Annotation Methods and Application Abstractions

R. Shankar[1], H. Kalva[1], A. Agarwal[1], A. Jain[2]
[1]Center for System Integration, Florida Atlantic University, Boca Raton, FL
[2]iDEN Group, Motorola, Plantation, FL

Abstract – **Embedded System complexity is increasing with time. This impacts negatively the product development cycle (PDC). The increasing software and hardware concurrency, and need for architectural optimization, will also complicate PDC. Innovations in PDC are needed to reduce product cost and time-to-market. Modeling a system can help in addressing architect's design concerns. To be effective in reducing PDC, a system model must incorporate performance parameters and quality of service (QoS) metrics of the modeled components. We discuss three different annotation methods for obtaining such metrics.**

I. INTRODUCTION

A system design process is inherently complex. The design involves multiple representations, multiple (design) groups working on different design phases, and a complex hierarchy of data and applications [1]. The different groups bring different perspectives towards system design. The system or product inconsistencies primarily arise out of lack of appropriate communication among various design teams. For example, concerns of a hardware (HW) design engineer are different from that of a software (SW) designer and they are often unable to understand and help address the problems of the other [2]. Such constraints lead to increase in PDC and product development cost, thereby reducing system design productivity [3].

To enhance this declining productivity, one will have to exploit the principle of "design-and-reuse" to its full potential [4]. Then, a system (subsystem) would be composed of reusable sub-systems (components). For example, Network-on-Chip (NOC) architecture can serve as a reusable communication sub-system for an embedded device [5]. This NOC architecture may comprise of components such as, routers, input and output buffers, network interface, switch, virtual channel allocator, scheduler and a switch allocator. To develop a system from such reusable components, one will have to design and develop variants of each component. For example, buffer size is a customizable parameter for an input buffer. Similarly, scheduling criteria provides customizability for scheduler component. A system architect estimates performance and QoS parameters for various system configurations. Components need to encapsulate their performance metrics for various useful parameter combinations, in order to help the architect to make informed decisions. We propose that a system be modeled in advance of the architect's design phase. Such a model is analyzed to ensure system functionality at an abstract manner. This model can then be ported to architect's design phase, for analyzing the performance of a system and for estimating the resources needed for mapping an application on to the system. This model must allow one to manipulate a set of parameters to fine tune system performance. Such a system model needs to have high level representation of various performance and QoS parameters for subsystems and components. These performance and QoS parameters can then be traded-off against each other in the system model, to yield a global optimization. Such a model at the design phase will allow one to make key decisions and reduce the scope of the multidimensional search space. These key decisions may include the number of processors, HW/SW partitioning, estimated performance values for new components, and the use of existing components in software or hardware.

Such design decisions have the potential to significantly enhance productivity of system design. Such system modeling, analysis and design will not be an effective solution until we have a mechanism for abstracting the parameter values from different applications and components of the system, and to represent these values in our model. For example, we may abstract the performance parameters for a multimedia application (e.g. MPEG4) on a particular architecture and capture the resource requirements in terms of processor speed, cache size, protocol requirements, power consumption, latency, and silicon area.

II. PERFORMANCE ANNOTATION IN DESIGN PHASE

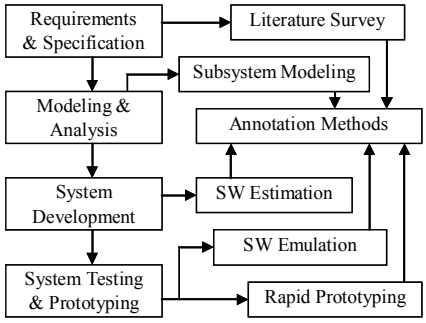

Figure 1: High level system design process

System design process comprises of the phases of requirement and specification, and system design, development and prototyping. Our focus in this paper is on the system design phase. We propose to provide annotation of QoS and performance metrics, to help system architect make appropriate design decisions. We document various annotation methods. These methods include annotation from literature survey, software estimation, software emulation, rapid prototyping, and subsystem modeling. See Figure 1. A

1-4244-1039-8/07/$25.00 ©2007 IEEE

system model then will be a more realistic model for performing system level analysis.

A. Literature Surveys

Literature survey (from published papers, technical specifications, and reviews) can often result in useful abstractions. Such information may relate to performance of a particular component, application, or the whole system. However such information may be incomplete and may require additional effort to map such information to a high level system model with a sub-system modeling tool such as Matlab; unknown information is estimated there with one of the other methods and combined [6]. It also involves a thorough study of various benchmarks and extraction of their abstractions.

B. Software Emulator

An example is the ARMulator from ARM Inc. We can flash a particular application on to an ARMulator and capture various required parameters. One will have to specify the target application in a high level language such as C before collecting the performance parameters. Such tools often provide platform specific measures [7]. Such measures are good enough to make initial decisions even if the targeted platform is different, since precise values are not necessary at this stage. Consider the fact that today's system architect may still use excel spreadsheets with estimates gleaned from various groups.

C. Software Estimation

Software estimation has been a popular way of abstraction. Researchers have used Lines of code (LOC) for estimating the time/cost required for software development. Cocomo cost model is one such popular method. Our method involves separation of computing and communication costs (see below) of the software and concurrency cost of mapping it to an architecture [8]. Others have proposed a function based cost/time estimation model.

D. Rapid Prototyping with FPGA

FPGA industry has evolved and progressed rapidly over last two decades. In the past FPGA was mainly used for developing hardware. Today's FPGA can also be used as a software or system prototyping environment. We have used rapid prototyping with ImpulseC to facilitate HW and SW co-prototyping. Impulse C is a set of platform libraries written for C. Using these libraries, the developer can create partitioning, and rapidly debug hardware and software simultaneously on the same virtual prototype [9].

E. Sub-System Modeling

We have used Matlab to model subsystems, such as VOIP and OMAP, to extract relevant parameters. This method is useful when lack of information on some components precludes one from estimating the subsystem metrics.

In this paper we describe three of these annotation methods. We show their application with different examples. These examples demonstrate their application in system design.

III. Annotation with Software Estimation

A module's COMM (Communication) and COMP (Computation) costs are calculated in terms of high-level attributes of RWXM (Read, Write, Execute and Multiply). COMP is calculated by determining the number of X and M instructions and their cycle usage. COMM is calculated by determining number of R and W instructions and their cycle usage. COMP is the execution time required by a software module to execute on a processor (excludes the CPU time associated with memory access). COMM is the time it would require to load instruction and data from memory/cache. COMP is eventually used to determine the size and/or the number of cores required to execute the software in a certain timeframe. COMM similarly is useful to determine the size and speed of cache and memory needed. This analysis is similar to source level timing scheme in Wolf [10]. The time function there has been given here a practical shape using a set of rules. Software behavior changes with respect to data input. There could be several data dependent control structures in the modules being analyzed [10]. In such cases we estimate a probability that the control structure is accessed. COMP and COMM of the control structure is multiplied by the probability of its usage. The probability factor here can be manipulated to simulate and generate multiple use cases for modules (such as worst case, best case and typical case scenarios). Size of data being processed also affects the COMP and COMM of the module. Data can only be known at run-time; however data can be considered constant for different use cases [10]. Analysis results of any scenario can be taken into consideration, depending on the type of multi-core decomposition required. This eliminates the need for dynamic analysis, while providing a simple, controlled environment. We recommend the use of results from typical use case scenarios.

A. Application

We have used the example of digital camera software (DCS) [11] to process a small 8 x 8 pixel size image. We computed the COMM and COMP using static analysis methods. In order to estimate hardware resource (processor

and bus) usage, the COMP and COMM are multiplied with the number of CPU cycles needed to complete the execution of four abstract instruction types, viz., R, W, X, and M. Tables 1 and 2 illustrate the number of cycles we assumed for RWXM costs. Note that CPU may or may not be actively involved in each of the CPU cycles of RWXM; the 'CPU cycle' merely indicates a unit of time used to express different execution/processing times.

TABLE I: CPU CYCLE TIME UNITS NEEDED FOR COMMUNICATION TYPE INSTRUCTIONS

COMM	CPU Cycles-Data Access from Memory	CPU Cycles – Data Access from Cache
Read	2	1
Write	2	1

TABLE II: CPU CYCLE TIME UNITS NEEDED FOR COMPUTATION TYPE INSTRUCTIONS

COMP	CPU Cycles
Execute	1
Multiply	10

The RWXM cycle estimates associated with software units in all the modules of the digital camera software are provided below. The base RWXM instruction estimate of a software unit is obtained by applying a set of rules to every LOC (Line of Code) of the C program, while considering the usage probability of a software unit. The final RWXM instruction estimate of a software unit is determined by calculating the number of times the software unit is invoked. In tables shown below 'Iteration' refers to the number of times the corresponding software unit is invoked. We restricted the final RWXM estimates to be whole numbers; the base RWXM instruction estimates were not similarly restricted as they are based on probabilities.

B. CCD Cost Analysis and Results

Table 3 enumerates the base RWXM instruction and iteration counts for CCD, one of the software modules of DCS. Table 4 provides the final RWXM instruction estimates for CCD. Table 5 shows the estimated cost associated with CCD as per Tables 1 and 2. Figure 2 shows the COMP and COMM for all the modules that make up DCS. We determined CODEC as the module of interest because its COMP and COMM are substantially higher than those of other modules. We used this annotated information to map CODEC to multiple concurrent processors. A substantial speed-up was achieved [8].

TABLE III: CCD - Base RWXM Instructions

	R	W	X	M	Iterations
Preprocessor statements	0	2	1	0	1
Initialize Function	9	17	0	0	1
Capture Function	1686	882	1200	240	1
R Pop Pixel Function	9.2	6.125	5.2	1	80
G Pop Pixel Function	9.2	6.125	5.2	1	80
B Pop Pixel Function	9.2	6.125	5.2	1	80

TABLE IV: TOTAL RWXM INSTRUCTIONS

	COMM (RW) Instructions	COMP (XM) Instructions	Total (RWXM) Instructions
Preprocessor statements	2	1	3
Initialize Function	26	0	26
Capture Function	2568	1440	4008
R Pop Pixel Function	1226	496	1722
G Pop Pixel Function	1226	496	1722
B Pop Pixel Function	1226	496	1722
Total (all functions)	6274	2929	9203

TABLE V: ESTIMATED COST, IN NUMBER OF CPU CYCLES

	COMM	COMP	Total Cost
Preprocessor statements	6	1	7
Initialize Function	78	0	78
Capture Function	7704	3600	11304
R Pop Pixel Function	3678	1216	4894
G Pop Pixel Function	3678	1216	4894
B Pop Pixel Function	3678	1216	4894
Total (all functions)	18822	7249	26071

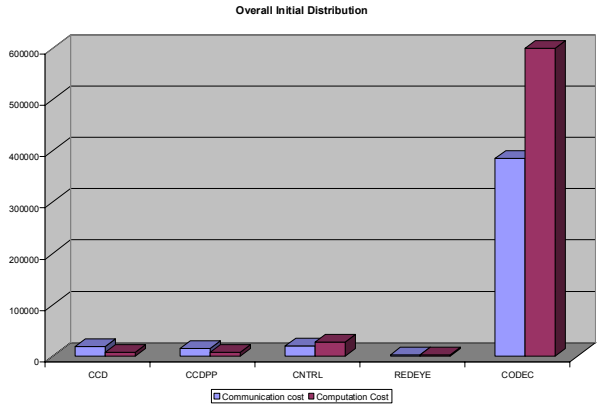

Figure 2: COMM and COMP comparison of DCMImage size: 8 x 8.

IV. ANNOTATION WITH FPGA RAPID PROTOTYPING

A system is defined in terms of processes. These processes can be declared as a hardware or software process. If the component is defined as a hardware process, then it gets implemented on the FPGA hardware. If defined as a software process, then the component is implemented in a soft-core processor. ImpluseC code for all the processes in a system is developed in any C++ compiler.

ImpulseC provides multiple methods for process communication, such as registers, streams, signals, and shared memory. Once these processes are developed they can be mapped on to the FPGA without the need for writing hardware description language (HDL) code. Execution of the processes on the FPGA will yield appropriate performance metrics. Figure 3 shows the implementation of an application on to Xilinx ISC platform. Performance parameters extracted from FPGA implementation will be platform dependent (based on the underlying FPGA architecture). However, they will have sufficient information in providing an estimate to aid in the early design phase. We extracted parameters such

as gate count (silicon area), maximum clock frequency (design latency), slack time, and critical path time. One can also use utilities such as XPower to extract power consumption information of a system or its processes.

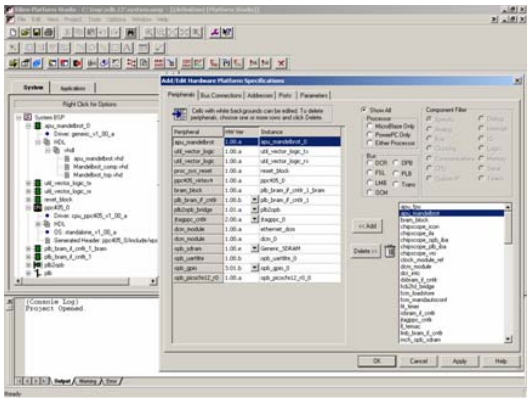

Figure 3: Application implementation in Xilinx

A. *Design Example and Results*

We modeled a Network-on-Chip (NOC) architecture. NOC is a multi-core packet switched communication backbone for use in designing complex embedded systems [12]. Figure 4 represents 3×3 mesh based NOC block diagram. NOC may have the following components: Input and Output Buffers (B), Producer (P), Consumer (C), Node (N), Scheduler (S) and Network-Interface (NI). Each of these components can be modeled either as a hardware or a software process. This NOC architecture offers various performance tradeoffs. For example, one would be able to analyze the impact of a particular routing strategy or scheduling criteria in terms of latency, power and area. In practice, about 60% of the area of the communication backbone will be occupied by the buffers. For a 3×3 mesh based NOC, there will be ninety buffers. Thus, the buffer size is a key parameter of NOC architecture. We prototyped Input Buffer as a hardware process. We provide in Table 6 the computed area of B for sizes of one to ten. This size parameter became an input parameter to a high level abstract model built to analyze effect of buffer size on system performance. Such an analysis allows one to make area-performance tradeoffs.

We designed the Input Buffer as a smart buffer with a built-in scheduler. The scheduler forwards the data as per certain scheduling criteria. Further, a buffer can be designed as 32-bit or 64-bit wide. In the 64-bit buffer, the chip area will increase but the number of packets to be transmitted will reduce to about half as compared to the 32-bit buffer. Therefore, it is likely that the latency of a system will reduce by transmitting a bigger packet. But it will require support for more parallel transmission lines (bus) in case of a parallel data transmission.

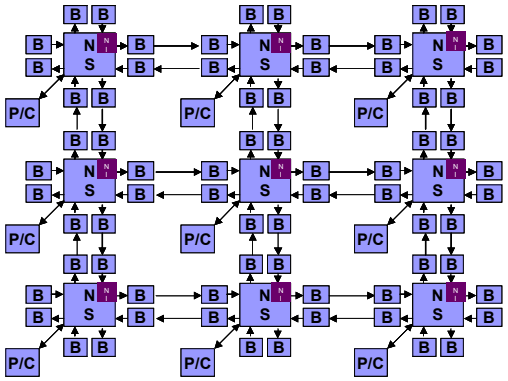

Figure 4: NOC architecture block diagram

TABLE VI: GATE COUNT FOR VARIOUS BUFFER SIZES

Buffer Size	Gate Count for 32-Bit	Gate Count for 64-Bit
Size 1	11,427	16,733
Size 2	13,381	20,135
Size 3	14,510	22,207
Size 4	14,920	23,292
Size 5	15,526	24,594
Size 10	16,108	24,940

V. ANNOTATION WITH SOFTWARE EMULATION

The resources consumed by an application depend on the complexity of the application, the target platform, and any data dependencies. Multimedia applications have significant data dependencies and hence resource consumption is heavily data dependent. The application abstraction methodology used had two components: 1) resource abstraction; and 2) data dependency abstraction. The resource abstraction methodology used software profiling with Intel VTune Performance Analyzer to determine the resources consumed for a given platform [13]. The metrics used for resources consumption were the reads (R), writes (W), executions (E), cache hits and cache misses. Data dependencies were abstracted by measuring resource consumption for different data input streams and then modeling the variation in resource consumption due to data dependencies. An application is then annotated with the resource consumption rate and the data dependency model. The annotated application also used a model of the target platform to estimate resources. High level target platform metrics such as the processor speed, and average number of instruction retired per second are sufficient to estimate the resources consumed for that platform. We developed a methodology to estimate resources for multimedia applications.

A. *Resource Estimation Methodology*

We demonstrate the methodology with the H.264 video decoder. The key factors that influenced the approach are: 1) data dependencies and 2) target architecture. We abstracted the data dependencies using a process called Bitstream

Abstraction (BA). The decoder is componentized and component level resource requirements determined in a process called Decoder Abstraction (DA). We used BA together with DA to develop a resource estimation model. We describe below the methodology and the integrated flow.

Content dependencies are inherent in video coding and resources required to encode/decode a video also depend on the content and the quality of the video. Our BA refers to the characterization of a compressed Bitstream with a few parameters that significantly influence the amount of computing resources required to decode the bitstream. The BA is specific to compression algorithms. The BA developed for the H.264 video used the following key Bitstream metrics: IntraMB 16x16, IntraMB 4x4, Inter MB 16x16, Inter MB 16x8, Inter MB 8x16, Inter MB 8x8, skipped MB, and non-zero coefficients. The per-frame averages of these metrics were used to represent the complexity of a Bitstream. Our results showed that the variation in resource consumption is strongly correlated with variation in one or more of the BA metrics. Modeling these variations will lead to resource estimation. The DA is the process of representing the decoder complexity with target platform independent metrics. We abstracted the decoder complexity by abstracting the complexity of the components in the decoder. The component complexity description should enable resource estimation for a given architecture. We used three metrics: R, W. and E. For the high level resource estimation we considered, further sub-classifying the instructions was not necessary. The Intel VTune performance analyzer was used to obtain the metrics. This was then used to develop a resource estimation model.

B. *Experimental Results*

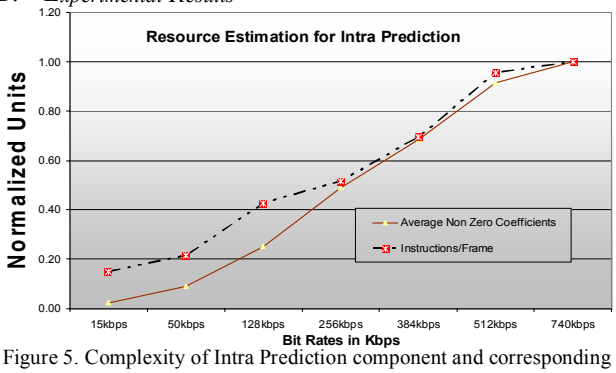

Figure 5. Complexity of Intra Prediction component and corresponding data dependencies in H.264

Nokia H.264 baseline encoder and decoder were used at different bit rates to develop a model for resource estimation. Nokia H.264 employs frame and macroblock level rate control. The experiments were conducted with three different videos, Akiyo, Foreman, and Football, with same resolution of 176 x 144 and 15 Fps. The video sequences were encoded at 9 different bitrates from 15 to 740 Kbps. The complexity of the components of a H.264 video decoder was measured using the VTune performance analyzer. The relationship between BA and DA for the Intra prediction component is shown in Figure 5. It shows that the Bitstream complexity metric of the average number of non-zero coefficients per frame follows the instructions per frame closely and can be used to obtain high level resource estimates.

VI. CONCLUSION

Increasing system complexity will adversely impact design productivity. Tighter integration of hardware and software will be required to attain performance gains. A system architect's design blue print will have to be significantly more precise to address both these concerns. In this paper, we document three different annotation mechanisms that can help the architect make more informed decisions. This should help avoid down-stream integration delays and unit cost run-ups due to the use of expensive, albeit unnecessary, components.

REFERENCES

[1] G. Desoli, E. Filippi, "An outlook on the evolution of mobile terminals: from monolithic to modular multi-radio, multi-application platforms", *IEEE magazine on Circuits and Systems*, vol. 6, No. 2, pp. 17-29, 2006

[2] W.C. Rhines, "Sociology of design and EDA", *IEEE transaction of design and test*, vol. 23, issue 4, pp. 304-310, April 2006.

[3] E. A. Lee, Yuhing Xiong, "System level types for component-based design, *Workshop on Embedded Software*, California, October 2001

[4] Y. Xiong and E. A. Lee, "An extensible type system for component-based design", *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Germany, April 2000.

[5] D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. De Micheli, "NoC synthesis flow for customized domain specific multiprocessor SoC", *IEEE Trans. on Parallel and Distributed Systems*, vol. 16, no. 2, pp. 113-129, February 2005.

[6] C. Yue, R. Song, R. Li, X. Zhou, "Study on the development of real-time digital simulation based on Matlab", *International Conference on Power System Technology*, Vol. 4, pp. 13-17, 2002

[7] H. Kalva, B. Furht, "Complexity estimation of the H.264 coded video bitstreams," *Computer Journal*. vol. 48, no. 5, pp. 504-513. 2005

[8] A. Jain, R. Shankar, Software decomposition on multi-core architecture", *10th IEEE Annual Workshop on High Performance Embedded Computing*, MIT, Boston, October 2006 (accepted)

[9] Dylan McGrath, "Impulse add support for Xilinx virtext-4 FPGA to CoDeveloper tool", *EE Times*, July 2005

[10] F. Wolf, "Behavioral intervals in embedded software," *Kluwer Academic Publishers*, Boston, MA, 2002

[11] F. Vahid, T. Givargis, "Embedded system design a unified HW/SW introduction", *John Wiley and Sons*, New Jersey, 2002

[12] A. Agarwal, R. Shankar, "A layered architecture for NOC design methodology", *IASTED International Conference on parallel and Distributed Computing and Systems*, pp. 659-666, November 2005

[13] J. Reinders, "VTune performance analyzer essentials: measurement and tuning techniques for software developers," *Intel Press*, 2004.