Retargetable Profiling for Rapid, Early System-Level Design Space Exploration

Lukai Cai and Andreas Gerstlauer and Daniel Gajski
Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
{lcai, gerstl, gajski}@ics.uci.edu

ABSTRACT

Fast and accurate estimation is critical for exploration of any design space in general. As we move to higher levels of abstraction, estimation of complete system designs at each level of abstraction is needed. Estimation should provide a variety of useful metrics relevant to design tasks in different domains and at each stage in the design process.

In this paper, we present such a system-level estimation approach based on a novel combination of dynamic profiling and static retargeting. Co-estimation of complete system implementations is fast while accurately reflecting even dynamic effects. Furthermore, retargetable profiling is supported at multiple levels of abstraction, providing multiple design quality metrics at each level. Experimental results show the applicability of the approach for efficient design space exploration.

Categories and Subject Descriptors: B.8 [Performance and Reliability]: Performance Analysis and Design Aids

General Terms: Performance.

Keywords: Profiling, Retargetable, System Level Design, Exploration.

1. INTRODUCTION

Recently, as system design is becoming more and more challenging due to decreasing time-to-market windows and increasing system complexities, trends are emerging to move the design process to higher levels of abstraction. System-level design, however, demands corresponding approaches that enable efficient exploration of the complete system design space in order to rapidly evaluate a large number of design alternatives in a short amount of time.

One of the most critical aspects is the feedback about design quality metrics based on which designers can make decisions. In order to meet the challenges of system-level design, estimation of metrics must be fast while providing accurate results in the sense that they are relevant and useful for evaluating and comparing alternatives. Estimation must be supported at all levels of abstraction, including early stages of the design process. At high levels of

abstraction, however, absolute accuracy is impossible. Rather, relative accuracy (fidelity) [11] allows designers to prune design space of infeasible alternatives. Furthermore, in order to evaluate complete system architectures in a comprehensive and unified manner, it must be possible to estimate a wide variety of target implementations in combination. Also, a wide range of metrics for performance, traffic, storage, etc. should be available for use in different design domains.

In this paper, we propose a novel profiling and estimation technique for system-level design based on a unique combination of dynamic profiling and static retargeting. Initial **profiling** derives the characteristics of the application through simulation of the design specification. By then coupling application profiles with target characteristics based on the designer's application-architecture mapping, profiling is **retargetable** for static **co-estimation** of complete system designs in linear time without the need for time consuming re-simulation or re-profiling. Since the system is only simulated once during the entire design process, the proposed approach is **ultra-fast** yet **accurate** enough to make high-level decisions in that it captures both static and dynamic effects. Furthermore, at each level of abstraction, the retargetable profiler delivers a set of results relevant to the design tasks at that stage of the design process for **multi-level, multi-metric** estimation.

The rest of this paper is organized as follows. An overview of related work is shown in Section 2. In Section 3, the proposed system-level estimation and exploration design flow consisting of profiling, retargeting, and simulation-estimation stages is introduced. Details of multi-level, multi-metric system profiling and retargeting are described in Section 4 and Section 5, respectively. In Section 6, experimental results that show the applicability of the approach to design space exploration are presented. Finally, the paper concludes with a summary and an outlook on future work in Section 7.

2. RELATED WORK

The estimation of embedded system has been well studied for decades. Traditionally, estimation approaches are based on either a purely static analysis or a purely dynamic simulation.

In static analysis-based approaches, upper and/or lower bounds for design metrics are computed by analyzing the code that will be running on a single target processor. In performance estimation, for example, computing the worst-case execution time (WCET) of a process [13] requires an analysis of possible program paths at the basic block level together with a micro-architecture model to determine the execution time of each basic block. Given WCETs, scheduling analysis of a group of tasks running on a processor can then determine upper bounds for overall response times [2]. Similarly, static analysis is employed to compute other metrics, e.g. to
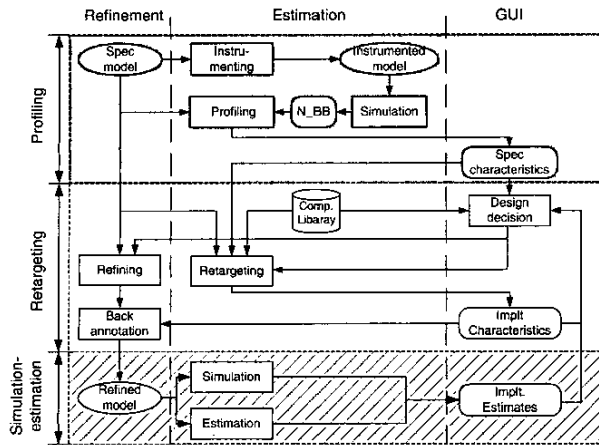
Figure 1: Estimation and exploration flow.

Figure 2: Design space exploration.

determine bounds for memory sizes [16]. In all cases, static analysis can be complex and time-consuming while the tightness of the bounds often depends on manual interference.

In dynamic simulation-based approaches, metrics are collected during simulated execution of the code. Traditional software profiling tools [6] collect profiling data while running the code on the actual target processor. Instruction-set simulators, on the other hand, execute the code on an abstracted model of the target processor. In both cases, the actual object code compiled into the processor's instruction-set has to be available and processors can only be simulated in isolation. For validation of complete systems, co-simulation of multiple processors, possibly at multiple different levels of abstraction, is supported [3, 7]. Finally, there are several approaches speeding up successive re-simulation of the same design at different abstraction levels by driving slow low-level simulations with traces collected from fast, abstract simulation runs [14, 12]. In all cases, however, time-consuming simulation of the system design is necessary for each design alternative.

In comparison to the proposed approach, although traditional approaches can provide more accurate results, they are too slow for exhaustive initial design space exploration. The proposed retargetable profiling approach can be used as their complementary approach at high abstraction levels. Because it is ultra-fast and relative accurate, it allows designers to exhaustive explore the initial design space and to prune the design space of infeasible alternatives before traditional estimation.

In contrast to implementation-dependent estimation, there is only a limited number of approaches that aim to derive implementation-independent characteristics of design specifications for system level design. Traditional profilers such as [6] and [9] usually provide target/host machine-dependent characteristics. Even though such profilers can produce some implementation-independent characteristics, they only support operation-related data such as function call statistics. In comparison, our profiling approach is targeted for system level design, and it computes not only operation-related characteristics, but also traffic- and storage-related characteristics.

3. DESIGN FLOW

We propose an estimation and exploration flow, which is shown in Figure 1. The flow is based on a **explore and trim** paradigm for design space exploration (Figure 2). As design progresses through profiling, retargeting, and simulation-estimation stages, the design
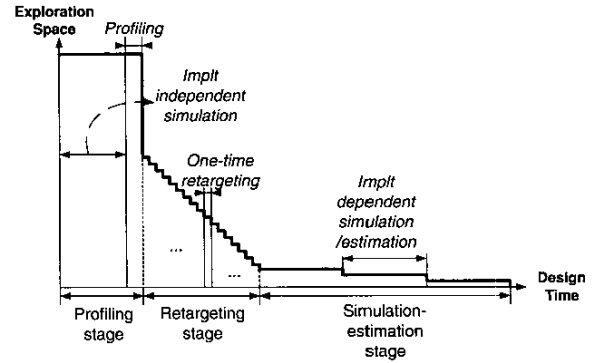
space is gradually trimmed and pruned of unsuitable design alternatives until a final optimal solution is reached.

In each step, design starts from a specification of the desired system functionality. In the profiling stage, the specification is instrumented and simulated to collect execution counts that capture the dynamic behavior of the application at the basic block level (N_{BB}). Because this stage requires simulation for the specification, we call it dynamic. Using the counters collected during simulation together with a static analysis of the code, a profiling of the specification then computes the **specification characteristics**. Specification characteristics are implementation-independent and provide information about the inherent characteristics of the application. Based on these specification characteristics, the design space can be reduced to a large part. For example, if the specification does not contain any floating point operations, allocating dedicated floating point processors is counterproductive.

In the retargeting stage, designers allocate a target architecture of processing elements (PEs) and/or busses from a component library by matching specification characteristics and component attributes. Given the allocated architecture or a predefined platform, designers then map the computation and communication in the specification onto PEs and busses, respectively. A retargeting of the specification then computes the **implementation characteristics** for computation and communication by coupling the design decisions and specification characteristics. These characteristics are implementation-dependent and represent the characteristics of the system design reflecting the designer's decisions. In an iterative process, the retargeting stage is executed repeatedly for different decisions in order to prune the design space of unpromising design alternatives. Because this stage doesn't require simulation, it is a purely static analysis. As will be explained later, retargeting is very fast in comparison with simulation and profiling. Therefore, retargeting enables designers to explore many alternatives and tr im a large part of the design space in a short amount of time.

Finally, the most promising design alternatives remaining after the retargeting stage are then evaluated further in the simulation-estimation stage. For each alternative, a refinement tool [15] generates a refined model of the design from the specification, integrating and implementing the corresponding design decisions. In the process, implementation characteristics are back-annotated into the resulting model. By simulating the refined model, accurate **implementation estimates** including dynamic implementation effects not observable by profiling and retargeting the specification (e.g. bus contention or dynamic scheduling) are generated. In order to derive the accurate implementation estimates, traditional estimation approaches introduced in section 2 can also be applied in this

stage. Using the implementation estimates, a final evaluation of the remaining alternatives can be performed, possibly requiring to return to the retargeting stage. Implementation estimates provide the accuracy of traditional simulation- or estimation-based approaches while similarly requiring time-consuming simulation or analysis of each design alternative. However, as the design space has been reduced down to a few alternatives through profiling and retargeting, exhaustive simulation and anlysis becomes feasible.

4. PROFILING

The profiling stage generates the specification characteristics from the system specification model. In general, design models are usually captured in the form of a system-level design language (SLDL). In our case, specification models are written in the SpecC SLDL [1]. However, the concepts apply equally to any other C-based SLDL such as SystemC [10].

Profiling computes the specification characteristics for each computation and communication entity in the specification. In the case of SpecC, computation in the form of C code is encapsulated as behaviors. Behaviors can be composed hierarchically in a sequential or concurrent fashion. Behaviors communicate through ports connected to shared variables or channels where a channel provides complex communication services to the behaviors through methods declared in its interface.

Given the execution counts of each basic block collected during simulation and the code for the basic blocks in each entity, profiling attaches raw characteristics $r_{i,d}, i \in I, d \in D$ to each behavior, port, variable, and channel in the specification where characteristics are computed hierarchically by summation over the characteristics of an entity's children. 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. SpecC defines 26 basic, standard data types where data types are further divided into integer data types, floating point data types, and other data types. In addition, profiling can optionally treat composite, user-defined data types (such as arrays or structs) separately, expanding D dynamically as needed. Otherwise, user-defined data types will be mapped down to the basic data types of the individual elements they are composed of.

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.

4.1 Operation

Operation characteristics (spec.) signify the complexity of the computation in the specification. Therefore, they are attached to behaviors as the computational units of the system. Each operation characteristic corresponds to a certain operation of a certain data type (as determined in C by the type of the result). Item types for operation characteristics are defined as the 84 different operations available in SpecC. They are further classified into ALU operations ('+', '&&', '<<', '>=', etc.), memory access operations ('=', '->', etc.), control operations ('if', 'for', 'f()', etc.), and others (braces and other syntactical overhead). In addition, similar to data types, profiling can optionally treat global functions as special operation types instead of mapping them down to the operations inside the function. In any case, operation metrics can be computed for different classes of operations, different categories of data types, or as the sum over all operation and data types.

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.

Dynamic operation characteristics are defined as the number of operations executed by each behavior during simulation. Note that in the presence of recursive function calls, dynamic operation characteristics need to be computed by solving a set of linear equations details of which are omitted due to space reasons [4]. Dynamic operations represent the **computational complexity** in the system which is related to performance issues. By identifying the most complex behaviors, they can point the exploration to the most critical aspects and the best candidates for optimization. Furthermore, the mix of operations in all or parts of the system can be used to determine the type of processor used for implementation, e.g. a DSP with a hardware multiplier for multiplication-intensive behaviors.

4.2 Traffic

Traffic characteristics (spec.) 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 for behaviors is computed as the sum of the traffic over all their ports. Note that at each abstraction level, communication is modeled differently. For example, at the transaction level [10], behaviors communicate via abstract, complex channels whereas variables connecting behaviors at the bus-functional level represent physical bus wires.

Static traffic characteristics are defined as the number of connected ports of a certain type. For a behavior's port, they 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. In all cases, static traffic characteristics represent **connectivity complexity**. For example, at the application level connectivity 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. At the bus-functional level, connectivity complexity relates to fan-in/fan-out and bus wire capacity.

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. Note that a special port-parameter binding algorithm resolves port accesses in the presence of recursive calls or multiple invocations of the same function. Details of this algorithm can be found in [4]. In summary, dynamic traffic characteristics represent **access complexity**. For example, at the application level, dynamic accesses relate to the traffic incurred for a shared memory implementation of communication between dependent behaviors. At bus-function level, they relate to the traffic over pins of the bus, e.g. data traffic in case of the data bus.

4.3 Storage

Storage characteristics (spec.) signify the amount of memory 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. In SpecC, this includes variables declared at the behavior/channel level and static variables inside functions. Static

storage represents **static memory requirements**, i.e. memory that needs to be allocated globally for the whole lifetime of the system.

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 memory allocated dynamically on the heap during runtime (e.g. via `malloc()` calls). Note that in contrast to other characteristics, dynamic storage requirements are computed hierarchically as the maximum over all children at each level.

5. RETARGETING

The retargeting stage computes implementation characteristics of an implementation of each design entity based on design decisions made by the user. Decisions include component allocation (PEs and busses) and entity mapping (behaviors and variables to PEs and channels to busses).

Given the design decisions and the specification characteristics computed during profiling, retargeting attaches implementation characteristics $e_{i,d}$, $i \in I$, $d \in D$ to each behavior, port, variable, and channel. These characteristics are computed by multiplying specification characteristics $r_{i,d}$ with weights $w_{i,d}^c$ for a mapping of the design entity to component c. Weight tables have to be defined for each component in the library. Depending on the component, they can be derived from the component's data sheet or from accurate simulations of selected, typical code kernels on the target component. In addition to the standard weights for basic data and item types stored in the library, the designer can manually tune weights for retargeting. Furthermore, the designer can specify weights for custom data and item types collected during profiling instead of mapping them down to the basic data and item types they are composed of.

Based on the specification characteristics, implementation characteristics can be classified into operation, traffic, and storage categories, and each category can be further subdivided into static and dynamic metrics. Static and dynamic implementation metrics E in each category are then computed by summation of the weighted characteristics e over subsets of item and data types:

$$E = \sum_i \sum_d (r_{i,d} \times w_{i,d}^c)$$

where w^c is the weight table for component c from the library. Because $r_{i,d}$ has been computed in the profiling stage and $w_{i,d}^c$ is predefined in the weight table, retargeting avoids the time-consuming simulation and profiling. Due to the simplicity of the computation, retargeting is fast and its time complexity is $O(n)$ where n is the number of behaviors, ports, variables, and channels in the system.

Similar to specification characteristics, implementation characteristics are computed hierarchically by adding implementation characteristics of children at each level. Retargeting supports two modes for hierarchical computation: analysis mode and estimation mode. The analysis mode provides mapping-independent results. It computes characteristics for each entity on each allocated component assuming that the whole entity (including children) is mapped to the target component. Results can be used by designers after allocation to select the most appropriate component to map each entity to. Estimation mode, on the other hand, computes characteristics based on both allocation and mapping decisions. For each entity, it generates characteristics on each target for those parts of the entity that are mapped onto this component. Results can therefore be used to evaluate mapping decisions.

5.1 Operation

Operation characteristics (implt.) are computed for behaviors mapped onto target PEs. For *static operation* characteristics, PE weights define the number of instruction or control words for each operation where, in the case of custom hardware, the number of control words is equal to the number of control states. By multiplying the characteristics with the PE's instruction or control word width, metrics for program memory size or size of the custom hardware controller can be computed, respectively. Therefore, static operation metrics represent **code size** requirements for each behavior.

PE weights for *dynamic operation* characteristics define the number of clock cycles needed to execute each operation. By multiplying the number of cycles with the clock period, **execution time** metrics for behaviors can be derived. In a similar manner, **power consumption** metrics can be computed through energy per cycle weights.

5.2 Traffic

Traffic characteristics (implt.) are computed for ports of behaviors mapped to PEs and for variables and channels mapped to busses. A PE's traffic weight table is equivalent to its storage weight table (see Section 5.3) and it defines for each data type the number of machine characters transfered over the PE's bus. Together with the PE's machine character width and bus bandwidth, the amount of data and time needed for each transfer are computed. A bus' weight table, on the other hand, defines the number of bus cycles needed to transfer each data type over the bus. Dividing bus traffic characteristics by the bus bandwidth, required communication time is computed. In all cases, static and dynamic traffic characteristics represent **communication delays**. For example, at the application level, communication delays for message passing or shared memory implementations are estimated, respectively. At the bus-functional level, on the other hand, characteristics for the traffic over the data bus pins and wires provide actual bus access times.

5.3 Storage

Based on a mapping of behaviors and channels to PEs, storage characteristics (implt.) determine the memory size requirements in each PE. A PE's storage weight table defines the number of machine characters needed to store variables of different data types. Multiplying storage characteristics with the PE's machine character bit-width, required memory size metrics are computed. Static and dynamic metrics for local and global storage therefore represent **static memory size**, **stack size**, and **heap size** requirements.

6. EXPERIMENTAL RESULT

A retargetable profiler supporting instrumentation, profiling, and retargeting has been implemented and integrated into our system design environment. We applied the estimation and exploration methodology using the profiler to the design examples of a voice codec for mobile phone applications (vocoder) [8] and a JPEG encoder [5]. The vocoder example demonstrates the usage of the profiler in the design space exploration and the profiler's ultra-fast attribute. The JPEG example demonstrates accuracy and fidelity [11] of the retargeting.

6.1 Vocoder

The vocoder is assumed to be part of a mobile phone baseband platform using a Motorola ColdFire processor as the CPU. The vocoder specification consists of appr. 13,000 lines of code. It encodes and decodes a frame of speech every 20 ms. For a testbench

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

Table 1: Computational complexity of top-level vocoder behaviors.

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

Table 2: *Codebook* operation mix.

that excercises the design with 163 frames, this translates to a total timing constraint of 3.26 s.

The vocoder is a computation-dominated design. Therefore, design space exploration is focused on computation design. We estimated an upper bound for the communication overhead using the profiler by mapping all system communication onto a ColdFire bus. Leaving a margin for the estimated communication delay of appr. 280 ms, we derived a timing constraint of appr. 3 s for the vocoder computation.

In the profiling stage, we instrumented, simulated, and profiled the vocoder specification to generate the specification characteristics. Table 1 shows the computational complexity for the vocoder's five top-level behaviors in millions of operations (MOp). Note that as a typical multimedia application, parallelism in the vocoder is limited and at the top level behaviors execute sequentially in a loop. Therefore, there is little promise of exploiting concurrency and design should focus on optimizing the critical parts of the behavior sequence. As the profiling results show, the *Codebook* search behavior is by far the most critical vocoder block. The profiler also provides the mix of operations in the *Codebook* behavior (Table 2; 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 3). 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.

For further exploration in the retargeting stage, we allocated a system architecture with three PEs: in addition to the Motorola ColdFire CPU running at 60 MHz, we selected a DSP (Motorola DSP56600 at 60 MHz) and a custom hardware processor (100 MHz) to explore vocoder speedups. Mappings of eight top-level behaviors (five top-level vocoder behaviors plus three levels of hierarchy of behaviors inside *Codebook*) to every PE were evaluated. Using the scripting capabilities of the design environment together with the profiler, we ran an exhaustive search of all $3^8 = 6561$ design alternatives. Running on a Pentium IV Linux PC at 2.0 GHz, the complete search was finished in 3:15 h. It contains one time simulation (2.23 s), one time profiling (8.41 s) and 6561 times retargeting (0.8 s for each) and mapping (0.97 s for each) respectively.

Figure 4 shows computation time vs. cost for all design alternatives. For both ColdFire and DSP a fixed cost of 20 each was assigned for the manufacturing cost. For custom hardware, a linear cost function with a base cost of 20 and an additional cost of 1$ per 10 static operations of code complexity was assumed to estimate costs of control logic and design overhead. If no behavior is
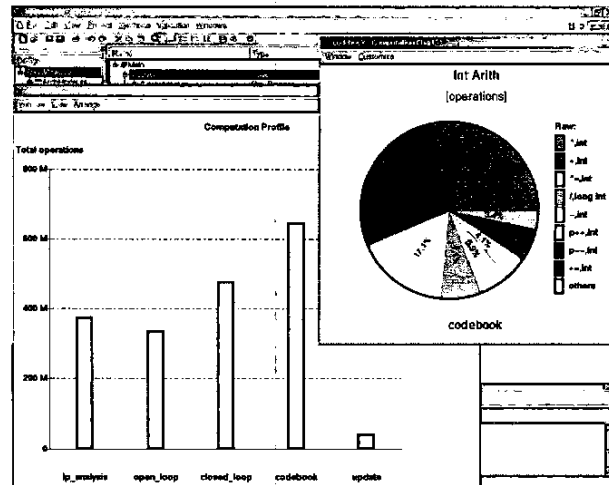

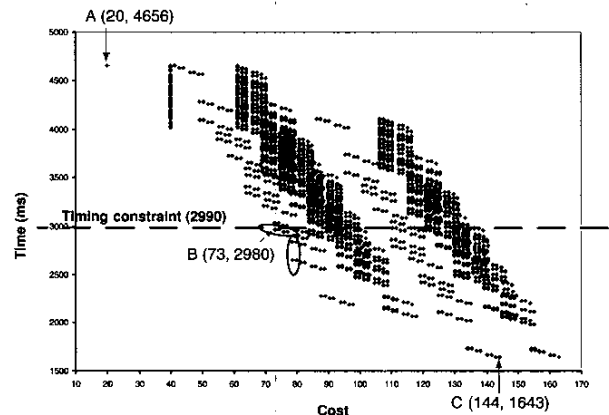**Figure 3: Vocoder specification characteristics GUI.**

Figure 4: Vocoder design space.

mapped to a PE, it's cost is assumed to be zero. A pure software solution running on the ColdFire (A) is the cheapest design but has the largest delay. A pure hardware solution (C) is the fastest design. An optimal solution in the sense of the cheapest design that meets the timing constraint is a mapping of *Open_loop*, *Closed_loop*, and *Search_10i40* (part of *Codebook*) behaviors to hardware while the rest of the system is running on the ColdFire (B). The circled parts of the design space mark candidates for further evaluation via simulation of refined implementation models. Comparing retargeting results with a previously implemented DSP-HW solution of the vocoder [8], implementation characteristics are accurate to within 15% of the actual delays.

6.2 JPEG Encoder

JPEG [5] is an image compression standard. It is designed for compressing either full-color or gray-scale photographic images. The JPEG encoder consists of four behaviors running sequentially: *HandleData (HD)*, *DCT (D)*, *Quantization (Q)*, and *HuffmanEncode (HE)*. The JPEG specification contains around 2,000 lines of code. The testbench for the design encodes pictures with sizes of 116×96 pixels (corresponding to 180 blocks of 8×8 pixels).

285

Alternatives				Estimates (ms)	Char. (ms)	Diff
HD	D	Q	HE			
SW	SW	SW	SW	205.00	199.44	2.71%
SW	SW	SW	HW	184.77	177.18	4.11%
SW	SW	HW	SW	189.00	180.87	4.30%
SW	SW	HW	HW	168.77	158.61	6.02%
SW	HW	SW	SW	73.35	76.79	-4.69%
SW	HW	SW	HW	53.12	54.53	-2.65%
SW	HW	HW	SW	57.35	58.22	-1.52%
SW	HW	HW	HW	37.12	35.96	3.12%
HW	SW	SW	SW	183.23	176.92	3.44%
HW	SW	SW	HW	163.00	154.66	5.12%
HW	SW	HW	SW	167.23	158.35	5.31%
HW	SW	HW	HW	147.00	136.09	7.42%
HW	HW	SW	SW	51.58	54.27	-5.22%
HW	HW	SW	HW	32.01	29.93	-2.11%
HW	HW	HW	SW	35.70	38.84	-0.34%
HW	HW	HW	HW	15.35	13.44	12.44%

Table 3: Comparison of implementation characteristics and implementation estimates for JPEG encoder delays.

We allocated a system architecture with two PEs: a Motorola DSP56600 (SW) running at 60 MHz and a custom hardware (HW) running at 80.8 MHz. By mapping four behaviors to two PEs in different ways, we derive 16 (2^4) design alternatives.

We computed both implementation characteristics and implementation estimates representing the encoding delays for all design alternatives. Implementation characteristics are computed by the proposed profiler. For the implementation estimates, we estimated the delays on SW by converting SpecC to C code, compiling the C code to the assembly code, and running the assembly code on the DSP56600's customized instruction set simulator. We estimated the delays on HW by simulating manually written RTL models.

Table 3 displays the computed encoding delays for 16 design alternatives. The first four columns represent the designer's behavior-PE mapping decision for the four behaviors. The delay in column *Estimates* represents implementation estimates. The delay in column *Char.* represents implementation characteristics. Their difference is displayed in column *Diff*.

Table 3 demonstrates that the implementation characteristics computed by the proposed profiler are accurate to within 12.5% of implementation estimates for the JPEG example.

We also computed the fidelity [11] of the proposed approach. The fidelity is defined as the percentage of correctly predicted comparisons between design alternatives. If the estimated values of a design metric for two design alternatives bear the same comparative relationship to each other as do the measured values of the metric, then the estimate correctly compares the two alternatives. Based on Table 3, we compute the fidelity of the proposed approach by comparing the implementation characteristics with the implementation estimates. For JPEG encoder example, the fidelity of our approach is 100%.

7. SUMMARY AND CONCLUSIONS

In this paper, we present a system-level estimation approach based on a novel combination of dynamic profiling and static retargeting. In an initial profiling stage, one-time simulation of the specification is done in order to collect specification characteristics about the dynamic behavior of the system. In the retargeting stage, by using specification characteristics together with a static analysis of the code, retargeting to different implementations for accurate co-estimation of whole system designs is done statically in linear time.

This ultra-fast approach enables initial, exhaustive exploration of design space with the results that are accurate enough to prune out infeasible design alternatives. Therefore, it is an ideal complementary solution for the traditional estimation approaches. Retargetable profiling is applied to all computation and communication entities in the description in a general manner. Therefore, the approach can be applied to models at all levels of abstraction. Furthermore, it computes a number of useful and relevant quality metrics for each entity, enabling efficient design space exploration and guiding the user in the design process.

In the future, we want to extend the retargetable profiler to provide additional metrics, including statistical information (minima, maxima, standard deviations, etc.) for each metric, and to handle dynamic micro-architecture features like caching or pipelining more accurately. Furthermore, we are investigating possibilities for re-profiling of refined models without re-simulation using profiling data from earlier design steps.

8. REFERENCES

[1] SpecC website (http://www.specc.org).

[2] G. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer, 1999.

[3] Cadence. VCC. http://www.cadence.com/products/vcc.html.

[4] L. Cai and D. Gajski. Introduction of design-oriented profiler of SpecC language. Technical Report ICS-TR-00-47, UC Irvine, 2001.

[5] L. Cai, J. Peng, and D. Gajski. Design of a JPEG Encoding System. Technical Report ICS-TR-99-54, UC Irvine, Nov 1999.

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

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

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

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

[10] T. Grötker et al. *System Design with SystemC*. Kluwer, 2002.

[11] F. Kurdahi et al. Linking Register-Transfer and Physical Levels of Design. *IEICE Transactions on Information and Systems*, Sept. 1993.

[12] K. Lahiri et al. Performance Analysis of Systems With Multi-Channel Communication Architecture. In *International Conference on VLSI Design*, 2000.

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

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

[15] J. Peng et al. Automatic Model Refinement for Fast Architecture Exploration. Technical Report CECS-IR-02-14, UC Irvine, Apr 2002.

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