System-Level Design Guidance Using Structural Algorithmic Properties

Lisa Guerra
EECS Department
University of California, Berkeley, CA 94720

Miodrag Potkonjak
C&CCS Department
University of California, Los Angeles, CA 90024

Jan Rabaey
EECS Department
University of California, Berkeley, CA 94720

System-Level Design Guidance Using Structural Algorithmic Properties

ABSTRACT

As the emphasis in a design process of DSP application specific systems is being shifted toward high levels of abstractions a need for systematic design space exploration is gaining importance. This paper introduces a system-level design approach to design space exploration which provides quantitative information used to aid in making system-level design decisions such as algorithmic or architectural selection. The approach is based on the idea of characterizing an algorithm by a set of size and structural properties. The structural properties include uniformity, temporality, spatial locality, regularity, cyclic properties and control flow properties. The properties provide insight and quantitative information on the suitability of the algorithm for implementation and a link between algorithms and architectures.

For several properties, the underlying intuition behind them, quantitative definitions, and illustrations of their utility are described. Their utility as estimators of implementation performance are shown using both simple, yet realistic examples and extensive empirical studies, primarily through a case study on area estimation of custom ASIC implementations.

For three properties, an in depth study is presented. The *temporal density* property, a measure of how the lifetimes of the variables are distributed in time, is shown to be correlated to the required number of registers. The *uniformity* property, a measure of how well resource accesses are distributed over the duration of the computation, can be used to predict the number of required execution units. The *regularity* property, a measure of the amount of common patterns that exist in the computation, is shown to be related to both the amount of interconnect and controller size.

1.0 Introduction

Common system-level design decisions include the choice of algorithms for a given functional specification, the partitioning and optimization of those algorithms, and the selection of architectures (Figure 1). Case studies indicate that these choices can often improve all design metrics by an order of magnitude or more [Pot93, Lid94]. Figure 1 illustrates the task of implementing a design which optimizes a set of performance goals, while meeting the functional and performance specifications. Consider, for example, the goal to optimize power in the design of an adaptive equalizer while meeting functional specifications for convergence rate and maximum estimation error, as well as the performance specification of minimum throughput. The designer has the choice of one of several algorithms for tap updates, as well as several architectural platforms. In comparing two algorithms, the more aggressive tap-update algorithm may require more parallel execution units in a semi-custom ASIC implementation, but only require a minimal amount of additional code when implemented on a DSP processor. Further exploration of the design space could show that optimizations result in the possibility of mapping onto a less expensive processor, or using significantly less interconnect area in an ASIC implementation.

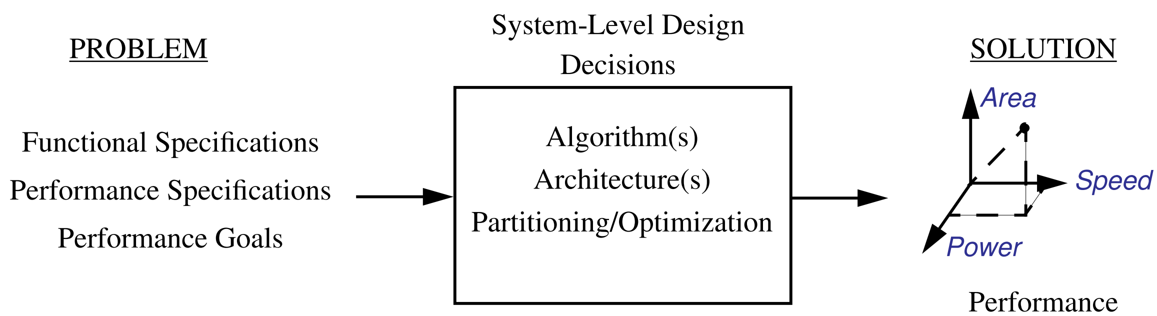


Figure 1: Common System-Level Design Choices

When doing system-level design, it is important to be able to analyze the possibilities and trade-offs of high level decisions, before investing a lot of effort in exploration at lower levels (e.g. optimizing a design on a fixed architecture for a particular algorithm). Because of the size and complexity of the design space, however, the designer often does not adequately explore many of the design possibilities. The choice of an architecture is often implicit, based on factors such as availability of the platform, supporting software tools, familiarity, or subjective opinion. Furthermore, the selection of an algorithm is also often subjective, or based on quantitative but incomplete measures such as the number of multiplications, the asymptotic computational complexity, or the performance on abstract computational machines (e.g. Parallel Random Access Machines (PRAM) [Kar90] or RAM).

In order to more fully explore the design space at the system-level, prediction/estimation will play an increasingly important and necessary role. It is unreasonable to expect a designer to have access to multiple platforms, tools to use them, and expertise in knowing how to use them all. Even if this were not the case, it would be difficult to devise a systematic and efficient scheme for exploring and analyzing all possible trade-offs. The effort necessary to perform a comprehensive search using contemporary tools is unreasonably large. It is thus necessary to provide a means by which comprehensive design space exploration can be done efficiently and in a qualified manner. This paper proposes a solution in which “design guidance” is provided by a set of structural properties of algorithms which are correlated to various aspects of performance on different architectural platforms. The goal is to use properties of an algorithm such as regularity and temporal density, properties previously used qualitatively, at best, in design decisions. Such information is computed efficiently, making it feasible for the designer to evaluate a larger number of solutions in a time efficient manner, and thus enabling a system-level design process based on informed and quantitative decisions.

The paper is organized as follows. First related research is described, then an overview of the proposed property-based design guidance environment is presented. Following this, the usefulness of this framework is illustrated in a specific case study on area estimation for custom ASIC implementations. This section demonstrates the utility of the properties using examples and by empirically correlating them to the components of area. Finally, conclusions and directions for future efforts are given.

2.0 Related Research

Related work can be traced along several lines of research: (1) prediction (estimation) techniques at lower levels of abstraction; (2) recent hardware-software co-design efforts; (3) algorithm design and selection research; (and) (4) properties for design guidance in the computer systems and VLSI Signal Processing domains Additional survey of efforts related to specific structural properties are presented in the detailed property descriptions.

Until now, estimations CAD research has been concentrated at lower levels of abstraction. The high-level synthesis literature contains a variety of techniques for computing expected or lower bounds on hardware resources [Kur89, Rab91b, Jai92, Sha93, Cha94]. These estimations are based or take advantage of predefined hardware models. This work, on the other hand, addresses estimation at a higher level of abstraction, in which it is necessary to understand relationships between algorithms and a *variety* of hardware platforms.

Previous work in system-level design has concentrated mainly on the hardware-software trade-off. These research efforts span a variety of issues, including partitioning, co-specification, co-simulations, and hardware-software interface design [Sri91, Cho92, Ern93, Gup93, Kal93, Woo94]. It is assumed that the algorithm specification is fixed. Also, the quality of the results are usually evaluated at the end of the design cycle by measuring the hardware area or profiling the software sections, whereas the proposed approach quantifies the expected quality of the results before traversal of the design cycle.

The previous works of most direct influence deal with analyzing properties on a set of applications to provide design guidance for general purpose architectures. Several properties have been proposed in the computer science literature, and in particular the computer architecture and parallel computing research. For example, Amdhal’s law [Amd64] indicates the amount of speedup attainable on an algorithm by using a parallel architecture. This property provides an argument against massive parallelism for a number of application domains. Several size and time complexity measures of parallel algorithms were also presented by [Bre74, OhE85]. The scalability measure [Hil90] tracks the behavior of a program's run time as its size increases. Locality and the 90/10 rule of thumb, convey the qualitative property of typical programs that 90% of execution time is typically spent on 10% of the instructions [Knu71]. This property has also greatly influenced architecture design, in particular, the design of memory hierarchies [Pat90].

In the VLSI DSP domain, the qualitative observation that many signal processing algorithms are regular, has motivated the development of systolic [Kun82] and wavefront arrays [Kun84]. Another property is the classification of recursive algorithms made by S. Y. Kung [Kun88], as having either inherently local or global communications. A recursive algorithm is global if it always implies global interconnect, and otherwise is considered local. Since local interconnect is strongly preferred in VLSI realizations, this information is a good indicator of whether or not a particular algorithm is well suited for implementation.

ADD L. JAMISON's EFFORTS REVIEW

There are several crucial differences between the related research and the work presented here. Whereas previously, a single property was analyzed and studied in isolation, this is, to the best of our knowledge, the first comprehensive and organized treatment of structural properties. Unlike the mentioned works, many of which characterize a *set* of algorithms (and have thus been used to influence and guide the design of general purpose machines), this work is concerned with characterizing a *single* algorithm (and thus its suitability for implementation on one of many available architectures). Previous works have dealt primarily with homogeneous resources (e.g. ALUs), while here a heterogeneous mix of resources is additionally supported. For the first time, the approach is completely based on quantitative definitions of all the properties and presents a novel unified methodology for system-level design guidance. Finally, statistical validation and detailed analysis of the effectiveness of system-level guidance is conducted on a large set of real-life examples. An overview of the approach is presented next.

3.0 Design Guidance Environment

3.1 Goals

The goal of the design exploration environment is to interactively aid the designer in making crucial high-level decisions such as selection of algorithms, selection of architectures, partitioning, and selection of transformations. The way in which this is achieved is by providing quantitative information that can be used to evaluate the impact of a potential decision (Figure 2). This can enhance and support the designer's intuition and quick "back-of-the-envelope" calculations.

A primary task addressed by the environment is that of providing *early estimation* of performance metrics such as cost (e.g. area, number of processors), speed, or power for a given algorithm, architecture, and set of performance constraints. For measures like area and power, this is done hierarchically, by first analyzing execution unit, memory, interconnect, and control sub-components, then deriving overall performance measures. In addition to early estimation of performance metrics, this information can be used to develop a deeper understanding of algorithm trade-offs. For example, an interesting observation could be that relaxing the throughput constraint changes the dominant power consumer from interconnect to registers. Studies on the architecture sub-components' contributions to performance measures can aid in algorithm understanding, and later algorithm design.

Another key task of the framework is to provide feedback to guide transformation selections that can improve the algorithm and proposed partitioning of the algorithm onto multiple datapaths or processors. Other types of feedback include suggestions for architecture instruction sets that are better suited for the algorithm, and identification of architecture sub-components that are in high demand but short supply (or vice versa).

An important side benefit of the approach is the development of measures for use in cost functions for synthesis tools (e.g. transformations, scheduling, partitioning). Most current tools use cost functions which supposedly model performance of the final design, but do not statistically validate their measures.

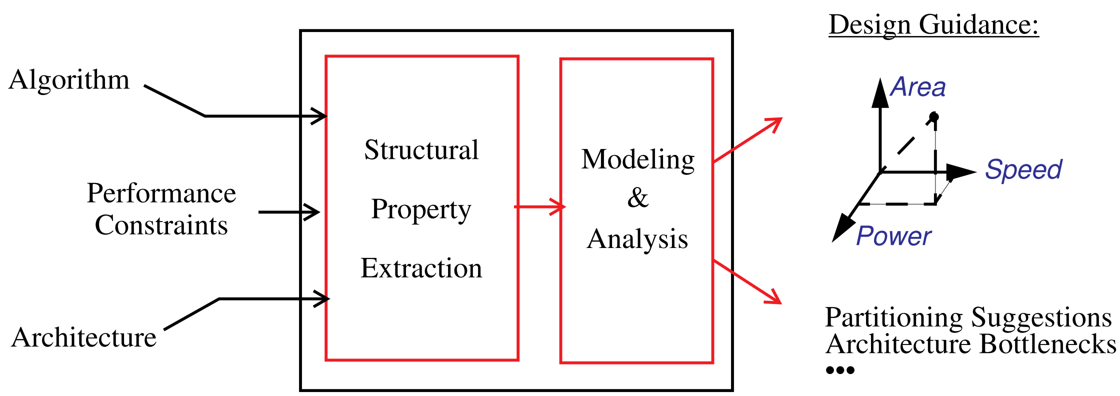
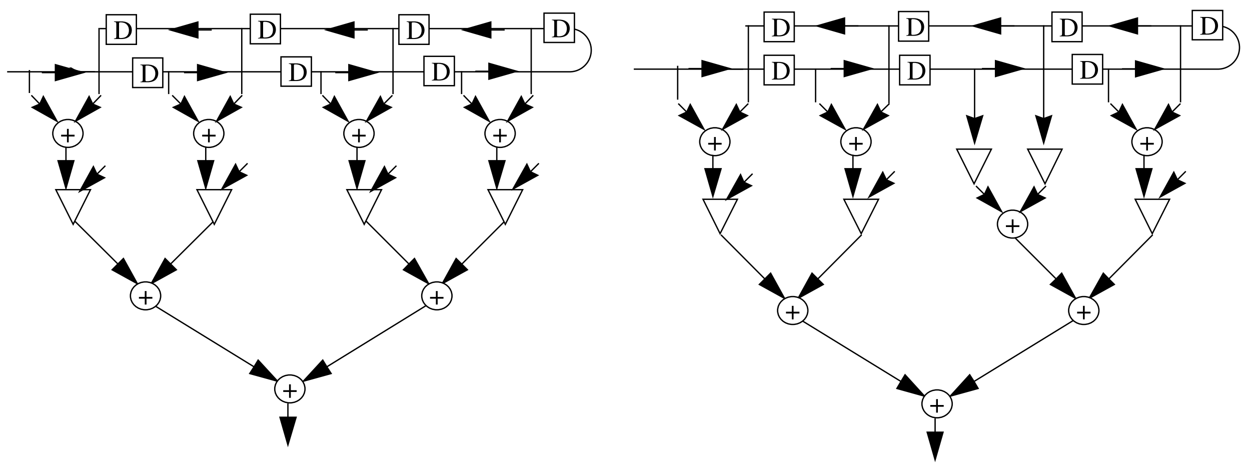


Figure 2: Design Space Exploration Framework using structural properties

3.2 Structural Properties

An important and widely used property of an algorithm is its size. The *number of multiplications*, for example, is a yardstick often used in the comparison of DSP algorithms. The suitability of an algorithm for a targeted application, however, is a function of not only size measures such as the number of operations and required word length, but also other structural qualities. This claim is well illustrated using the symmetric direct-form FIR filter of Figure 3a. Figure 3b shows the filter after distributivity is applied to one of the addition-multiplication pairs. It is assumed that an ASIC implementation is being used, where each operation takes 1 control step and the required throughput is 4 control steps. In both graphs, the critical path is 4, and all operations lie on the critical path. Note that the original filter structure has 7 additions and 4 multiplications, while the transformed one has the same number of additions but an additional multiply operation. It is easy to see that the implementation of Figure 3a requires 4 adders and 4 multipliers, while that of Figure 3b requires just 3 adders and 3 multipliers. The transformed filter, although it has a greater number of operations, has a structure which can be implemented with less execution units. The key differentiator between the two algorithms is not their size, but another property called *uniformity* (see Section 4.1.1).



(a) (b)

Figure 3: Size versus Uniformity Structural Property

There are many other properties like *size* and *uniformity* which can be used in design guidance. For each, there exist a number of measures that quantify different aspects of the property class. The following property set is not claimed to be exhaustive, however the experimental studies indicate that it does characterize the relevant sub-features of algorithms encountered so far. The set includes the following properties:

- *Size measures* includes quantities such as the number of nodes, the bit-width, the number of i/o operations, and the number of memory accesses.
- The *timing and structural freedom* are quantified through measures such as the ratio of sample rate to the critical path, the number and types of nodes on the ε-critical network (the subset of the graph which includes all nodes and edges on paths of length greater than (1-ε) times the critical path length), and various statistics of the scheduling slack.
- In describing the *uniformity* of a computation, the goal is to capture the degree to which operations and interconnect accesses are evenly distributed in time over the course of the computation.
- *Temporality* captures information about the lifetimes of variables in the computation. A computation is considered to be *temporally local* if the expected lifetimes of the variables are short. It is *temporally dense* if the measured maximum expected number of variables alive at any time is large.
- A computation is labelled *spatially local* if it has natural clusters of computation, where a significant amount of computation can be done independently.
- The idea behind measuring the *regularity* of a computation is to capture the degree to which common patterns appear. Note that a pattern can be a functional pattern of specific grouped types of nodes and edges, or a structural pattern such as a mesh interconnection.
- *Cyclic properties* involve measuring various characteristics of cycles such as the iteration bound and percentage of nodes in cycles.
- *Control flow properties* involve determining the structure and interaction between loops, and several bounds on the performance of multi-threaded implementations.

Methods for quantifying the majority of these structural properties have been derived. The usefulness and predictive accuracy of many of these structural properties have been confirmed, as demonstrated in section 4.0.

The properties themselves are independent of any particular implementation platform, and characterize only the computation. The architectural information however, determines which properties are relevant for a given design guidance task. For example, the iteration bound is an important indicator of speed on an architecture with parallelism, while it is not relevant for a uniprocessor implementation. On a uniprocessor, however, the types of operations and operator counts are relevant speed indicators.

The continuing development and definition of a set of structural properties is based on adherence to the following requirements and goals. First of all, it is desired to find a minimal, non-redundant set of properties, that is able to characterize all parts of the algorithm space. Secondly, the heuristic quantification of the property measures should be of low computational complexity. In all properties, the aim is for polynomial complexity in the number of nodes and edges of no greater than the 2nd or 3rd power. Thirdly, each property class and its quantitative definitions should be relevant for at least one of the targeted design guidance tasks.

3.3 Current Algorithmic, Architecture, and Synthesis Framework

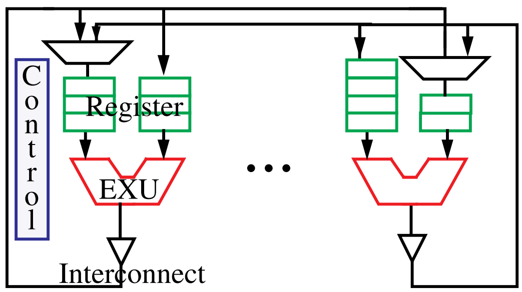
Obviously, testing the ideas, approaches, and algorithms related to structural properties . SO, in this paper we take a more fouces view and restrict our attention to

Our current targeted application domain includes DSP, video, image, graphics and other numerically intensive applications. Within this domain, algorithms with not only functional diversity, but also structural diversity (as dictated by the structural properties) are studied. The typical examples include various filters (FIR, IIR, Nonlinear), transforms (FFT, DCT, Convolution, wavelet, Hilbert), sorting, elementary functions, and subsystems (DFE, echo cancellation, dynamic programming, etc.). The algorithms are represented using a hierarchical signal flow graph, where nodes represent operators, and control/data edges the dependencies between them [Rab91a]. In terms of architectures, ASIC type datapath architectures have been of primary interest so far, but the proposed methodology and properties is being used for other platforms as well (programmable DSPs, etc.).

4.0 Case Study: Early Estimation of Custom ASIC Area

The properties and the general methodology for using them within a design guidance framework is best introduced through examples. This section thus looks at the properties in the context of one design guidance task, early estimation of custom ASIC implementation area. The hardware platform of interest is shown in Figure 4 and consists of hardware-shared execution units, registers, and interconnect. A finite state machine based controller is used to drive the datapaths.

The following sub-sections consider the estimation of each of these components, and the properties that are relevant for their estimation. In particular, the relations between components and properties that have been established are shown in Figure 4. For all components, size measures are also relevant.



Component	Relevant Properties
EXU	operator and register-access uniformity, freedom
Register	temporal density
Interconnect	interconnect uniformity, regularity
Control	regularity, temporal density, interconnect uniformity

Figure 4: Custom ASIC implementation platform and relevant properties

4.1 Execution Units

The execution unit area depends on both the number of units of each type, and the size of each unit. The number of units is related primarily to *uniformity*, but is also influenced by several measures in the *freedom* class. The size of each unit is a function of the required operation word length, which is a *size* property. As the key property for execution unit prediction, this section will concentrate on the introduction and description of *uniformity* property measures.

4.1.1 Uniformity

An object that is uniform has little or no variation. The property of interest here is *operator uniformity*, which characterizes the degree to which operator accesses are evenly distributed over the course of the computation. A distribution graph is constructed, which presents an expected number of operations to be executed in each clock cycle. These graphs form the signature of the proposed *uniformity* property. In high-level synthesis research, these types of distribution graphs have been studied extensively and form the backbone of some of the most popular synthesis algorithms. Paulin [Pau89] first proposed and used information from such graphs in the force directed scheduler as an indicator of the number of units that might be needed, and thus to guide the selection of control steps and operations for scheduling. This approach has since then been refined in terms of speed, efficiency, and the number of covered issues in more than 20 schedulers [Wal91]. These ideas are the starting point for the development and quantification of *uniformity* measures. The graph computation and relevant measures on these graphs which quantify the property class are presented in the next section.

4.1.1.1 Calculating Uniformity Graphs

As the exact times in which each operation will be executed is not known, and will not be known until after scheduling, it is necessary to determine the probability that the operation will be executed in a particular time step. To calculate the true probabilities, it is necessary to enumerate all possible schedules. Since this is a computationally formidable and complex problem, approximations of these probabilities must be derived. There are several ways in which to calculate the approximated uniformity graphs, with different ratios of computational complexity to accuracy of approximation. One solution is to assume that it is equally likely that an operation, x, be executed in any of the time slots between its as-soon-as-possible ($ASAP_x$) and as-late-as-possible ($ALAP_x$) times [Pau89].

In this case, the following contribution is made by an operation, i:

for $ASAP_i \leq t < ALAP_i$: $\quad u_i(k) = \dfrac{1}{ALAP_i - ASAP_i + 1}$ to each time slot $t \leq k < t + Duration_i$.

The values U(t) of the uniformity graph are the accumulation of the contributions from all operations of the specified hardware type.

For example, computation of the distribution graph for adder operations of Figure 3a and 3b results in Figure 5. Once

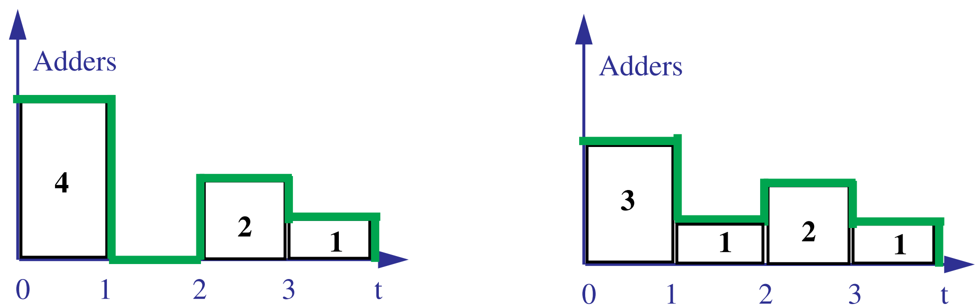


Figure 5: Distribution Graphs of Figure 3

the distribution graphs have been computed, it is necessary to quantify the amount of uniformity that exists. There are several measures of uniformity that correlate to the required number of execution units. The first is the maximum height of the distribution graph. Note that the maximum height of the graph can be considered an approximation of the maximum number of operations that need to be executed simultaneously, which therefore relates to an approximate number of required resources. The measure $maximumheight' = Max(1, maximumheight)$, is used since the number of units must always be at least one. Another measure of interest is the variance of the distribution graph, which while not relevant for execution units, indicates the potential of transformations which target alternations of ASAP and ALAP times (e.g. retiming).

Thus far, uniformity measures regarding operator accesses have been discussed. For the architectural model of Figure 4 however, another uniformity measure is important: that of register access. The reason for this is that the register files are single-ported, and thus only a single register in the file can receive data each control cycle. Thus hardware units are determined by the maximum register writes that need to execute simultaneously as well as the maximum operations needed to execute simultaneously. In order to account for this constraint of the architecture, distribution graphs of register accesses to register files of each execution unit type are computed. For each of the distribution graph measures, the dominant one (from operator or register bandwidth distribution graphs) is chosen.

4.1.1.2 Experimental Results

An extensive empirical study was conducted to detect the trends between actual numbers of units in implementation, and the uniformity measures. For each of 140 examples (20 application algorithms at a variety of sampling rates), the uniformity measures were computed, then the number of units was determined using the HYPER [Rab91a] synthesis system. The scatter plot of Figure 6 shows the strong correlation that exists between the uniformity max height and the actual number of units. For each design, the sums of the maximum heights for each hardware class was used:

$$\sum_{i=1}^{NumberofHardwareTypes} MaximumHeight_i .$$

The linear correlations of the total number of units to the sums of maximum height, and sums of maximum height' were 0.92 and 0.93, respectively.

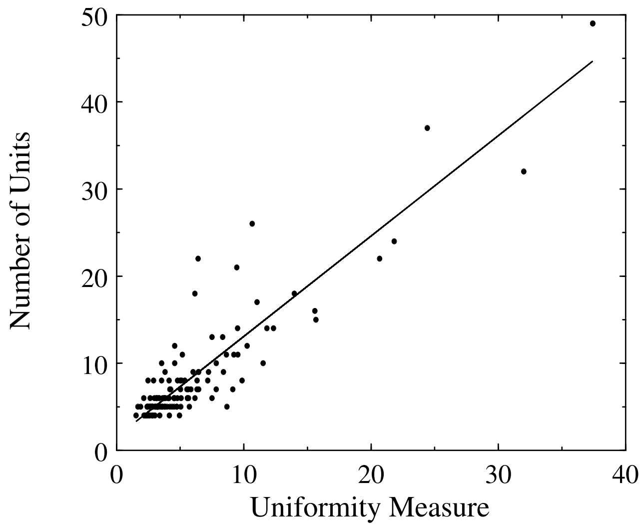


Figure 6: Maximum Height versus Actual Number of Units

4.2 Registers

The register area depends on both the number of registers, and the size of each register. The number of registers is related to a property called *temporal density.* The size of each register is a function of the required operation word length. As the key property for register prediction, this section will concentrate on the introduction and description of the *temporal density* property measure for predicting the number of registers.

4.2.1 Temporal Density

A computational flow graph can be thought of as temporally dense if the maximum number of variables that must be simultaneously kept alive is high. The following example illustrates the type of information to be captured. Consider the graphs of Figure 7a and 7b, both of which have the same number of nodes and edges. Assuming an available time of 4 control steps, we see that both graphs have all nodes on the critical path. Nevertheless, while implementation of Figure 7a

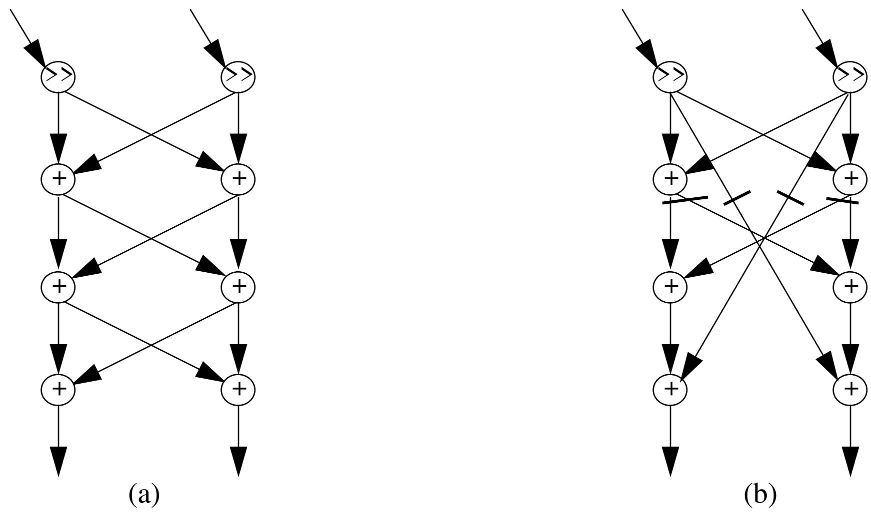


Figure 7: Temporal Density Measure: a) less dense, b) more dense

requires simultaneous storage of only 2 variables that of Figure 7b requires storage of up to 4 variables. The edges corresponding to the 4 variables which are alive after the 2nd control step are marked in Figure 7b.

4.2.1.1 Calculations of Lifetime Distributions for Variables

In general, operator execution times are not known until scheduling is performed. These times determine the lifetimes of the variables and thus the number alive at any one time. Two heuristic measures have been defined, which approximate the expected lifetime distributions of the variables in the flow graph. For these calculations, operations are assumed to have equal probability of executing in any of its valid time slots.

Deterministic Approach (Midpoint Approximation)

In this approach, the lifetime of a variable is approximated from the expectation of the execution times of its production and consumption nodes. For each variable, v, the expected start and end times are calculated, as given by the following formulas:

$$\mathrm{ExpectedStart}_v = 0.5 \times (\mathrm{ASAP}_x + \mathrm{ALAP}_x) + \mathrm{Duration}_x \,, \tag{1}$$

$$\mathrm{ExpectedEnd}_v = \max_{y_i \in y} \left(0.5 \times (\mathrm{ASAP}_{y_i} + \mathrm{ALAP}_{y_i}) + \mathrm{Duration}_{y_i}\right) \tag{2}$$

where x is the producer node, and $y_i \in y$ are the consumer nodes. Figure 8 illustrates this calculation.

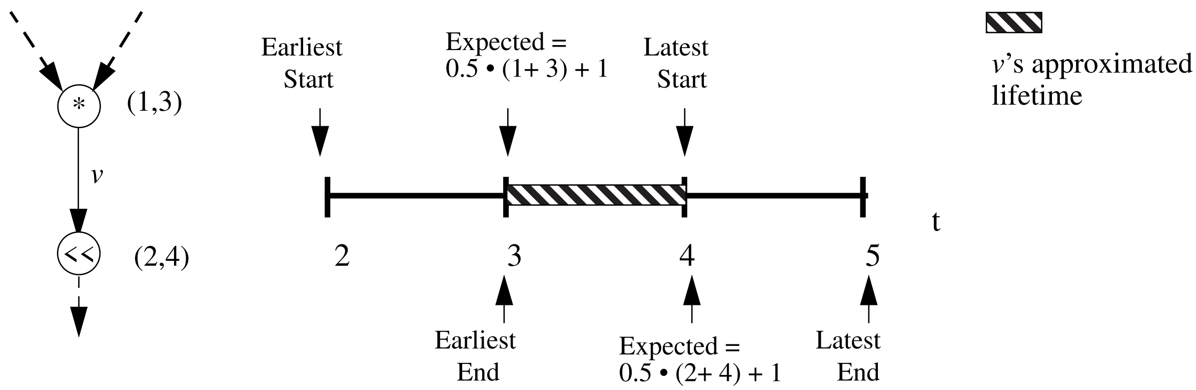


Figure 8: Midpoint Approximation of Variable Lifetime

Probabilistic Approach (Distributed Approximation)

Another approach involves calculating conditional probabilities, $P_v(t)$, that the variable v will be alive during time slot, t. This is calculated locally for a variable, taking into account the slacks of the consumer and producer of the variable, and the dependency between them. For a variable that fans out to several operations, these values are independently calculated for each fan out edge, then the maximum probability of existence is used at each time slot. Depending on producer and consumer ASAP and ALAP times, two cases are defined for the computation. The second case occurs when a variable is produced and consumed in different sample periods. The first case is shown below.

Case 1: $\mathrm{ASAP}_{y_i} > \mathrm{ASAP}_x$

for $\mathrm{ASAP}_x + \mathrm{Duration}_x \le t < \mathrm{ALAP}_{y_i} + \mathrm{Duration}_{y_i}$:

$$P_v(t) = \left(\sum_{j = ASAP_x}^{t - Duration_x} P_x(\mathrm{X} = j) \bullet P_{y_i}\left(Y_i > t - Duration_{y_i} \middle| \mathrm{X} = j \right) \right) \tag{3}$$

Case 2: $\mathrm{ASAP}_{y_i} \le \mathrm{ASAP}_x$

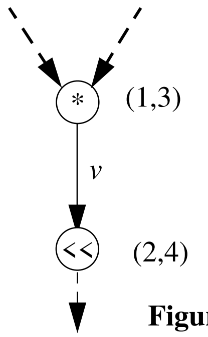
for $\mathrm{ASAP}_x + \mathrm{Duration}_x \le t < \mathrm{ALAP}_x + \mathrm{Duration}_x$:

$$P_v(t) = \left[\sum_{j = t - Duration_x - 1}^{ASAP_x} P_x(\mathrm{X = j}) \sum_{k = t - Duration_x + 1}^{j} P_{y_i}(Y_i = k) \right] + \sum_{z = ASAP_x}^{t - Duration_x} P_x(X = z) \tag{4}$$

for $\mathrm{ALAP}_x + \mathrm{Duration}_x \le t < \mathrm{ASAP}_{y_i} + \mathrm{Duration}_{y_i}$: $P_v(t) = 1$ (5)

$$\text{for } \mathrm{ASAP}_{y_i} + \mathrm{Duration}_{y_i} - 1 < t < \mathrm{ASAP}_x + \mathrm{Duration}_x \quad : \quad P_v(t) = 1 - \sum_{j = ASAP_{y_i}}^{t} P_{y_i}(Y_i = j) \tag{6}$$

Figure 9 shows a simple example of these calculations. Note that this is the probability that a variable is alive at time t, and the area under the curve should not necessarily be 1.



$P_v(2) = 1/3 \bullet 1 = 1/3;$

$P_v(3) = 1/3 \bullet 2/3 + 1/3 \bullet 1 = 5/9;$

$P_v(4) = 1/3 \bullet 1/3 + 1/3 \bullet 1/2 + 1/3 \bullet 1 = 11/18$

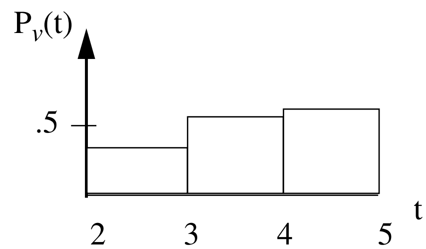


Figure 9: Distributed Approximation of Probability that the variable *v* is alive at time t

4.2.1.2 Temporal Density of a Graph

Based on the calculations of the previous section, the temporal density measure of the computational flow graph can be found. Consider first that the deterministic midpoint approximation is used for determining the lifetimes of all variables. In this case, a cyclic lifetime graph displaying the lifetimes of all variables in the graph can be constructed. The temporal density measure, T_m, is defined to be the maximum number of variables that cross a vertical line situated at any time t.

The graph temporal density measure can be found in the same manner when the probabilistic measure of variable temporal locality is used. The measure T_d is defined as:

$$T_d = \underset{t \in [0 \ldots AvailableTime)}{\mathrm{Max}} \left(\sum_{v=1}^{NumVariables} P_v(t) \right) \tag{7}$$

These measures estimate the maximum number of variables expected to be alive at any given time.

4.2.1.3 Experimental Results

An extensive empirical study was conducted to detect the trends between actual numbers of registers, and the temporal density measures. The scatter plots of Figure 10 clearly indicate a strong correlation between temporal density and the number of registers in the final implementation. The linear correlation between the deterministic and probabilistic measures and number of registers is 0.97 for both. For the purpose of register estimation, either measure works equally well.

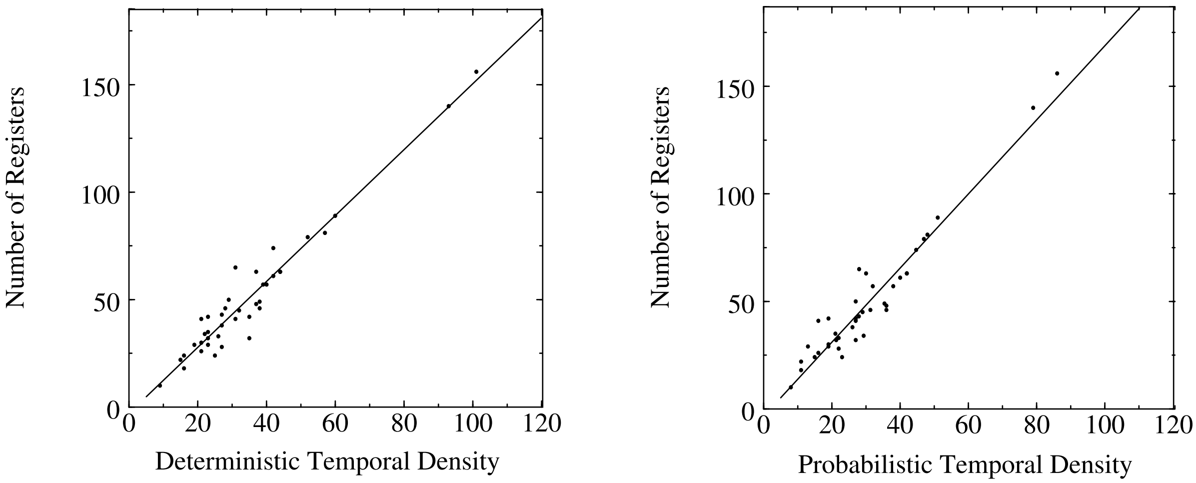


Figure 10: Number of Registers versus Temporal Density Measures for 40 Examples

4.3 Interconnect

Often the largest component of semi-custom ASIC implementations is the interconnect area, which consists of not only wiring area, but also the supporting muxes, buffers, and white space introduced. This area depends on the bitwidths of the datapaths, number of each interconnect element, and amenability of the design to compact routing. Several algorithmic properties are good predictors of the later 2 elements of interconnect area. These are *regularity,* which is related to the descriptive complexity of the graph, *operator uniformity*, and *interconnect uniformity,* which is similar to uniformity of execution unit accesses considered previously. Uniformity for interconnect can be easily derived by substituting transfer operations for execution units in the previous treatment of uniformity of operators (Section 4.1.1). Therefore, this section will concentrate on the introduction and description of the *regularity* property measure.

4.3.1 Regularity

The influence of regularity on physical layout characteristics has been observed by several high level synthesis groups [Not91, Rao92]. Regularity has only been treated qualitatively until now, and the exploration of its potential is in an early phase.We define *regularity* using the following simple formula:

$$\text{Regularity} = \frac{\text{Size}}{\text{Descriptive Complexity}} \tag{8}$$

The *size* component is the number of operations and data transfers executed in the computation. The *descriptive complexity* is a measure of the shortest description from which the graph can be reproduced.

A regular graph has repeated patterns of nodes and interconnections, and can thus be concisely described. The inverse proportional relation is defined since lower *complexity* implies higher regularity. It is also necessary to consider how *complexity* scales with the *size* of the computation. The proportional relation is established since *complexity* is a function of the size of the graph being described. In comparing two graphs with the same complexity but different sizes, the larger would be considered more regular.

4.3.1.1 Heuristic Complexity of a Graph

Motivated by the underlying ideas in descriptive complexity of strings (also called Kolmogorov complexity) [Li90], a measure of graph complexity has been developed. The Kolmogorov complexity of a string is “the size of the smallest program that, starting with blank memory, outputs the string then terminates” [Li90]. Unlike a graph, a string has a natural ordering of symbols from left to right. In particular, a symbol has a single left and single right neighbor, and can only be the right or left neighbor of a single other symbol. In a graph, a symbol can have an arbitrary number of neighbors. A graph has more degrees of freedom (represented by its interconnect structure) than a string, and finding a Kolmogorov complexity measure for a graph is more difficult problem than finding the Kolmogorov complexity of a string. However, even for a string, the problem is undecidable [Li90]. Therefore, the most useful approach is the establishment of a heuristic graph complexity measure, which can be efficiently computed, while preserving the key ideas.

Given an encoding scheme (language by which to describe the graph and a measure of the length of a program), the complexity of a graph is defined as the length of the shortest program which can represent the graph. The main components in determining the complexity of a graph are 1) a language by which to specify the graph and a quantification of the “length” of a program in this language, and 2) a methodology to find a short program.

Encoding (Language and Length)

Description of a graph involves describing the types and numbers of nodes, and the exact interconnections between them. A pseudo-descriptive language was developed for this purpose. Note, however, that the language is purely for conceptual purposes, as the actual information needed to determine the “program length” can be obtained from the graph directly. The pseudo language has 2 primary instructions: instantiate and connect. The instantiate instruction takes 2 parameters - the operation to instantiate, and the number of instances of that operator that are desired. If the operator is not a primitive of the language (i.e. templates, loops, subgraphs), then the operator must be described using basic elements of the language.

For a given graph, there exist a number of programs that can describe it, some being more compact than others. Consider, for example, the graph of Figure 11. The Figures 11a -11f show various descriptive versions of this graph, where the degree to which a subroutine is used is indicated on the corresponding graphs. In each of the flow graphs, a different number of patterns has been detected. Sample descriptive programs are shown for the first two.

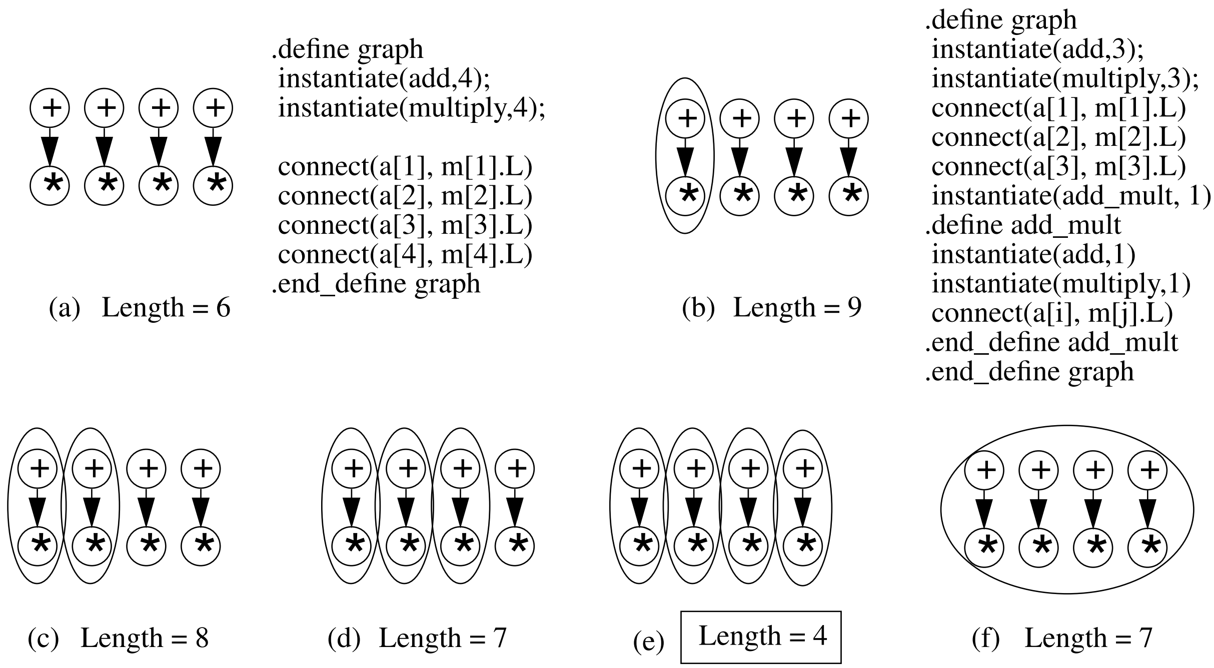


Figure 11: Example of Language and Program Length

The length of a program in this pseudo-descriptive language is defined as the number of instantiate and connect statements. The original cost of describing the graph if no patterns are detected is 6. By detecting a single pattern, but none of its copies, the cost actually is 9. Detection of the single pattern has not contributed anything in terms of making the system easier to describe and has in fact made the system harder to describe. Finding all of the patterns as shown in Figure 11e results in the lowest cost of 4, which is the complexity of the graph. Another example is shown in Figure 12. This example emphasizes the fact that the majority of complexity in graph descriptions is due to the description of the interconnect structure. Both graphs of Figure 12a and 12b have the same types and numbers of nodes. The flat description of either graph is 10. The interconnect structure of Figure 12a is much more regular, however, and thus allows the efficient use of subroutines in describing the graph. The complexity of the graph of Figure 12a is 5, while that of 12b is 10.

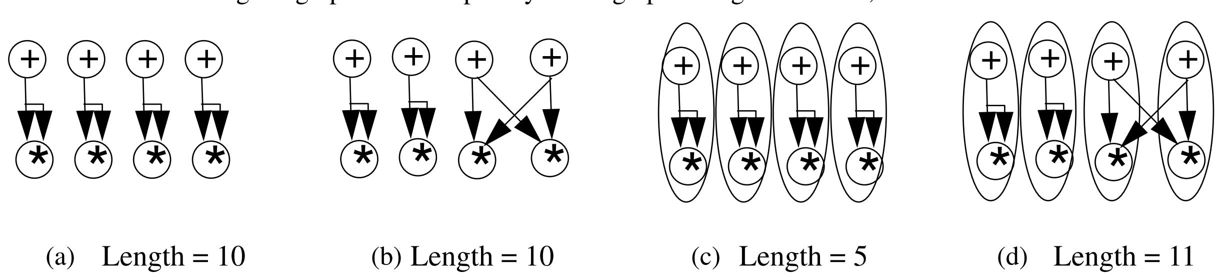
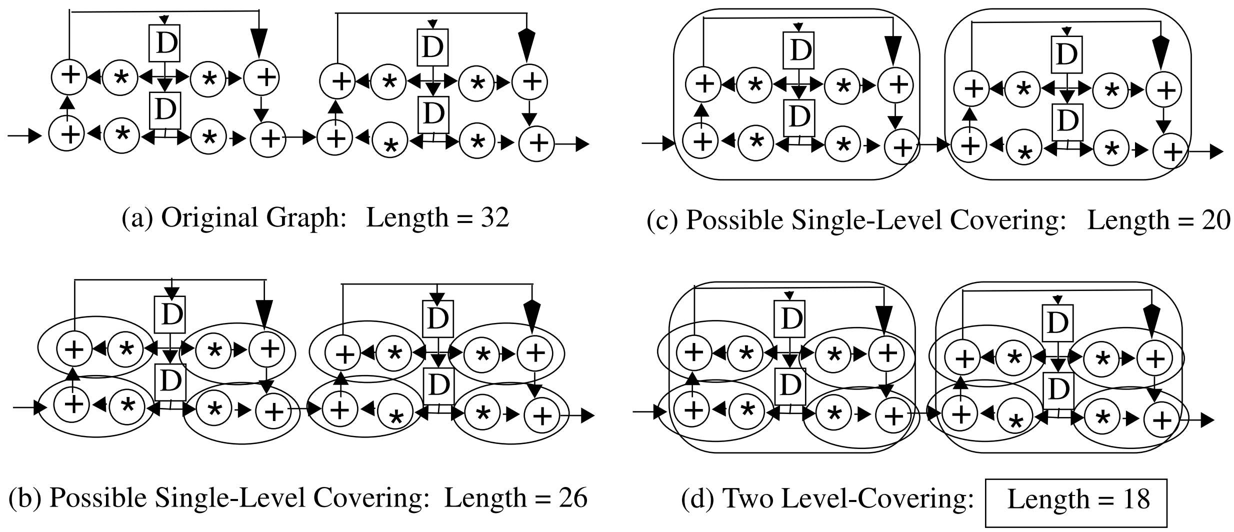


Figure 12: Encoding Example: Majority of Complexity is in Interconnect Structure

Methodology to find a Short Program

Template matching techniques are clearly the tool of choice for the detection of the common patterns that enable a more concise description of a graph. The template covering routines involve both template generation and hierarchical template covering. The template matching routines of [Cor93] are used in conjunction with a template selection for area



(a) Original Graph: Length = 32

(c) Possible Single-Level Covering: Length = 20

(b) Possible Single-Level Covering: Length = 26

(d) Two Level-Covering: Length = 18

Figure 13: Hierarchical Template Matching

minimization [Cor94]. The objective function for the *regularity*-based template selection is changed to the encoding cost. Thus the template selection targets minimum encoding.

Figure 13 shows the basic template matching methodology. The covering must be done in a hierarchical fashion in order to capture the regularity at different levels of granularity. In this example, applying a mult-add template and a biquad template results in the minimum descriptive length.

4.3.1.2 Examples

Design experience shows that in semi-custom ASIC applications, *regularity* is a good indicator of interconnect area. High regularity corresponds to simple description of the interconnections between nodes in the signal flow graphs, which can translate into *fewer* types of interconnections, and interconnections which are more highly utilized. Unfortunately, current assignment and scheduling tools do not detect and exploit regularity. The design implications for a regular graph are thus not always propagated to the final implementation if the assignment and scheduling do not preserve the regularity of the initial specification. Adaptation of current tools to use regularity is the topic of future research which aims at a comprehensive exploration of structural properties as design goals and criteria in system-level synthesis tools. This section looks closely at several examples to illustrate the underlying trend of the regularity property. Note that the absolute regularity numbers are relative metrics only when derived for algorithms which target the same application.

First consider the two cascade recursive filters of Figure 14. Both can be implemented for a throughput of 10 control steps using 1 multiplier, 1 adder, and 1 transfer unit. Interconnect requirements are shown in Table 1. The more regular version indeed results in less busses and muxes.

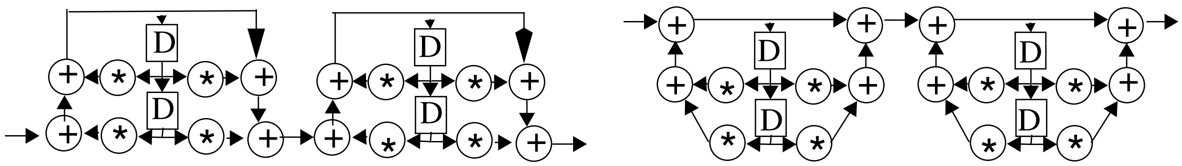


Figure 14: 4th order IIR structures: a) Regular version; b) Less Regular Version

Structure	Regularity	Size	Number of Busses	Number of Muxes
Less Regular Version	2.45	49	8	4
Regular Version	2.72	49	6	2

Table 1: Regularity for Interconnect Area - Two Versions of a 4th order IIR filter

Consider next the well-known wave digital and ladder IIR filter structures [Mit93]. For filters of the same order, both have approximately the same size, except that the ladder structure is sometimes slightly larger. The interconnect requirements are shown in Table 2. The more orderly network of communications that is present in the ladder structure translates into a simpler overall interconnect structure, regardless of its larger size.

Structure	Regularity	Size	Number of Busses	Number of Muxes	Number of Buffers
8th order wave digital	1.7	127	6	5	11
8th order ladder	3.5	132	4	4	5

Table 2: Regularity for Interconnect Area - wave digital versus ladder structures

4.4 Control

The controller can be modelled as a FSM plus control logic or a sequencer with microcode ROM. The key elements in determining the controller area are the number of states and the width of the output control bus (signals to the registers, muxes, tri-state buffers, and execution unit in each datapath). Early estimation of controller size is therefore based on detecting trends in these two parameters. For the architecture model described earlier (Figure 4), a majority of the control lines are dedicated to the registers and the interconnect elements. The measures described in sections 4.2 and 4.3 (*temporal density*, *interconnect uniformity*, and *regularity*) can thus be used for correlation to the output bus width. For the number of states metric, the *regularity* measure is important. In particular, the number of states is related to $\mathrm{Throughput}/\mathrm{Regularity}$.

The most direct example can be seen by considering a computation with loops. If the for-loops are unrolled, and the synthesis tools are not able to produce a regular schedule and assignment, the controller will combinatorially explode in size. On the other hand, if the loops are preserved, the regularity of the structure will be reflected in the ability to share states, resulting in a smaller controller in contrast to an unrolled version. Even when graphs do not have for-loops, high regularity can result in designs which can share states, leading to smaller controllers (for instance using sub-routines).

Table 3 shows four examples.The first two examples are very regular, and thus have very small states-throughput ratios. In the 3rd example, the regularity was not identified and preserved during scheduling, resulting in a high states-throughput ratio. The last example has low regularity, and thus the states-throughput ratio is large as expected. Note that in some cases, high level synthesis tools intentionally do not preserve regularity, since it conflicts with other goals, such as small data path area. The properties and design guidance tools indicate these trade-offs, however, so the designer can make his or her own trade-offs.

Structure	Regularity	Throughput (#control steps)	Number of States	Number of States/ Throughput
histogram	297	22000	47	.002
cordic	14	100	5	.05
cordic_unrolled	10	77	77	1
8th order wave digital	1.7	25	25	1

Table 3: Examples of relation between properties and the Number of states

4.5 Summary

This case study has illustrated several relations that exist between ASIC area components and properties. In this context, the *uniformity, temporal density,* and *regularity* properties were presented. Most of our work to date has concentrated on custom-ASIC implementations due to the ease of implementation and experimentation provided by high-level synthesis tool infrastructure. It appears that at least initially, high-level synthesis tools and ideas are likely to form an integral part of system-level design, as they form a natural point from which to abstract.

This work forms a foundation for extension to other design metrics. For example, area prediction is in fact crucial for accurate power prediction. In addition, the methodology holds in generalization to other implementation platforms: for a given metric on a particular platform, a few relevant properties are combined to construct a prediction and a confidence metric as feedback to the user.

5.0 Conclusion and Future work

In behavioral synthesis, a key task is the exploration of the design space, which involves the tasks of comparing computational structures, architectures, and the power of optimizing transformations. Design guidance is a necessary and indispensable part of any effective system-level synthesis system.

This paper has presented a framework for system-level synthesis exploration which is founded upon characterization of algorithms by their structural properties. As evidenced by the increasing use of many of these idioms within the design community, structural properties such as spatial locality, regularity, the length of the critical path, cycle properties (e.g. iteration bound), and reconvergences are believed to be related to the quality of a final design. This is however, the first presentation of a comprehensive set of quantified structural properties. The approach proposes a conceptually new point of view from which to address the problems of understanding and making informed system-level design decisions. The presented theoretical and statistical study of several intangible structural properties of a computation have demonstrated their influence on the cost efficiency of ASIC designs. Other experimentation has indicated that these few fundamental algorithmic properties can go a long way in classifying algorithms and predicting how amenable they are to a *variety* of hardware platforms. Future work involves building statistically validated models, where the properties serve as input parameters to the models, and the models predict various performance metrics. There will also be continued study into the property definitions, with emphasis on their relation to other hardware platforms. The hope is that the presented property classification scheme can form a building block for a plethora of applications, including not only performance estimation and modelling, but also algorithm design, and synthesis algorithm optimization.

6.0 References

[Amd64] G.M. Amdahl, G.A. Blaauw, F.P. Brooks, "Architecture of the IBM System/360," *IBM Journal of Research and Development,* Vol. 8, No. 2, pp. 87-101, 1964

[Bre74] R. Brent, "The parallel evaluation of general arithmetic expressions," *Journal of the ACM,* Vol. 21, pp. 201-206, 1974.

[Cha94] A. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, R. Brodersen, "Optimizing Power Using Transformations," submitted to *Trans. on CAD,* 1994.

[Cho92] P. Chou, R. Ortega, G. Borrielo, "Synthesis of Mized Hardware-Software Interface in Microcontroller-Based Systems," *IEEE/ACM Proc. of the ICCAD*, pp. 488-495, 1992.

[Cor93] M. Corazao, M. Khalaf, L. Guerra, M. Potkonjak, J. Rabaey, "Instruction Set Mapping for Performance Optimization," *IEEE/ACM Proc. of the ICCAD*, 1993.

[Cor94] M. Corazao, "Instruction Set Mapping for Performance and Area," *MS Thesis in preparation*, 1994.

[Ern93] R. Ernst, J. Henkel, T. Benner, "Hardware-Software Cosynthesis for Digital Systems," *IEEE Design & Test of Computers,* Vol. 10, No. 4, pp. 64-75, 1993.

[Gup93] R. Gupta, C. Coehlo, G. De Micheli, "Program Implementation Schemes for Hardware-Software Systems," *IEEE Computer*, Vol. 27, No. 1, pp. 48-55, 1993.

[Hil90] M. Hill, "What is Scalability?" *Computer Architecture News,* Vol. 18, No. 4, pp. 18-21, 1990.

[Jai92] R. Jain, A. Parker, N. Park, "Predicting system-level area and delay for pipelined and non-pipelined designs," *IEEE Trans. on CAD*, vol. 11, Aug. 1992.

[Kar90] R. Karp, V. Ramachadran, "Parallel Algorithms for Shared-Memory Machines," in *Handbook of Theoretical Computer Science, Volume A*, ed. by J. v. Leeuwen, Vol A. Ch. 17, The MIT Press, Cambridge, MA, 1990.

[Kal93] A. Kalavade, E.A. Lee, "A Hardware-Software Codesign Methodology for DSP Applications," *IEEE Design & Test of Computers*, Vol. 10, No. 3, pp. 16-28, 1993.

[Knu71] D. Knuth, "An Empirical study of FORTRAN programs," *Software Practice & Experience,* Vol. 1, No. 2, pp. 105-133, 1971.

[Kun82] H.T. Kung, "Why systolic Architectures?," *IEEE Computer,* Vol. 15, No. 1, pp. 27-36, 1982.

[Kun88] S.Y. Kung, *VLSI Array Processors,* Prentice Hall, Englewood Cliffs, NJ, 1988.

[Kur89] F. Kurdahi, A. Parker, “Techniques for area estimation of VLSI layouts,” *IEEE Trans. on CAD*, Vol. 8, No. 1, 1989.

[Li90] M. Li, P.M.B. Vitanyi, “Kolmogorov Complexity and its Applications,” in *Handbook of Theoretical Computer Science, Volume A*, ed. by J. v. Leeuwen, pp. 187-254, The MIT Press, Cambridge, MA, 1990.

[Lid94] D. Lidsky, J. Rabaey, “Low-Power Design of Memory Intensive Functions Case Study: Vector Quantization,” submitted to *VLSI Signal Processing*, 1994

[Mit93] S. Mitra, J. Kaiser, *Handbook of Digital Signal Processing,* John Wiley, New York, NY, 1993.

[Not91] S. Note, W. Geurts, F. Catthoor, H. De Man, “Cathedral-III: Architecture-Driven High-Level Synthesis for High Throughput DSP Applications,” *Proc. 28th DAC*, pp. 597-602, 1991.

[OhE85] M. O’hEigeartaigh, S. Lenstra, A. Rinnoy Kan (eds.), *Combinatorial Optimization: Annotated Bibliographies,* John Wiley, New York, NY, 1985.

[Pat90] D. Patterson, J. Hennessy, *Computer Architecture: A quantitative approach,* Morgan Kaufmann, 1990.

[Pau89] P. Paulin, J. Knight, “Force-Directed Scheduling for the Behavioral Synthesis of ASIC’s,” *IEEE Trans. on CAD*, Vol. 8, No. 6, pp. 661-679, 1989.

[Pot93] M. Potkonjak, J. Rabaey, “Exploring the DSP Algorithm Design Space using HYPER,” *VLSI Signal Processing*, pp. 123-131, 1993.

[Rab91a] J. Rabaey, C. Chu, P. Hoang, M. Potkonjak, “Fast Prototyping of Data Path Intensive Architectures,” *IEEE Design & Test Magazine*, Vol. 8, No. 2, pp. 40-51, June 1991.

[Rab91b] J. Rabaey, M. Potkonjak, "Complexity Estimation for Real Time Application Specific Circuits," *Proc. IEEE European Solid State Circuits Conference*, Milan, Sept. 1991.

[Rao92] D. Rao, F. Kurdahi, “Partitioning by Regularity Extraction,” *Proc. 29th DAC*, pp. 235-238, 1992.

[Sha93] A. Sharma, R. Jain, "Estimating Architectural Resources and Performance for High-Level Synthesis Applications," *IEEE Trans. on VLSI Systems*, Vol. 1, No. 2, pp. 175-190, 1993.

[Wal91] R. Walker, R. Camposano (ed.), *A Survey of High-Level Synthesis Systems*, Kluwer Academic, Boston, 1991.

[Woo94] N-S. Woo, A. Dunlop, W. Wolf, “Codesign from Cospecification,” *IEEE Computer*, Vol. 27, No. 1, pp. 42-48, 1994.