports and variables and channels connected to them. Furthermore, traffic characteristics are also attached to behaviors.

Static traffic characteristics are defined as the number of connected ports of a certain type. They represent **connectivity complexity**, which relates to the message passing traffic incurred between two dependent behaviors in order to make the output of a behavior available at the next behavior's inputs.

For a behavior's port, static traffic characteristics reduce to the size of the port itself (1 in most cases). For a variable or channel, they are equivalent to the number of connected behaviors. The static traffic characteristics of behavior equals to the summation of the static traffic of its ports.

Dynamic traffic characteristics are defined as the number of times a port or a variable/channel of a certain type is accessed during simulation. An access is generated whenever a statement in the code reads from a port variable, writes to a port variable, or calls a port interface method. Dynamic traffic characteristics represent **access complexity** which relate to the traffic incurred for a shared memory implementation of communication between dependent behaviors.

We compute dynamic traffic characteristics for variables, ports, and behaviors. To our knowledge so far, there is no any profiling approach for traffic computation. Traditionally, in order to get traffic information, designers add a monitor on the interested port of behavior and trace the data through the port. However, adding monitors to all the ports of behaviors is unskillful because it not only requires designers to tediously add such monitors, but also slow down the simulation speed.

This paper proposed a traffic profiling approach, which is illustrated by the example in Figure 1. Behavior *B1* has an instantiation *B2*. The port *p2* of *B2* is directly connected to *p1* of *B1*. Behavior *B2* contains three functions: *main*, *f1*, and *f2*, the codes of which are displayed in Figure 1(b). Because *i2* of *f2* is bound to *i1* of *f1*, *p2* of *B2*, and further *p1* of *B1* during function calls and behavior instantiations, a read access of *i2* will generate a read access of *p2* and *p1*. The proposed algorithm analyzes such port-parameter binding information and computes the traffic of ports accordingly. Details of this algorithm can be found in [1].

The dynamic traffic for a channel equals to summation of the dynamic traffic of channel's functions. We compute dynamic traffic of channel's function by treating the total number of function's parameter access as the read access and treating the total execution number of channel functions with return value as the write access. The dynamic traffic for a variable equals to the summation of the dynamic traffic of ports that the variable is directly connected to . The dynamic traffic for a behavior equals to the summation of the dynamic traffic of behavior's ports.

C. Storage Characterization

Storage characteristics signify the amount of storage required to hold the system's data. For each behavior and channel, storage requirements are computed where item types distinguish between local and global storage.

Static storage characteristics are defined as the number of static variables of a certain data type declared inside the behavior/channel and its children. This includes variables declared at the behavior/channel level and static variables inside functions. Static storage represents **static storage requirements**, i.e. storage that needs to be allocated globally for the whole lifetime of the system.

The static storage of leaf behavior equals to the summation of the size of variables declared at its the behavior/channel level and the size of static variables in-

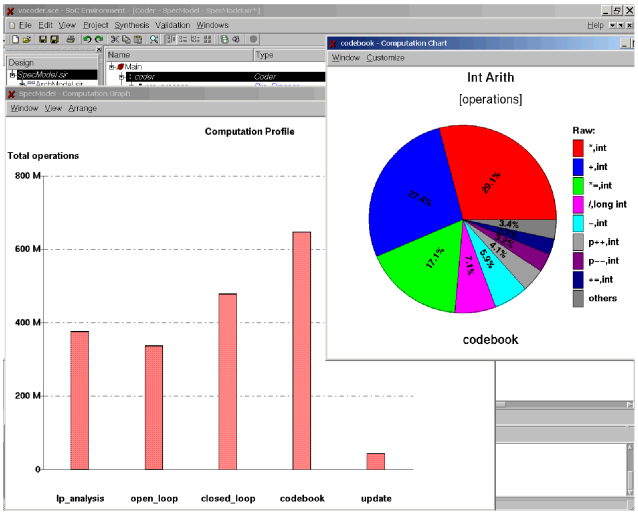

Fig. 2. Vocoder specification characteristics GUI.

side functions. The static storage of hierarchical behavior equals to the summation of the size of variables declared at its the behavior/channel level and the size of static storage of all its child behaviors.

Dynamic storage characteristics are defined as the number of variables of a certain data type allocated and deallocated dynamically during runtime. The local item type of dynamic storage represents **stack requirements** based on the number of local variables declared inside functions. The global item type of dynamic storage, on the other hand, represent **heap requirements** based on the amount of storage allocated dynamically on the heap during runtime (e.g. via `malloc()` calls).

The stack storage of leaf behavior equals to the stack storage of its main function. The stack storage of hierarchical behavior equals to the summation of the stack storage of its child behaviors. For a function, its stack storage contains not only the summation of the size of variables declared in it, but also contains the maximum of the stack storage of its called functions.

III. Usage for Exploration

The computed specification characteristics are critical because it helps for the system design. In this section, we introduce a number of the usage scenarios of specification characteristics in the design space exploration.

Concurrency optimization. Concurrency optimization derives all the parallelism existing among behaviors of the system behavior and generates the behavior hierarchy which explicitly specifies all the parallelism. The derived parallelism implies the possibility of mapping different behaviors to different PEs for concurrent execution. We analyze the parallelism based on the static/dynamic traffic characteristics and the port connections. For example, an application contains two behaviors: *B1* and *B2*. The only port *p1* of *B1* is connected to the only port *p2* of *B2* through variable *v1*. If the traffic [1].of *v1* is 0, then we conclude that behaviors *B1* and *B2* can be executed concurrently.

Architecture selection. In the system synthesis approach, designers are responsible to select PEs from PE library to assembly the system architecture. We select PEs by matching the behavior's operation characteristics with PE attributes. For example, if the application are multiplication-intensive identified by operation characteristics, we then prefer to select DSP with a hardware multiplier.

Behavior mapping. We map the system behavior to the system architecture. Similar to architecture selection, we match the attributes of behaviors and PEs. Furthermore, operation characteristics identifies the most complex behaviors, which may be the best candidates for mapping to the fastest PE. We also prefer to map the behaviors which communicate heavily (identified by traffic characteristics) to the same PE or to the PEs connected by dedicated busses.

Variable mapping. We map variables to local memories of PEs, or global memories in order to pursue the fastest memory access time and smallest memory size. We compute memory access time based on traffic characteristics of variables and compute memory size based on the storage characteristics.

Design alternative estimation. By giving weight to the specification characteristic of each data and item type, the attributes of different design alternatives such as performance and power are estimated. For example, assuming a behavior *B1* is mapped to PE *PE1*. If *B1* contains 30 integer-type multiplication operations, and executing such an operation on *PE1* requires 2 clock cycles, then the executing time of *B1* on *PE1* for these operations is 30 * 2 = 60 clock cycles.

Multi-level profiling. In addition to use the proposed approach on system behavior, we can also applied it on models at different abstraction levels, such as transaction level or RTL level. For example, using the proposed approach on bus functional level can obtain PE's fan-in fan-out and communication delay, based on the traffic characteristics.

We developed two tools: *System Profiler* and *System Explorer*. *System Profiler* implemented multi-matric, multi-entity, and multi-level profiling/estimation covering the content in this paper and tasks 5 and 6. *System Explorer* automatically makes design space exploration, which includes automation of tasks 1 to 4. The output of *System Profiler* such as specification characteristics is directly read by *System Explorer* as the evaluation base. The details of *System Profiler* and *System Explorer* are described in [1].

[1]Global variables are not allowed in the specification

LP_Analysis	377.0 MOp
Open_Loop	337.1 MOp
Closed_Loop	478.7 MOp
Codebook	646.5 MOp
Update	43.6 MOp

TABLE I
Computational complexity of top-level vocoder behaviors.

IV. Experimental Results

We use *System Profiler* on the design examples of a voice codec for mobile phone applications (vocoder) [5]. The vocoder specification consists of appr. 13,000 lines of code. For a testbench that exercises the design with 163 frames, this translates to a total timing constraint of 3.26 s. The produced specification characteristics of Vocoder provide the following indications.

Computation-intensive application. The total number of executed operations during simulation is 1,921,874,900. The total number of data transferred among behaviors is 472,944. The operations and transferred data are both for basic data types, such as integer and float. The comparison of operation and traffic indicates that the Vocoder is computation-intensive application. Therefore, designers should more concentrate on computation optimization.

Sequentiality. Based on the traffic and operation characteristics, we also conclude that the Vocoder is a mostly sequential application. The possible parallel executed behaviors contains 261,813,860 operations. In terms of the dynamic operation characteristics, it denotes the maximin 13.6% speed up by exploiting the concurrency. Therefore, while keeping such limited concurrency in mind, designers should more focus on optimizing the critical parts of the behavior sequence.

Criticality. Table I shows the computational complexity for the vocoder's five top-level behaviors in millions of operations (MOp). The *Codebook* search behavior is by far the most critical vocoder block. It takes up 33.7% of the whole computation and is a good candidate for further optimization.

PE matching. Table II shows the mix of operations in the *Codebook* behavior. (a screenshot of the operation mix pie chart and the bar graph of the top-level behaviors as displayed in the design environment GUI is shown in Figure 2). The codebook search (and the vocoder in general) does not contain any floating-point but only integer-type operations, i.e. processors with dedicated floating-point units are not necessary and processor selection should focus on integer performance instead. Furthermore, most of the operations are multiplications, i.e. selected processors should have dedicated hardware multipliers.

Behavior mapping. Table III shows the traffic among five top-level behaviors, in the unit of data transfer per basic data type. Because the traffic between *Codebook* and other behaviors are relatively small, mapping *Codebook* to faster PE for optimization while mapping rests to slower one will not generate big communication overhead. On the other hand, because heavy traffic between *Closed_loop* and *Update*, these two behaviors should be mapped to the same PE.

(*,int)	(+,int)	(-,int)	(/,int)	others
46.2%	33.5%	9.1%	7.1%	4.1%

TABLE II
Codebook operation mix.

	Open _Loop	Closed _Loop	Code-book	Update
LP_Analysis	8802	0	163	0
Open_Loop	-	272	0	0
Closed_Loop	-	-	79544	315568
Codebook	-	-	-	69112

TABLE III
Communication among top-level vocoder behaviors.

V. Conclusions

This paper presents a characterization approach to characterize the system behavior for early design space exploration. The proposed approach generates multi-metric (operation, traffic, and storage) and multi-entity (behavior, channel, port, and variable) characteristics. In order to obtain above characteristics, new algorithms, such as an algorithm for profiling traffic, are proposed. The introduced usage for exploration and experience results shows that our approach not only helps designers to intensively comprehend the application, but also provides valuable indications for early design space exploration.

References

[1] L. Cai. Estimation and Exploration Automation of System Level Design. Ph.D. Dissertation, CECS, UC, Irvine, Apr 2004.

[2] L. Cai, A. Gerstlauer, and D. Gajski. Retargetable Profiling for Rapid, Early System-Level Design Space Exploration. In *DAC*, June 2004.

[3] J. Fenlason and R. Stallman. The GNU Profiler (http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html).

[4] P. Gerin et al. Scalable and Flexible Cosimulation of SoC Designs with Heterogeneous Multi-Processor Target Architectures. In *ASPDAC*, 2001.

[5] A. Gerstlauer et al. Design of a GSM Vocoder using SpeccC Methodology. Technical Report ICS-TR-99-11, UC Irvine, 1999.

[6] R. Grehan. Code Profilers: Choosing a Tool for Anlyzing Performance. A Metrowerks White Paper.

[7] Y. Li et al. Performance Estimation of Embedded Software with Instruction Cache Modeling. In *ICCAD*, 1995.

[8] P. Lieverse et al. A Trace Transformation Techinique for Communication Refinement. In *CODES*, 2001.

[9] Y. Zhao and S. Malik. Exact Memory Size Estimation for Array Computation without Loop Unrolling. In *DAC*, 1999.