FPGA Resource and Timing Estimation from Matlab Execution Traces

Per Bjuréus
Saab Avionics
Nettovägen 6
175 88 Järfälla, Sweden
+46 (8) 790 4132
perb@imit.kth.se

Mikael Millberg
Royal Institute of Technology
Electrum 229
164 40 Kista, Sweden
+46 (8) 790 4145
micke@imit.kth.se

Axel Jantsch
Royal Institute of Technology
Electrum 229
164 40 Kista, Sweden
+46 (8) 790 4124
axel@imit.kth.se

ABSTRACT

We present a simulation-based technique to estimate area and latency of an FPGA implementation of a Matlab specification. During simulation of the Matlab model, a trace is generated that can be used for multiple estimations. For estimation the user provides some design constraints such as the rate and bit width of data streams. In our experience the runtime of the estimator is approximately only 1/10 of the simulation time, which is typically fast enough to generate dozens of estimates within a few hours and to build cost-performance trade-off curves for a particular algorithm and input data. In addition, the estimator reports on the scheduling and resource binding used for estimation. This information can be utilized not only to assess the estimation quality, but also as first starting point for the final implementation.

Keywords

Design exploration, Matlab, FPGA, Estimation

1. INTRODUCTION

System-level estimation of design properties such as performance, area and power is hard and can be performed with sufficient accuracy for specific design flows and target architectures. In the sequel we propose an estimation technique for Matlab descriptions to be implemented on an FPGA. The entire workflow is presented as shown in Figure 1.

The execution trace, which is derived from a simulation of the Matlab program, contains information about the number of times all Matlab operations and functions are executed for a particular set of input vectors. From this trace, an acyclic data flow graph (DFG) is derived. It is acyclic because all loops are unfolded in the execution trace. The operations of the DFG are scheduled and bound to FPGA resources by way of a greedy scheduling and binding algorithm. The FPGA performance model provides timing information, which is detailed enough to estimate pipelining and resource sharing effects.

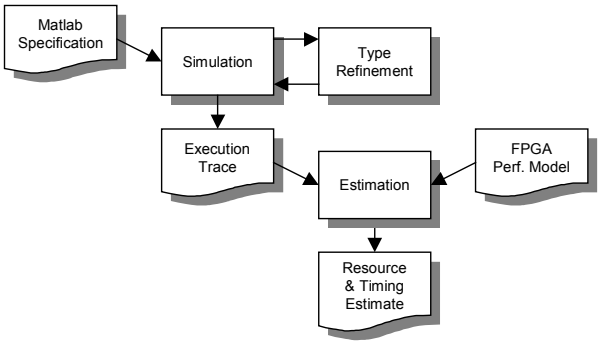

Figure 1. Design space exploration workflow

The main target application area is regular data flow systems such as signal or image processing applications. In that domain Matlab is a very popular language to explore and develop algorithms. During algorithm development it is extremely beneficial to approximately estimate performance and cost of a possible implementation. Having this feedback available quickly may allow tuning the algorithm for optimal performance-cost trade-off.

In earlier work we have developed a system-level modeling environment that integrates the languages Matlab and SDL to allow to evaluate and simulate the complete system including the data flow and the control dominated parts. This allows the designer to get the system functionality right before implementing it in lower level C and VHDL code. In order to also optimize for performance and cost at the system level, fast feedback through estimation is needed. However, a useful estimation must take the target architecture and technology into account. In [5] we have reported a stochastic approach to estimate the performance of Matlab programs in a SW implementation. Here we complement this work with an estimator for an FPGA implementation. Even though a complete estimation tool suite should also contain estimators for other target technologies and architectures, we have already come a long way by developing a simulation trace based estimation framework, which allows the integration of new estimators with moderate effort.

2. RELATED WORK

This paper spans a large number of disciplines in electronic system design from high-level specification to scheduling and resource binding. However, our focus is on the high-level modeling, type refinement, and trace generation techniques. We have chosen a simple approach for low-level scheduling and resource binding

to justify the high-level concepts. A data flow graph is generated from the high-level model. This is a commonly used representation, enabling alternative, more advanced and accurate low-level methods to be integrated with our framework.

High-level design exploration is recognized as an important activity in the design flow and other frameworks have been proposed. The Polis framework [1], developed at UC Berkeley targets control dominated embedded systems. Polis uses Esterel as its front-end language, and employs a Codesign Finite State Machine (CFSM) model of computation to model communication and concurrency. In Polis, the user performs design exploration by mapping CFSMs onto different architectural components. Mapping onto SW yields a sequential execution of the state machine whereas mapping onto hardware yields a state machine that is always executed in one clock cycle. The Polis framework controls synthesis and the estimation relies on knowledge about the code generation. Our work differs in several ways. We do not have knowledge about the code generation, which makes estimation significantly harder. The algorithms we target are data flow dominated and cannot in most cases be executed in a single clock cycle in hardware.

Matlab code generation and synthesis is targeted by the Match (MATlab Compiler for Heterogeneous computing systems) project at the Northwestern University [4][8]. The Match project allows VHDL generation from Matlab code for FPGA implementation. This requires a static analysis of the code and gradual transformations to exploit the parallelism. Since Matlab is dynamically typed, a static analysis requires that the size and shape of the Matlab matrices be bounded explicitly. In our approach, we only try to *estimate* an implementation. Therefore, to relieve the designer from the task of explicitly defining matrix bounds, the matrix size and shape is recorded during execution. This speeds up design exploration when the designer wants to explore different matrix bounds.

Bammi et al. [3] use a virtual instruction set with an associated performance model for each virtual instruction to estimate performance of a C program. Similar to us they also use a simulation run and perform part of the compilation/synthesis to increase estimation accuracy. However, they have C programs as input and estimate SW implementations while we work with Matlab to estimate FPGA implementation.

A different approach, also for the performance estimation of C programs, is taken by F. Balarin [2] who asks the user to provide abstractions for the components to be estimated. These abstractions are executed during a simulation run and can be used to collect information about the run-time of the considered component in sophisticated ways. The user has some control over the accuracy of the estimation by choosing the abstraction.

Yet another approach, taken by Brandolese et al. [7], is to derive a probability distribution for the performance. The objective is again to estimate the performance of a C program with the help of a simulation run during which information about execution count of individual statements and operations is collected. Together with a probability distribution of the execution delay of operations, a probability distribution for the execution time of the entire program is derived.

We have previously used a similar, execution-trace based, approach to estimate the execution time of software processes on microprocessors [5]. The work in this paper complements our previous efforts and is a further step to a complete estimation tool suite for HW/SW codesign.

The novelty with our work lies in the extraction and processing of execution data from a Matlab simulation using trace-generating functions. We have developed a method, tools and libraries that support the techniques presented. The usage of Matlab is not restricted per se, but rather by the extent of the trace-generating library, which can easily be expanded to facilitate any Matlab operators and functions. The results from the scheduling and binding algorithms are preliminary but show the validity of the concept.

3. PREMISES

Starting from a Matlab function specification we want to trade off design parameters such as throughput, latency and resource utilization (area) for an FPGA implementation of the function. We allow the Matlab specification to contain control statements such as if-then-else and unbounded loops. The primary target applications are resource dominated dataflow processes with a constant execution rate.

Consider the following Matlab code:

```
r = a*b;
```

Estimating the area and latency of this code fragment is virtually impossible without knowing the size, shape, and type of the variables. The variables, `a` and `b`, can be scalars, vectors, or matrices, and the number of multiplications is not known until run-time unless the size of `a` and `b` is bounded beforehand. Matrix bounding is not enforced in Matlab and is not common practice.

Furthermore, the default Matlab data type is double precision floating-point. It is unlikely that an FPGA implementation will use this data type because it would require floating-point arithmetic, which is very expensive in terms of area and latency. A more realistic approach is to use fixed-point arithmetic. However, Matlab does not natively support fixed-point data types and therefore the designer must explicitly define the fixed-point format (bit width and fraction), at least for the input data stream.

3.1 Functional Specification

The functional specification is written in Matlab. The Matlab code can be executed to verify the correct behavior of the specification. To perform trace generation, the type of the variables must be changed to a trace generating class called “agent” by adding the “agent” keyword when a new variable is declared.

```
a = 4;
```

The above variable declaration is changed into:

```
a = agent(4);
```

The agent class has the ability to keep track of the operations and dependencies throughout execution by operator overloading. The agent class also has the ability to record and use external type specifications, see section 4.

Definition 1: Dataflow Process. A dataflow process *P* is a functional unit with a set of input ports and a set of output ports. When a dataflow process executes, a number of tokens are consumed from the input ports, and a number of tokens are produced at the output ports.

Definition 2: Rate Constraint. A rate constraint C_r defines how often (exactly) data arrives at a process port.

In the Matlab specification, constraints can be added according to the below example. The constraints will be recorded in the execution trace and used by the estimation tool.

```
addconstraint('DD1Re', '16MHz');
addconstraint('DD1Im', '16MHz');
```

The process execution rate can be derived from the rate constraints of the data streams that are connected to the process ports.

3.2 Execution Trace

To be able to estimate control statements correctly, a simulation-based approach is used. During simulation, an execution trace is recorded that accurately captures all computation and communication that occur in the system. Here, we are only interested in computation, which appears in the execution trace as process invocations. A process invocation represents the activation and execution of a dataflow process in the specification

Definition 3: Process Invocation. A process invocation ξ is a sequence of operation traces $\xi = \{\omega_1, \omega_2, \ldots, \omega_k\}, k \geq 0$.

The operation traces in the process invocation captures the execution of functions and operations inside the process.

Definition 4: Operation Trace. An operation trace ω is a 9-tuple $\omega = (i, op, P, loc, n, D, \Omega, t, r)$ where $i \in \mathbf{N}$ is an instance count, op is an operation identifier, $P=\{p_1, p_2, \ldots, p_m\}, m \geq 0$ is a list of parameters, loc is a location identifier, n is an operation node identifier, $D=\{n_1, n_2, \ldots, n_j\}, j \geq 0$ is a list of dependency node identifiers, $\Omega=\{\omega_1, \omega_2, \ldots, \omega_k\}, k \geq 0$ is a list of child operation traces, t is a type identifier, and $r \in \mathbf{N}$ is a repeat count.

The parameter list P contains a set of structural parameters. For example the multiplication operation shown below has three parameters, which specify the size of the matrices being multiplied.

```
2 # MUL (2,3,2) [test: line 5] *n45* <n43, n44> {
} [fixpt4.3] # 1
```

The instance count i specifies how many independent instances of the operation that are executed. The repeat count r on the other hand, specifies how many times the operation is executed sequentially. The instance and repeat counts result from vector processing in Matlab and is just a compressed way to write a symmetric dependency graph.

Figure 2 shows how the instance and repeat counts are interpreted. The same operation is laid out in an $r \times i$ matrix, where operation instances on the same row are independent of each other, whereas all operations in a single row are dependent of all operations in the previous row.

The operation node identifier n and the dependency node list D are used to derive dependencies. An operation A that depends on another operation B will have B's node identifier n_B in its dependency list D_A.

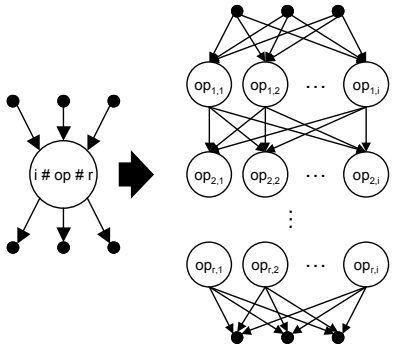

Figure 2. Instance and repeat count interpretation

The operation trace is hierarchically decomposed through the child operation trace list Ω. Note that this relation is very different from the operation dependency. An operation trace with a non-empty child operation list can be fully substituted by its children during estimation. For example, the Matlab `filter` operation is a complex operation that can be decomposed into multiply-accumulate (MAC) operations. The MAC operation in turn can be decomposed into multiplication and addition. If the performance model can estimate an entire filter, the top-level operation is used, and the child operations will be disregarded. Otherwise, the MAC child operations will be used, and, if the MAC operation cannot be estimated either, it will be further broken down into multiplication and addition operations.

The type identifier t specifies the data type of the operation and is used to determine the operation bit width.

3.3 FPGA Performance Model

The FPGA is represented by a performance model, which is used to retrieve the area and latency information about different operations in the execution trace. Each operation is mapped to a particular FPGA resource such as e.g. a comparator. The resource is modeled as a function that maps an operation to an estimated area, latency, and service rate.

Definition 5: Performance Model. A performance model M is a function that maps an operation identifier op, argument bit widths $<width>$ and clock speed clk, to area, latency and service rate: $M(op, <width>, clk) = (S, lat_{\#}, rate^{-1})$, where $S \in \mathbf{R}$ is the resource area (size), $lat_{\#} \in \mathbf{N}$ is the latency expressed in number of clock cycles, and $rate^{-1} \in \mathbf{N}$, $1 \leq rate^{-1} \leq lat_{\#}$ is the inverse of the service rate expressed in clock cycles per data.

The service rate is used to exploit pipelining. When the rate inverse equals the resource latency, i.e. $rate^{-1} = lat_{\#}$, the resource can service a new input data at the same rate as its total latency and no pipelining is possible. When the service rate inverse is less than the resource latency ($rate^{-1} < lat_{\#}$), pipelining is possible.

4. TYPE REFINEMENT

Since the hardware resources used in an implementation are very sensitive to the bit widths of the input and output arguments, the bit widths need to be specified before estimation.

Starting from an executable Matlab specification the user must perform type refinement to be able to generate an execution trace for hardware estimation. The goal has been to allow maximum reuse of the original code during the refinement task.

The type refinement technique that we propose is similar to [10], which requires two execution passes. In the first pass, the data types of all variables in the specification are recorded. This procedure must be repeated every time a change is made in the functional specification. During recording, operator output data types are automatically derived from the input data types. When the data types have been recorded, they can be modified externally, allowing the designer to control the bit widths and accuracy. In the second execution pass, the specification uses the externally defined data types, checking and issuing warnings for overflows. In this way, data type exploration can be performed until satisfactory accuracy is achieved. The recorded variables are displayed and modified in a type configuration window; see Figure 3.

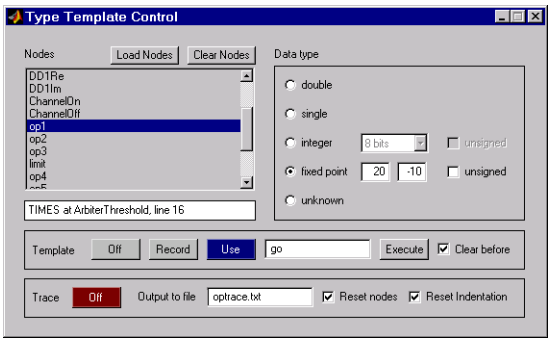

Figure 3. Type configuration window

The user has an option to set the data type explicitly in the Matlab code, which will then be used in the recording pass.

```
a = agent(4, 'a', 'fixpt10.-5');
```

The code fragment above shows how this is done. The variable 'a' is declared as a fixed-point variable with bit width 10 and the binary point located 5 bits to the left of the least significant bit. Note that the type definition found in the Matlab code will be overridden by the external type specification during the second execution pass.

5. ESTIMATION

When the Matlab specification has been type refined, and an execution trace has been recorded, the estimator takes over. During estimation the following steps are performed:

- Build DFG
- Annotate DFG nodes with resource data
- Create area/latency grid from constraints
- Schedule and bind operators and resources

In the first step, the execution trace is read and a data flow graph (DFG) is built. The DFG is a bipolar directed acyclic graph where each node corresponds to an operation.

In the second step, the DFG is annotated with resource data, e.g. latency as shown in Figure 4. The resource data is fetched from the FPGA performance model according to section 3.3. This step requires that the clock speed f of the FPGA and the execution rate r be set in advance. If rate constraints are available, the execution rate is derived from those.

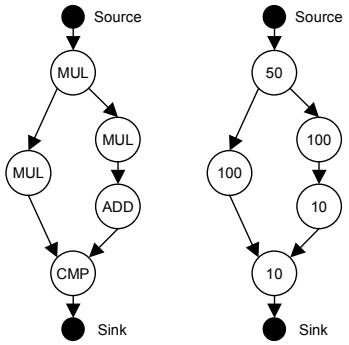

Figure 4. DFG node latency annotation

The annotated DFG is then traversed from top to bottom and back to compute the earliest execution time (EET) and the remaining execution time (RET) for each node. Figure 5 shows this step applied to the example in Figure 4. The EET of the sink node corresponds to the minimum expected latency of the DFG.

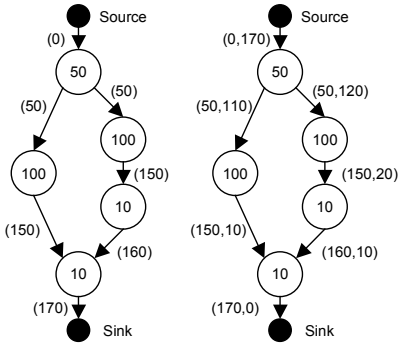

Figure 5. EET and RET annotation

During estimation, there are four design parameters to trade off: FPGA clock speed f, execution rate r, latency l, and area A. The latency defines the time from data input to data output. The area parameter specifies how much area the resources occupy in the chosen device. The clock speed determines the device clock frequency and is used to derive the clock period $T=1/f$.

The estimator uses the annotated DFG together with the area and latency to determine a schedule and resource binding of an implementation of the Matlab function. To this end we have used an area/latency grid.

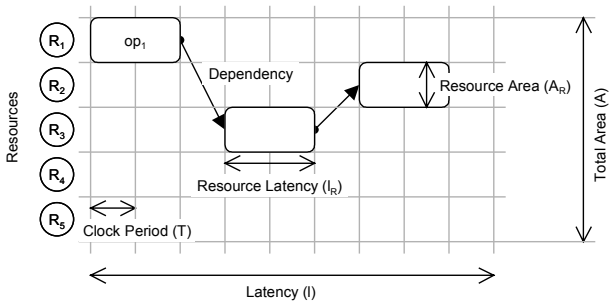

Figure 6. Area/latency grid

Figure 6 shows a schematic picture of the area/latency grid. Each resource is assigned to a row, where the height of a row corresponds to the area of the resource. The total grid height is constrained by the area A. Each column corresponds to a clock cycle

with period time T. Resources are instantiated in the grid when they perform an operation and are labeled with the operation they perform. A resource instance spans one or more columns in the grid according to its resource latency $lat_{\#}$. A resource can be reused at an interval that is equal to its inverse service rate: $rate^{-1}$. This allows us to model pipelined resources that can be reused at an interval that is shorter than its latency.

The scheduling and resource-binding algorithm that we have used is a greedy algorithm that works as follows:

- Initialize the area/latency grid G by searching the DFG for all different resources. At least one resource of each kind must be available in the grid.
- Initialize the DFG by setting the modified execution time (MET) to EET, and marking each node as "not visited".
- Initialize a set E of enabled nodes. The set will initially contain all the successors of the source node
- While the set of enabled nodes is not empty, $E \neq \varnothing$, do the following:
  - Select the next node $n \in E$ to schedule by choosing the node with (1) smallest MET and (2) greatest RET.
  - Find an idle resource R for the next operation. If no resource is currently idle, create a new resource. If a new resource cannot be created due to area constraints, select the resource that will become idle first.
  - Schedule the node n on the selected resource R by adding it to the grid. Update the MET of the node and all its successors down to the sink node accordingly.
  - Mark the scheduled node n as "visited" and update the enabled node list E by removing n and adding all successors that have all predecessors marked visited.
- When the active node list is empty, all operations have been scheduled. Calculate the area and latency from the grid.

The scheduling and resource binding optimization problem is well known in the high-level synthesis field, and a much more extensive treatment of the subject is available e.g. in [9].

6. EXPERIMENTS AND RESULTS

To validate the proposed estimation technique, we have applied it to the `ArbiterThreshold` process fetched from a model of a Digital Receiver [5]. The selected process was simulated, refined, and used to generate an execution trace. A performance model of an FPGA was developed and used for estimation. Design exploration was then performed to trade off different design parameters.

6.1 Type Refinement and Trace Generation

The `ArbiterThreshold` process was first simulated in Matlab to verify the function on a behavioral level. After functional verification, the data types of the process inputs were changed to the 'agent' class. The process was then simulated again, recording the data types in the process. After recording the variables, the type configuration window was used to display and alter the data types. At this point, all data types were refined into fixed-point with appropriate bit widths. To verify that the correct behavior was preserved after type refinement, the process was again simulated using the refined types. After verification, the process was simulated one last time in "trace"-mode, storing all operations in a file.

Table 1. `ArbiterThreshold` process execution time

Simulation Mode	Simulation Time
Normal	5.8680
Record data types	57.7730
Use data types	138.9400
Trace	255.0670

Table 1 shows the simulation time of the `ArbiterThreshold` process in different simulation modes.

6.2 FPGA Performance Modeling

To model the performance of the FPGA, three operations (addition, multiplication, and comparison) that are central in the `ArbiterThreshold` process were characterized. The characterization process was automated with a script, allowing a large number of possible implementations to be investigated.

Table 2. FPGA Performance Model Operations

Op	#	BW	Delay	Pipe	Size
ADD	12	4x4-32x32	8.29 - 103.47	0	6-62
CMP	12	4x4-32x32	10.66 - 70.49	0	3-50
MUL	160	4x4-32x32	8.94 - 150.13	0-5	35-2128

Table 2 shows a summary of the FPGA performance model used. The first column displays the operation (Op) and the second column the number of possible operations available in the model (#). The third column shows the range of bit-widths available (BW). Column four shows the range of delays acquired (Delay), where the delay corresponds to the service rate inverse: $rate^{-1}$. Different pipeline lengths were derived for the MUL operation and range from zero to five (Pipe). Finally, the last column shows the range of sizes (Size), expressed in number of look-up tables (LUTs).

6.3 Design Exploration

We have developed a tool that operates on the execution trace and allows the designer to alter design parameters and run the estimation. The tool used for estimation is shown in Figure 7.

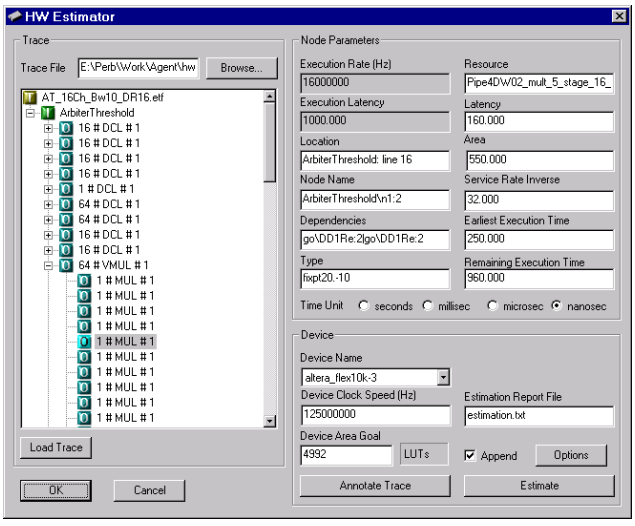

Figure 7. Estimation Tool

The tool shows the hierarchical execution trace to the left and the node parameters after annotation to the right.

The design exploration was performed considering several design alternatives. The interesting design parameters are:

- Number of input channels (8 or 16)
- Input stream bit width (8 or 16)
- Input stream data rate (1 MHz)
- Device Clock Speed (8, 16, and 32MHz)
- Execution Latency
- Device Area

Altering the first two parameters requires that the Matlab specification be changed and that new execution traces be generated for each set of parameters. The last four parameters can be explored using the same execution trace with a fixed number of channels and input stream bit width.

We explored the design parameters according to the above specification and the result is shown in Figure 8.

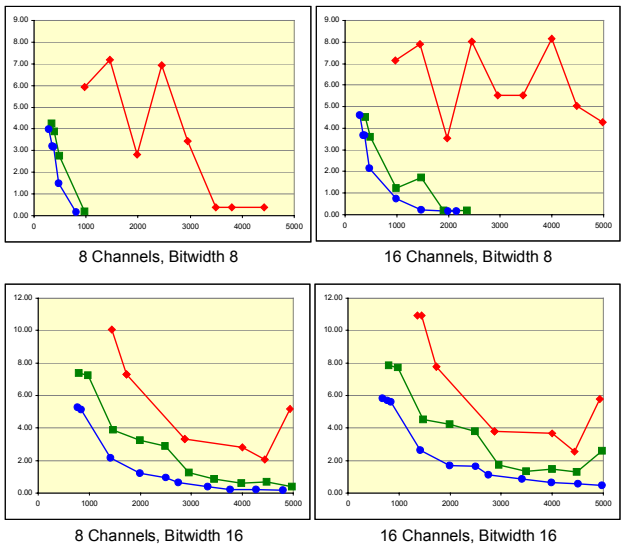

Figure 8. Design exploration

The input bit width was 8 and 16 bits, and the number of input channels 8 and 16. Thus, in total four execution traces were used. In each graph, the latency (y-axis) and area (x-axis) is shown for device speeds 8MHz (diamonds), 16MHz (squares), and 32MHz (circles). In general, the latency increases with decreasing area and device speed. There are however discrepancies that can be explained by the greedy algorithm, which sometimes lead to local optima. The entire exploration, containing 98 data points in the design space, took less than an hour to perform.

Since the estimation is based on real component libraries, and the estimator generates a valid schedule, the accuracy of the estimate has been shown to be within ±10% if that particular solution is implemented. However, the estimate should be viewed as an upper bound, because clever engineers and modern high-level synthesis tools are capable of more sophisticated optimization.

7. DISCUSSION

The high-level estimation technique presented allows a user to efficiently trade off different design parameters for an FPGA implementation of a Matlab specification. The estimator only presents *one possible* implementation of the function. However, it also provides a resource and schedule report that shows the designer *which* implementation that was estimated. This gives the engineer a fair chance to actually implement the function in compliance with the design goals. Other scheduling and resource binding techniques can be integrated with the same framework since the internal DFG representation is commonly used in high-level synthesis. The framework is therefore flexible, and we have implemented a simple low-level technique to demonstrate how the high-level techniques can be used.

8. REFERENCES

[1] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, B. Tabbara, *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*, Kluwer Academic Press, 1997

[2] F. Balarin, "STARS of MPEG decoder: a case study in worst-case analysis of discrete event systems", *Proceedings of the International Workshop on HW/SW Codesign*, pp. 104-108, April 2001.

[3] J.R.Bammi, E.Harcourt, W.Kruijtzer, L. Lavagno, M.T.Lazarescu, "Software Performance Estimation Strategies in a System-Level Design Tool", *Proceedings of the International Workshop on HW/SW Codesign*, pp. 82-86, May 2000.

[4] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, A. Periyacheri, M. Walkden, D. Zaretsky, "A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems", *Proceedings 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, USA, 17-19 April 2000

[5] P. Bjureus, F. Hoffman, "Modeling a Digital Receiver with Mascot", *Proceedings of the Designer's Forum at Design, Automation & Test in Europe Conference (DATE)*, Munich, Germany, 2001

[6] P. Bjureus, A. Jantsch, "Performance analysis with confidence intervals for embedded software processes", *Proceedings of the International Symposium on System Synthesis (ISSS)*, Montreal, Que., Canada, 30 Sept.-3 Oct. 2001

[7] C. Brandolese, W. Fornaciari, F. Salice, D. Sciuto, "Source-Level Execution Time Estimation of C Programs", *Proceedings of the International Workshop on HW/SW Codesign*, pp. 98-103, April 2001.

[8] M. Haldar, A. Nayak, N. Shenoy, A. Choudhary, P. Banerjee, "FPGA hardware synthesis from MATLAB", *Proceedings of 14th International Conference on VLSI Design*, Bangalore, India, 3-7 Jan. 2001

[9] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994

[10] H. Olson, A. Jantsch, H. Tenhunen, "Floating- to Fixed-Point Refinement in Matlab with an Object-Oriented Library", *Proceedings '99. 17th NORCHIP Conference*, Oslo, Norway, 8-9 November 1999