FPGA Architecture Characterization for System Level Performance Analysis

Douglas Densmore
University of California, Berkeley
densmore@eecs.berkeley.edu

Adam Donlin
Xilinx Research Labs
adam.donlin@xilinx.com

Alberto Sangiovanni-Vincentelli
University of California, Berkeley
alberto@eecs.berkeley.edu

ABSTRACT

We present a modular and scalable approach for automatically extracting ***actual*** *performance information from a set of FPGA-based architecture topologies. This information is used dynamically during simulation to support performance analysis in a System Level Design environment. The topologies capture systems representing common designs using FPGA technologies of interest. Their characterization is done only once; the results are then used during simulation of actual systems being explored by the designer. Our approach allows a rich set of FPGA architectures to be explored accurately at various abstraction levels to seek optimized solutions with minimal effort by the designer. To offer an industrial example of our results, we describe the characterization process for Xilinx CoreConnect-based platforms and the integration of this data into the* METROPOLIS *modeling environment.*

1. INTRODUCTION

The benefits of System Level Design (SLD) have been touted for quite some time: entering and verifying designs at levels of abstraction higher than RTL has the advantage of identifying clearly the goals of the design and discovering errors early in the design process. In addition, architecture design space exploration made possible by SLD allows optimizations beyond what is possible today. Finally, if a rigorous successive refinement process is followed, the path to implementation is much faster and yields superior designs. However, especially when designing high-performance circuits, designers have been skeptical about the inaccuracy of SLD models that may lead to implementations that are far from desirable. For this very reason, most high-performance designs such as complex ASICs, ASSPs, and microprocessors use an RTL model as the "golden model" for verification and analysis. While RTL models can be quite accurate, simulating a large design at this level does not allow performing extensive functional verification. For example, booting an operating system on an RTL level is impossible unless expensive emulation engines are used or prototype designs are available. For this reason, architecture exploration is in general performed only in a cursory fashion and actual architectures are selected on designer expertise (or faith!). Some designers develop "C" models that can be simulated rapidly, but the translation into RTL is done manually without any guarantee that the RTL so obtained reflects the behavior of the C model. In addition, once this RTL model is generated, it is used as the "golden model" where future modifications are made thus breaking the link between RTL and system-level models.

The reason for adopting this design flow rests, as already noted, in the accuracy of RTL level models that are considered faithful predictors of the actual performance of the implementation. An important source of inaccuracy in SLD models is the true cost of transactions are obscured because some properties of the architecture are not known or have been purposefully abstracted away to increase the simulation performance of the model. Average and worst case transaction cost *estimates* can be used in performance analysis to allow designers to measure the *relative performance* of the system components and guide further design decisions. We call the property of maintaining the true ordering among performances of alternative implementations *fidelity*. *Fidelity* requires that for all pairs of corresponding measurements m_1, m_2 in an abstract model and p_1, p_2 on the actual implementation, $m_1 < m_2$ holds if and only if $p_1 < p_2$. Since m and p can be at vastly different abstraction levels it is very difficult, if not impossible, to relate m and p directly with predictable accuracy. Accuracy of m in terms of p is a function of abstraction. We cannot of course claim that our approach will match RTL but we believe it to be better than standard Transaction Level Modeling (TLM) with the same abstraction benefits previously mentioned.

1.1 Approach

We contend that by exploiting knowledge of the components that constitute the designer's IP library, we can develop models that are both accurate *and* fast to simulate. In particular, we focus on an important class of architectures: highly-programmable platforms consisting of IPs implemented in an FPGA fabric together with powerful embedded processing elements (for example, the Xilinx Virtex II Pro Family). Today, the full flexibility and compute power of these platforms are difficult to leverage: partitioning of functionalities among embedded microprocessors and the FPGA fabric-based architecture elements is often based

3-9810801-0-6/DATE06 © 2006 EDAA

on a qualitative analysis. SLD tools must support quantitative, accurate analysis to demystify the performance impact of the multitude of design options available.

A variety of SLD use models that are directly applicable to FPGAs is discussed in [5]. The aim of this paper is to address accurate system level performance analysis of FPGA-based embedded systems. Our central proposition is that it is possible to increase the *fidelity* of performance analysis in FPGA-SLD by *pre*-characterizing a large number of synthetic subsystems mapped to one or more FPGA architectures. This characterization is done ***well in advance*** and ***independently*** of specific application modeling. A system designer can then use SLD tools to analyze the performance of their particular system with accurate models. Physical timing data is extracted from the mapped, synthetic subsystems and made available to the higher level SLD tools to seed the system level performance analysis with actual, physical timing data.

Our pre-characterization technique is built on some assumptions about the type of system being designed. First, we presume the application designer will reuse a substantial amount of IP. Second, we assume that the IPs have a standardized interface and that a structured interconnect (e.g., a standard bus) is used to interface the IPs [1]. Finally, we assume that one or more instruction set processors will be used in the system architecture. These assumptions are not only valid for the FPGAs we are targeting, but also represent a large segment of current design practices.

It is important to note the *synthetic* systems we characterize do *not* implement any given application: they only consume a prescribed amount of FPGA resources. The information we extract from the synthetic system is *representative* of a system with an equivalent floorplan, an equivalent number of bus masters and bus slaves, an equivalent bus-IP parameterization, etc. It is our hypothesis that a system designer can correlate between the properties of the current system model and the properties of synthetic architectures that have been pre-characterized. More importantly, a designer can use the pre-characterized data for "what-if" analysis to determine the performance of the system if he/she were to implement it with the properties of a given synthetic system.

An essential contribution of our approach is combining a system level design environment that supports *platform-based design* [6] and targets programmable platforms. A platform-based design environment like METROPOLIS [1] allows for the separation of communication, computation, and coordination. The coordination aspect can be isolated and used to annotate efficiently models with performance information gathered during pre-characterization. The fact that the pre-characterization data does not imply a given application is consistent with this separation and is vital to allowing various functional mappings to the architecture model.

1.2 Organization

In Section 2 we report the pre-characterization process. This involves the formalization of the data into the database structure. Section 3 bridges pre-characterization data and the METROPOLIS design environment. We provide background regarding METROPOLIS architecture modeling (3.1) and the METROPOLIS model annotation scheme to bind events to the appropriate characterization records in the database (3.2). Section 4 provides a use model and example MJPEG encoder using our methodology. Conclusions and future work in Section 5 complete the paper.

[1]An example is Xilinx's implementation of the IBM CoreConnect bus and the adoption of CoreConnect interfaces on the IPs of the Xilinx Embedded Development Kit (EDK).

2. CHARACTERIZATION PROCESS

Platform characterization is the process of gathering information regarding a specific metric or property of the platform. Obtaining this information is often a tradeoff between accuracy and effort. Our approach is unique since the architecture model described in METROPOLIS is **directly** correlated to an FPGA design that has been pushed through the Xilinx tool suite as a "synthetic" design to capture timing information. This is the **exact** information used to annotate events in simulation and tied to a transaction level model of the Xilinx hardware. The characterization process (see Figure 1) consists of three stages :

1. Create a single system description for the Xilinx tools via a Xilinx Microprocessor Hardware Specification (MHS) file. This is considered a *template* file.

2. Generate representative permutations of the architecture using this template and run them through the Xilinx tool flow. Permutations may be **incrementally** generated using heuristics such as maximum device resources, likely application domain, or others involving non-benign IPs (i.e. IPs that have strong effects on the characterization process). The entire permutation space need not be generated and in most cases only a small percentage is needed to capture realistic, practical designs.

3. Extract the desired performance information from the tool reports for database population.

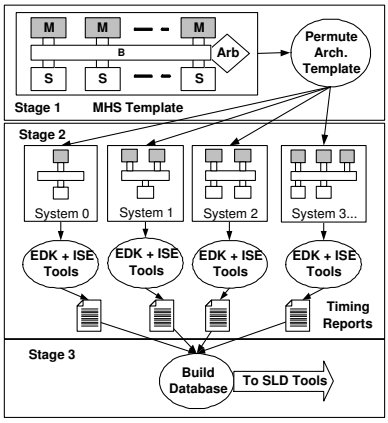


Figure 1: Characterization Flow

Information that can be gathered from this process includes (but is not limited to) various clock cycle values, longest signal path analysis, critical path information, signal dependency information, and resource utilization. The database is populated prior to simulation, perhaps by the FPGA vendor as part of the software development process. It is then instantiated as an object in METROPOLIS. This

database has to be generated *only once* since it holds information on a variety of systems. We admittedly present a static element characterization process which does not explicitly capture communication. However those static characterizations are then accumulated in complex ways during a dynamic simulation of a functional model mapped to an architectural model. This is more than a simple sum of static estimations. Section 3 presents more about how this explicitly ties into METROPOLIS.

2.1 Characterization Process Example

To exemplify our process, we pre-characterized a range of typical FPGA embedded system topologies. From a suitable MHS template file, we generated CoreConnect-based architectures with permutations of the IPs listed in Table 2.1. The table also shows the range in the number of IP instances that can be present in each system along with the potential quantities of each. In addition to varying the number of these devices, we also permuted design strategies and IP parameters. For example, we influenced the system's address decoding strategy by specifying tight (T) and loose (L) ranges in the peripheral memory map. We also permuted the arbitration policy (registered or combinatorial) for systems that contained an On-Chip Peripheral Bus (OPB). These axes of exploration were used to investigate the relationship between peripherals and the overall system timing behavior.

The columns of Table 2.1 show three permutation "classes" that were used. The implementation target was always a Xilinx XC2VP30 device. The first class (column *µBlaze*), refers to designs where µBlaze and OPB were the main processor and bus IPs respectively. The second class (column *PowerPC*) represents PowerPC and Processor Local Bus (PLB) systems. The third class (*Combo*) contain both µBlaze and PowerPC. The number of systems generated is significant (but not unnecessarily exhaustive) and demonstrates the potential of this method. Note each system permutation can be characterized **independently** and hence, each job can be farmed out to a network of workstations. For reference, the total runtime to characterize the largest *Combo* system with Xilinx XPS 6.2i on a 3GHz Xeon with 1 GB of memory was **15 minutes**. The physical design tools were run with the "high effort" option and a *User Constraint File* (UCF) that attempts to maximize the system clock frequency.

Table 1: Characterized System Configurations

Component	*µBlaze*	*PowerPC*	*Combo*
PowerPC (P)	None	1-2	1-2
µBlaze (M)	1-4	None.	1-4
BRAMS (B)	1-4	1-4	1-2 (per Bus)
UART (U)	1-2	1-2	1-2 (per Bus)
Loose/Tight Addr.	Yes	Yes	Yes
Reg/Comb. Grants	Yes	N/A	Yes
Total Systems	**128**	**32**	**256**

An observation of the characterization data shows that as resource usage increases (measured by slice [2] count) the overall system clock frequency decreases. This is evident in Figure 2, a graph of sample *Combo* systems, their size, and reported performance. Note that the graph's performance

[2]A slice contains two 4-input function generators, carry logic, arithmetic logic gates, muxes, and two storage elements.

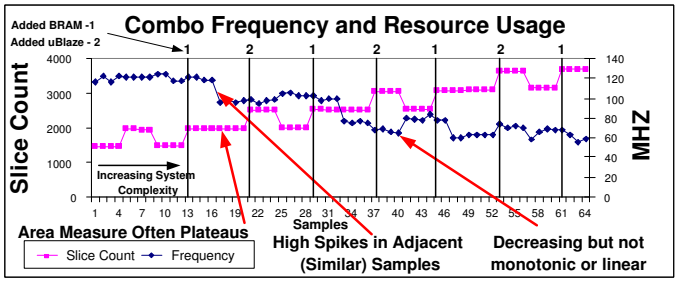


Figure 2: *Combo* **Frequency and Resource Usage**

trace is **neither linear nor monotonic**. Often area is constant while frequency changes drastically. This phenomenon prevents area based frequency estimations. The relationship between the system's area utilization and performance is complex, showing that building a static model is difficult, if at all possible, and confirming the hypothesis that *actual characterization can provide more accurate results.*

Table 2.1 highlights an interesting portion of the data collected in the *PowerPC* class. Each row is a PPC system instance: the leftmost columns show the specific IP configuration for the system and the remaining columns show area usage, max frequency, and the % change (Δ) between the previous system configuration (representing potentially a small change to the system). We contend that a difference of 10% is noteworthy and 15% is equivalent to a device speed-grade. Note that there are large MHz swings (14%+) even when there are small (<1%) changes in area. This is **not** intuitive, but seems to correspond to changes in addressing policy (T vs. L) and indicates that data gathered in pre-characterization is easy to obtain, not intuitive, and more accurate than analytical cost models.

Table 2: Non-linear Performance: PPC Systems

P	B	U	Addr.	Area	MHz	MHz Δ	Area Δ
1	2	1	T	1611	119	16.17%	39.7%
1	2	1	L	1613	102	-14.07%	0.12%
1	3	0	T	1334	117	14.56%	-17.29%
1	3	0	L	1337	95	-18.57%	0.22%
1	3	1	T	1787	120	26.04%	33.65%

Figure 3 illustrates Table 2.1 and shows area and separate performance traces for PPC systems in two addressing range styles. The graph demonstrates that whilst area is essentially equivalent, there are clear points in each performance trace with deviations greater than 10%.

2.2 Database Organization

Once the raw characterization data has been extracted, it is organized in a database (characterizer) for an architecture exploration tool to use. In this paper, we use METROPOLIS as a SLD tool. To annotate events generated in the architecture model of METROPOLIS, the characterizer contains how long an *atomic operation* (one that is primitive to the IP model) takes and how many atomic operations make up a *transaction* corresponding to a CoreConnect function or operation. The first area is the *Physical Timing* information. The second area is the *Transaction Cycles*. An atomic operation generated in METROPOLIS is a single event. A

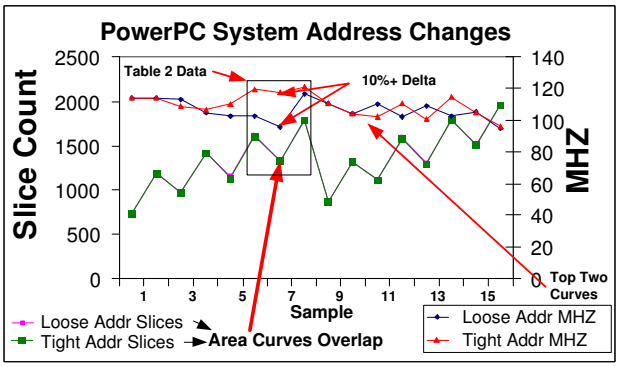


Figure 3: *PowerPC* **System Addressing Effects**

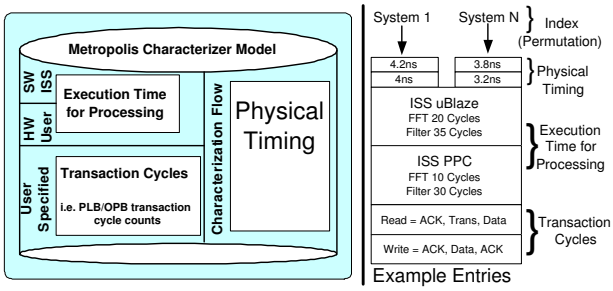


Figure 4: METROPOLIS **Characterizer**

transaction is a series of events (also called a *signal* in the Lee-Sangiovanni-Vincentelli (LSV) tagged signal model [7]). In addition to "hardware" related operations, METROPOLIS supports software tasks, an essential feature to analyze the design space of a complex FPGA that supports soft or hard microprocessor cores. Hence, the characterizer must also maintain information regarding software routines (the third and final *Execution Time for Processing* portion).

An example of METROPOLIS architecture service functions is *execute(operation)*, *read(addr, size)*, and *write(addr,size)*. When a *read* is a requested service, there is an event generated that may correspond to a bus request. This high level request, made up of various atomic operations, has to be decomposed into a set of events. How each transaction is organized resides in the characterizer and so does the physical timing. Together they can give overall execution time. For example, if it takes five cycles for a particular *read atomic operation* and each cycle is 10ns, then the total execution of this atomic operation is 50ns. A *read transaction* will be made up of several of these atomic operations. Notice that the cycle and timing information are decoupled. This allows for a variety of formulas to be used based on this information depending on the model and desired metrics.

Another scenario is the *execute(operation)* function that indicates an operation to be executed on an architecture computation resource. The mapping between the architecture and functional models determines whether this operation is to be carried out by a software routine or a hardware resource. If it is a HW resource, there is a corresponding cycle cost in the characterizer for this operation. In case this is a software routine, the characterizer has a cycle time for the routine based on extracted information from the instruction-set-simulator (ISS) for the general purpose CPU modeled. The ISS extraction is also done only once and before any simulation occurs. Figure 4 shows the organization of the Characterizer and a sample entry.

3. INTEGRATION WITH METROPOLIS

3.1 Background

We refer the reader to [1] for the details of METROPOLIS. We briefly discuss only its approach to architecture modeling and design as it is most pertinent to the topics covered in this paper. METROPOLIS architectures [2] are organized into two *netlists*. A netlist is an instantiation and connection of METROPOLIS models. The netlist separation is both for organization as well as for operational semantics. Organizationally these netlists reflect the topology of the various METROPOLIS Meta-Model (MMM) [8] designs when connected. The first such grouping is the *scheduled netlist*. This netlist contains the media (communication) and processes (computation) themselves that create the topology of the architecture and relationships between the services that can be provided by this arrangement. The second netlist is the *scheduling netlist*. This netlist provides the quantity managers (coordination) that schedule the use of the services in the scheduled netlist. When the two netlists are combined, an *architecture instance* results that serves as the complete platform for design space exploration and performance estimation through simulation. The operational semantics regarding these netlists has to do with their interaction to annotate events generated in the scheduled netlist with quantities (e.g., power and execution time) managed in the scheduling netlist. Figure 5 shows an example of the two netlists. These represent architecture services along with their schedulers while GTime manages the global record of the execution time. Requests go to the scheduling netlist, they are resolved, and the tasks are then told that they can proceed. The tasks, T1 and Tn, are the objects that actively make the requests for the services shown. It is this interaction for annotation that provides the performance estimation and the characterization work that provides the annotation data. Figure 6 shows the interaction for the annotation and the execution of an architecture model that will be discussed in more detail in Section 3.2. The scheduling netlist in Figure 6 contains the characterizer holding the annotation information.

Annotation is the process of assigning a value to the event during the *resolve()* portion as mentioned. In the case of METROPOLIS architectures, this function is performed by a quantity manager (the rhomboidal objects in Figure 5) scheduling an event to interact with a global time manager that keeps track of the "execution time" of the simulation. It is the job of the characterization mechanism to measure how long events should take and ensure that the execution time is captured accurately. While this discussion does not cover all the details involved in the process, we hope the reader has now a basic understanding of the essentials regarding how METROPOLIS architectures use annotations. Section 3.2 describes how the information is gathered for that annotation.

As indicated earlier, METROPOLIS' separation of coordination from computation and communication allows the characterizer to be reused in multiple designs and by multiple

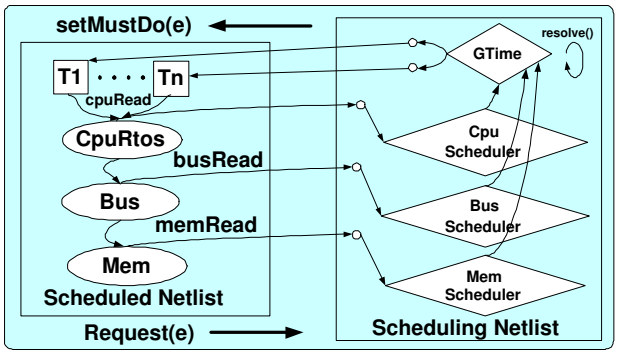


Figure 5: METROPOLIS **Architecture Netlists**

designers. How METROPOLIS implements the interface between the model and the physical timing data in the characterizer is significant. The user's model may issue transactions at a variety of abstraction levels and METROPOLIS must calculate an accurate cost from the physical timing data, no matter the abstraction of the initiating process. We will focus specifically on METROPOLIS annotation semantics for events generated in the architectural network.

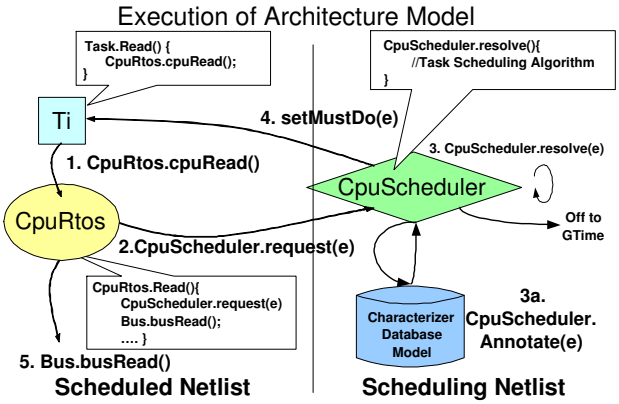


Figure 6: METROPOLIS **Architecture Execution**

3.2 Annotation Semantics

The way events acquire and use annotation information to evaluate quantities is an important contribution of this paper. We limit our analysis here to *execution time* and we ignore power, area, or other quantities that could be of interest and are indeed supported by the METROPOLIS environment. As already mentioned, there is a semantic relationship between scheduling and scheduled netlists. This involves the *request()* and *resolve()* phases of simulation. The *request()* phase is the time during which events generated in the scheduled netlist ask the scheduling netlist to make use of the services in the scheduled netlist. The *resolve()* phase is the time at which selected events that have requested services are selected not only to use those services but also when those events are assigned a value via annotation as described by the tagged signal model [7]. Figure 6 shows the *request()* and *resolve()* phase relationship. As labeled in Figure 6, the following steps apply:

1. A task acting as an active thread of control (process) calls a function on one of its ports. The port is connected to a service (media). The call asks for the right to use the service that may be contended by other processes. In the case of the diagram, it is a *cpuRead()* call. The call generates an event, *e*.

2. The service will make a request to its corresponding scheduler (a *request(e)* call). The request passes the event *e* to the scheduler. This event joins a list of *pending* events. While this event is unscheduled, the process is blocked from requesting further services. See *CpuScheduler.request(e)* in Figure 6.

3. At every step of the simulation the scheduling netlist goes through a *resolve()* phase. This is the point at which a scheduling method selects a "winning" event from a list of pending events waiting on the schedulers (scheduling algorithm can reflect what is appropriate for a particular architecture). *CpuScheduler.resolve(e)* call deals with the following operation **(3a)**: Once the event has been scheduled to run, it is annotated with the execution time it is responsible for via the characterizer. This is indexed by information related to the event and reports its execution time to GTime. This information is **highly accurate** while at the same time ***requiring no more effort*** at this point of the simulation than having estimated or arbitrary numbers attached to an event.

4. Once the event has been scheduled, it is reported back to the task that it can proceed. The media handling the service can now fulfill that request. The function *setMustDo(e)* communicates to the task that the event has been scheduled.

5. The process can occur recursively when a *read* is in a hierarchical system involving CPU, Caches, Bus, and Memory systems. For more information we refer the reader to [8].

4. USE MODEL

The previous sections explained the concept and mechanisms that underpin pre-characterization in SLD modeling. We anticipate that most architectures are characterized well in advance of a designer's use of the characterizer. However, it is conceivable that some systems may introduce new IP types or require characterization metrics not previously calculated. The designer may populate the missing data into the characterizer by explicitly invoking the characterization flow on their system. An example of the designer's use model is given as the following steps:

4.1 Develop an Architecture Model

In METROPOLIS, the user selects, from a set of architecture models pre-created to reflect Xilinx components, a system architecture model instance. If a particular component does not exist in the current set of architecture models, the user may create a new component model reflecting the service and insert it into the architecture netlist. The component's construction indicates which entries in the characterizer should be used during simulation. In [3] more information on building METROPOLIS architectures is presented.

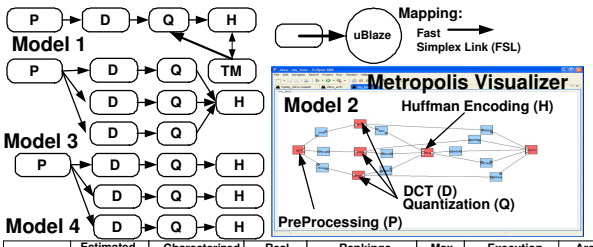
4.2 Specify Functional and Mapping Models

Proper application simulation requires a functional model be created and mapped to the architecture model above. This is the *application space* of METROPOLIS platform-based design; its detailed design is outside the scope of this paper. We mention its construction as it drives the mapping network and, in so doing, forms the application stimulus of the architecture model. An appropriate mapping network must also be constructed. See [2] for more information.

4.3 Simulate and analyze

With the characterizer and other models in place, the designer simulates the design, extracting the relevant performance data. Because the characterizer contains data for more than one architecture, the designer selects just one of the timing records before simulation. The selection is guided by informing the designer of the various topological and parameter properties in the system corresponding to each timing record. After each simulation, the designer analyzes the performance data and determines whether to proceed to a more detailed implementation. Alternatively, they can explore the design space by modifying the functional network, or varying the architectural network [4] and selecting a different timing record from the characterizer.

4.4 MJPEG Example



System	Estimated Cycles	Characterized Cycles	Real Cycles	Rankings (Real, Char, Est)	Max MHZ	Execution Time (Secs)	Area (Slices)
Model 1	145282 (52%)	228356 (25%)	304585	4, 4, 4	101.5	0.0030	4306
Model 2	103812 (33%)	145659 (6%)	154217	3, 3,2	72.3	0.0021	4927
Model 3	103935 (29%)	145414 (1.2%)	147036	2, 2,3	56.7	0.0026	7035
Model 4	103320 (28%)	144432 (<+1%)	143335	1, 1, 1	46.3	0.0031	9278

Figure 7: MJPEG Encoder Design

An example is provided in Figure 7 to demonstrate the importance of accuracy and exemplifies the *fidelity* achieved with our method. Shown are four Motion-JPEG models [9]. Each functional model was created in METROPOLIS to achieve a different level of task concurrency between the DCT, Quantization, and Huffman processes present in the application. Three of the diagrams show these topologies directly and Model 2 is shown as it looks in the Metropolis visualization software. Functional model processes and media were mapped for simulation to architectural μBlaze and Fast Simplex Link (FSL) elements respectively.

The results of a 32×32 image encoding simulation are shown in Figure 7. The first column denotes which model was examined. The second column shows the results of simulation in which estimations based on area and assembly code execution were used. The third column shows the simulation results using the characterization method described in this paper. Notice that the estimated results have an average difference of **35.5% with a max of 52%** while the characterized results have an average difference of **8.3%**. This is a significant indication of the importance of our method. In addition, the fifth column shows the rank ordering for the real, characterized, and estimated cycle results respectively. Notice that the estimated ranking does not match that of the real ordering! Even though the accuracy discrepancy is significant, it is equally (if not *more*) significant that the **overall fidelity of the estimated systems is different**. Finally the maximum frequency according to the synthesis reports, the execution time (cycles * period), and area values are shown. This confirms that while one might be tempted to evaluate only the cycle counts, it is important to understand the physical constraints of the system only available with characterized information.

5. CONCLUSION

By leveraging the properties of a platform-based design environment such as METROPOLIS and of highly-programmable rich Platforms such as the Xilinx Virtex II Pro, we created a scalable, modular, accurate, and efficient System Level Design environment. Key to this process is the creation of synthetic models of real architecture configurations, pre-characterizing these, and integrating them efficiently into the simulation environment. As designs become more complex and time-to-market pressure increases, a design flow utilizing these techniques has a clear advantage over other approaches to design space exploration via simulation.

Future work in this area includes the analysis of *fidelity* in our approach especially when more complex architecture and functional models are considered. Our fidelity analysis at this point is based on experimental evidence. We are planning to analyze this property in detail to see whether some theoretical results can be derived. In addition, various levels of modeling abstraction will be investigated analyzing *fidelity's* relationship to accuracy.

6. REFERENCES

[1] F. Balarin, H. Hsieh, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, and Y. Watanabe. Metropolis: An Integrated Environment for Electronic System Design. *IEEE Computer*, April 2003.

[2] A. Davare, D. Densmore, V. Shah, and H. Zeng. A Simple Case Study in Metropolis. Technical Memorandum UCB/ERL M04/37, Univerity of California, Berkeley, CA 94720, September 2004.

[3] D. Densmore. Metropolis Architecture Refinement Styles and Methodology. Technical Report UCB/ERL M04/36, University of California, Berkeley, September 2004.

[4] D. Densmore, S. Rekhi, and A. Sangiovanni-Vincentelli. Microarchitecture Development via Metropolis Successive Platform Refinement. In *Design Automation and Test Europe (DATE)*, February 2004.

[5] A. Donlin. Transaction Level Modeling: Flows and Use Models. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'04)*, 2004.

[6] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System Level Design: Orthogonalization of Concerns and Platform-Based Design. *IEEE Transactions on Computer-Aided Design*, Dec. 2000.

[7] A. E. Lee and A. Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *IEEE Transactions on CAD, Vol. 17, No. 12*, June 1998.

[8] T. M. P. Team. The Metropolis Meta Model Version 0.4. Technical Report UCB/ERL M04/38, University of California, Berkeley, September 2004.

[9] G. Wallace. The JPEG Still Picture Transmission Standard. *Communications of the ACM*, pages 30–34, April 1991.