Functional Model Exploration for Multimedia Applications via Algebraic Operators

Shinjiro Kakita
Sony Corporation
Kita-shinagawa, Tokyo 141-0001, Japan
Shinjiro.Kakita@jp.sony.com

Yosinori Watanabe
Cadence Berkeley Laboratories
1995 University Ave., Berkeley, CA 94704, USA
watanabe@cadence.com

Douglas Densmore, Abhijit Davare, Alberto Sangiovanni-Vincentelli
Dept. of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720, USA
{*densmore, davare, alberto*}*@eecs.berkeley.edu*

Abstract

An optimized functional design space exploration method for multimedia applications is proposed. The basis of the method is a way of representing the dependency and the concurrency of an application in a compact form exploiting algebraic operators and expressions. The optimized design process consists of mapping one of the possible expressions in the application space onto a concurrent architecture. We use the Metropolis design framework to demonstrate the effectiveness of the procedure using an FPGA architecture as the target implementation platform. The advantage of using this platform is the availability of models that approximate well the performance of the final implementation when performing the mapping from function to architecture thus yielding a robust design methodology.

1 Introduction

Embedded system design has become increasingly relevant in the electronic industry evolution towards products that couple complexity with tight requirements on time to market and cost. The trend is clear: the use of highly concurrent and programmable platforms that allow quick design cycles with reasonable cost and performance will be pervasive. The design problem then becomes one of mapping a given application onto the architecture of a flexible implementation platform. Choosing a good mapping is non trivial as the application has to be manipulated to expose an amount of concurrency and complexity that matches what is offered by the implementation platform: this is the design space exploration problem we tackle in this paper.

The functionality required by an application is best seen as a denotational specification: in this case, the specification describes *what* we would like to do including constraints, and not *how* it should be done. Often the design process starts with a mix of denotational and operational descriptions; the operational description outlines how the application should operate, it already constraints the way the functionality will be implemented, thus potentially eliminating a vast part of the design space that could be explored. For example, a specification of an MPEG algorithm is often expressed in a sequential language such as "C" that cannot expose the degree of concurrency that is possible given the features of the algorithm. Designers when confronted with execution platforms that exhibit parallelism attempt to partition the code so that the final implementation exploits the services offered by the architecture. It is not surprising that this approach is error prone, tedious, and may stop far from optimality.

Our approach is to use a design process where we wish first to extract a set of concurrent behaviors that are implied by the functionality of the system. This set is represented in the most compact possible way and, at the same time, can be explored efficiently. Second, we wish to map, possibly optimally, one of these behaviors to a given architecture. The essential part in this phase is the selection of the behavior from the set identified in the first phase. This design process is similar to the one that is commonly in use for logic synthesis; the challenges are related to the choice of the representation of the behaviors (in logic synthesis the choice has been standardized to Boolean networks) and of the exploration algorithm.

The design space to explore in the functional domain is very large, as many are the ways to combine the operations needed to achieve the desired functionality including the choice of the granularity of the basic elements to use in

Proceedings of the Sixth International Conference on Application of Concurrency to System Design (ACSD'06)
0-7695-2556-3/06 $20.00 © 2006 IEEE

the decomposition. In this paper, we propose a method that effectively enumerates a large number of these possibilities in a systematic manner. We take as input a set of operations in a program that describes the functionality and their dependence. We define an algebra to represent the relations, and laws that allow transformations of the algebraic expressions without changing the functional correctness of the expressions. By successively applying these transformations, starting from an expression that corresponds to a particular structure of the operations (e.g., the original functional description), we obtain a transitive closure of algebraic expressions in which each expression represents a valid way of executing the operations with the concurrency represented by the expression.

We demonstrate the effectiveness of this approach by applying the method to a multimedia application, where we focus on operations in loop structures of the original descriptions. The expressions enumerated by the method represent possible re-structuring of the loops and their operations. Applications in this domain are characterized by the presence of many loop structures [11] that will be restructured using our approach. As already mentioned, the restructuring must be guided by optimization criteria and constraints that are often related to the implementation architecture. Hence, to demonstrate fully the method, we chose a flexible architecture that can be reconfigured easily, the Xilinx Virtex II Pro platform [13], and performed design space exploration with the goal of optimizing timing performance and workload balances. Experimental evidence shows that small differences in loop structures and operations result in dramatic effects on the quality of implementations and that our method selects a good set of candidate implementations.

The rest of the paper is organized as follows: Section 2 defines algebraic operators and laws. Section 3.1 details the method to enumerate possible functional models with these algebraic laws. Section 3.2 uses those operators and laws to create concise formal descriptions of potential functional model realizations. Section 4 explains how to partition functional models for implementation on computing resources. Section 5 provides a case study which ties these techniques together for an exploration of a subset H.264 application. Conclusions are provided in Section 6.

2 Algebraic Operators and Laws

In this section, algebraic operators and laws are defined to express and manipulate *functional topologies*. A functional topology is a collection of *computational blocks* and *dependency relations* between the computational blocks. A computational block may be defined at any abstraction level, e.g. an instruction, statement, or an entire process. The composition laws for the operators allow exploring functional topologies that are equivalent to the seed of the search. The operators and the laws are essentially term rewriting rules that have been used for code optimization in software and more recently for hardware implementations. The novelty of our approach is in the use of the rewriting rules to generate implicitly a large search space to map optimally a given programmable platform.

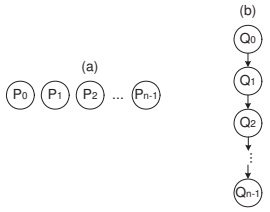


Figure 1. Parallel and Sequential Composition of Computational Blocks

2.1 Operator Definition

After [7], we represent parallel composition and sequential composition of computational blocks with operators $\parallel$ and $\circ$ respectively.

Let P and Q be two computational blocks. Parallel composition of P and Q is denoted by $P \parallel Q$, which denotes that unit P and unit Q can be executed concurrently. Meanwhile, their sequential composition is denoted by $P \circ Q$, which denotes that unit P must be followed by unit Q.

We introduce two more operators, $\prod$ and $\bigodot$. Figure 1 illustrates two types of directed acyclic functional topologies. An arrow in Figure 1 represents a dependency relation which determines the execution order of adjacent nodes. Figure 1 (a) depicts that computational blocks $P_i(i = 0, \cdots, n-1)$ are executed concurrently, and the relation is expressed with parallel operators $\parallel$ and $\prod$ as Equation (1). Figure 1 (b) represents that $Q_i(i = 0, \cdots, n-1)$ are executed sequentially, where Q_0 needs to be followed by Q_1 for instance, and Equation (2) gives the relation.

$$\prod_{i=0}^{n} P_i = P_0 \parallel P_1 \parallel P_2 \parallel \cdots \parallel P_{n-1} \tag{1}$$

$$\bigodot_{i=0}^{n} Q_i = Q_0 \circ Q_1 \circ Q_2 \circ \cdots \circ Q_{n-1} \tag{2}$$

Sequential operators bind more strongly than parallel operators. This defines the **order of operations** and is required for the following axioms.

Finally suppose that we have the three statements in Figure 2 (a). Notice unit P and unit Q have a dependency relation. P and R have a concurrency relation and so do Q and R. Therefore the dependency graph is expressed as Figure 2 (b) and the algebraic expression is given as follows in Equation (3).

$$P \circ Q \parallel R \tag{3}$$

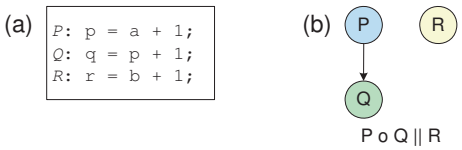


Figure 2. Code Snippet and Corresponding Potential Functional Topology

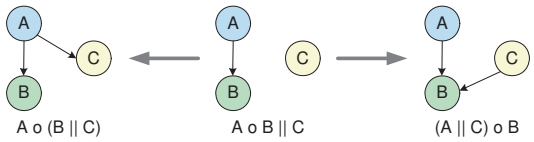


Figure 3. Distributive Law Example

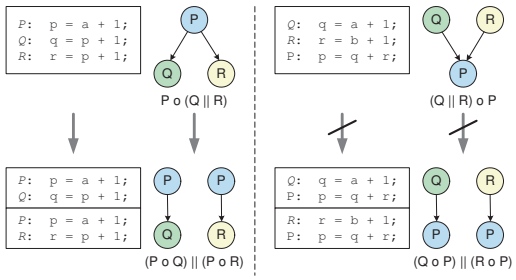


Figure 4. Split Law Example

2.2 Operator Axioms

Five laws regarding operators are defined. Applying these laws to a functional topology creates another valid functional topology. Let A, B, C, P, Q, and R be computational blocks.

2.2.1 Commutative Law

$$A \parallel B = B \parallel A \tag{4}$$

The commutative law defines that terms with a parallel operator commute, but those with a sequential operator do not commute.

2.2.2 Associative Law

$$A \parallel (B \parallel C) = (A \parallel B) \parallel C = A \parallel B \parallel C \tag{5}$$

$$A \circ (B \circ C) = (A \circ B) \circ C = A \circ B \circ C \tag{6}$$

The associative law shown in Equations (5) and (6) allows for various groupings of a given algebraic expression.

2.2.3 Substitutive Law

$$\parallel \Rightarrow \circ \tag{7}$$

$$\prod \Rightarrow \bigodot \tag{8}$$

The substitutive law (7) states that a parallel operator $\parallel$ can be replaced with a sequential operator $\circ$, and similarly $\prod$ can be replaced with $\bigodot$ as (8). Note that the opposite does not apply. $\Rightarrow$ denotes "replacement".

2.2.4 Distributive Law

$$A \circ B \parallel C \Rightarrow (A \parallel C) \circ B \tag{9}$$

$$A \circ B \parallel C \Rightarrow A \circ (B \parallel C) \tag{10}$$

The distributive law is equivalent to an edge addition as shown in Figure 3. The topology in the center of the diagram corresponds to the left hand side of Equations (9) and (10). The left portion of the diagram is Equation (10) and the right portion is Equation (9).

2.2.5 Split Law

$$P = P \parallel P \tag{11}$$

The two split Ps in the right hand side have the same functionality and state executing concurrently. Figure 4 illustrates an example of the split law applied to a software program. As shown in Figure 4 (left), P before a sequential operator can be split with the law. It is equivalent to Equation (12). Meanwhile P after a sequential operator **cannot** be split as shown in Figure 4 (right) and Equation (13).

$$P \circ (Q \parallel R) = (P \circ Q) \parallel (P \circ R) \tag{12}$$

$$(Q \parallel R) \circ P \neq (Q \circ P) \parallel (R \circ P) \tag{13}$$

An expression (14) shows a composition method for two split Ps. When expression $f1$ includes P and is connected with a sequential operator, the P which appears later is eliminated.

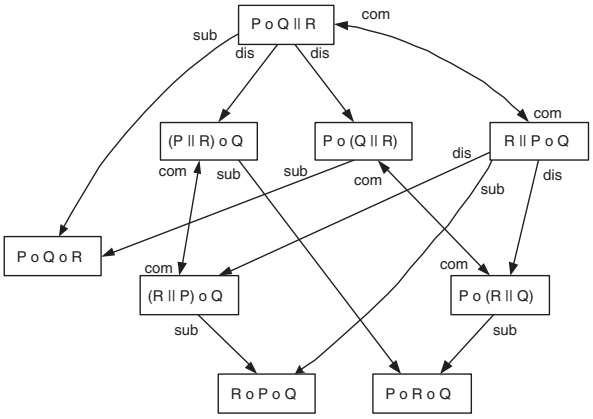
$$f1(A, P) \circ f2(B, P) \Rightarrow f1(A, P) \circ f2(B) \tag{14}$$

3 Enumeration of Functional Topologies

3.1 Enumeration with a Transitive Closure

The method can effectively enumerate possible topology candidates in a systematic manner. Starting with an algebraic expression of a given functional topology, possible topology candidates are generated by applying the axioms outlined in Section 2. With each selected functional topology equation, algebraic laws are applied to create a set of successor topologies. For example, the topology shown in Figure 2 (b) has nine candidates according to the transitive closure as illustrated in Figure 5. The partial order created in the transitive

closure is compatible with the original functional topology. Nine candidates are grouped into six types of topologies as shown in Figure 6 by taking into account the commutative law. Since usage of the split law can lead to an infinite number of candidate nodes, we ignore the law for simplicity.



com: Commutative, asc: Associative, dis: Distributive, sub:Substitutive

Figure 5. Generating Candidates with Transitive Closure

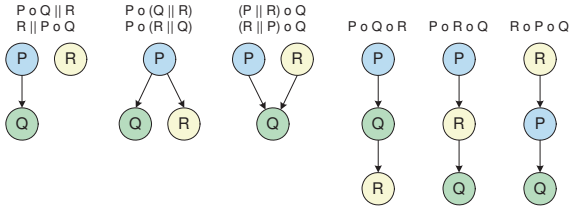


Figure 6. Functional Topology Candidates

3.2 Recurrence Formula and Design Space Exploration

A naive approach of applying the axioms could enumerate a large number of candidate topologies in general. Furthermore, an algebraic expression itself could become complex. One could observe however that in many cases, the enumerated expressions may involve sub-expressions, and thus if the sub-expressions could be commonly represented, the resulting expressions can become less complex. This is especially the case when we consider operations involved in loop structures of a program, which often are the primary target of optimization in multimedia applications. For instance, consider the program in Figure 7 (left). Extracting data dependency gives the functional topology shown in Figure 7 (right), and its algebraic expression is given in Equation (15).

$$P_i \circ Q_i \; (i = 0, \cdots, 7) \tag{15}$$

Instead of applying the axioms directly to these eight expressions as a whole, one could first re-define the expressions recursively so that an expression for a particular loop iteration is given by the terms of that iteration together with an expression of the preceding iterations, and then apply the axioms to that expression. For example, the same functionality can be represented as follows (16):

$$L_n = \begin{cases} P_0 \circ Q_0 \; (n = 0) \\ P_n \circ Q_n \parallel L_{n-1} \; (n = 1, \cdots, 7) \end{cases} \tag{16}$$

When one applies the axioms to L_n, L_{n-1} is treated as a single term, and therefore the number of expressions enumerated in this process is determined by the three terms with the two relations; in this example six expressions are enumerated. However, each appearance of the term L_{n-1} in any of the six expressions can be equivalently replaced by any of the expressions obtained for L_{n-1}, and thus the total space of exploration is exponentially larger than the number of expressions explicitly enumerated by this method.

Note that there are many different recurrent expressions to represent the same functionality. For example, let L_0 be a group of expressions given by $L_0 = \{P_0, Q_0, P_1, Q_1\}$, and L_{2n} denote $L_{2n} = \{P_{2n}, Q_{2n}, P_{2n+1}, Q_{2n+1}, L_{2n-2}\}$. Then one obtains the following (17) as opposed to (16).

$$L_{2n} = \begin{cases} P_0 \circ Q_0 \parallel P_1 \circ Q_1 \; (n = 0) \\ P_{2n} \circ Q_{2n} \parallel P_{2n+1} \circ Q_{2n+1} \parallel L_{2n-2} \\ \quad (n = 1, 2, 3) \end{cases} \tag{17}$$

```
for i=0,..,7
   Pi: pi = a + 1;
end for
for i=0,..,7
   Qi: qi = pi + 1;
end for
```

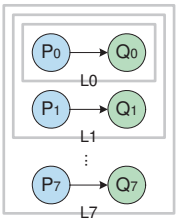


Figure 7. Code Snippet 2 and Corresponding Potential Functional Topology

In general, one can produce recurrence expressions by first defining terms that constitute the expressions and then by applying some rules based on the relations of the terms specified in the original expressions. Examples of such rules are depicted in Figure 8, where the computational blocks A, B and C denote the terms. Figure 8 (a) states that two sequential computational blocks are clustered into a computational block with an expression $A \circ B$, and (b) shows that A and C are clustered into a computational block with an expression $A \parallel C$. Figure 8 (c) is the split law described in Section 2.

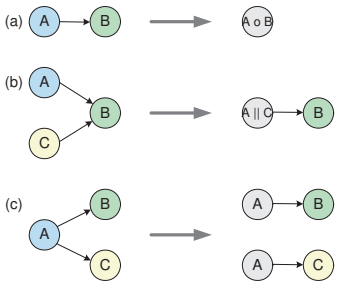


Figure 8. Rules to Produce Recurrence Expressions

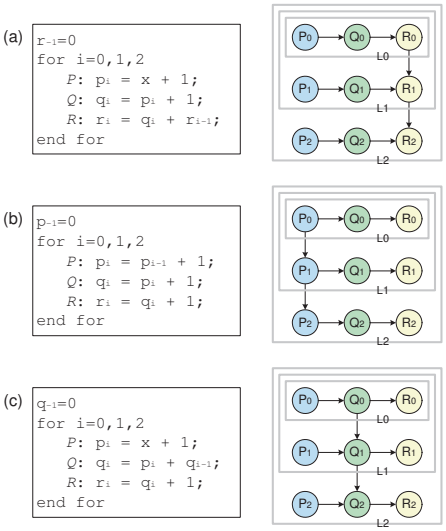


Figure 9. Multimedia Program Structures

For the case of a program shown in Figure 9 (a), we obtain the following by applying these rules:

$$L_n = \begin{cases} P_0 \circ Q_0 \circ R_0 \ (n = 0) \\ (P_n \circ Q_n \parallel L_{n-1}) \circ R_n \ (n = 1, 2) \end{cases} \quad (18)$$

Note that this process of producing recurrence expressions does not employ an assumption that the loop boundary must be statically known. Therefore, the same can be applied for an unbounded loop.

Similarly a recurrence expression for Figure 9 (b) is obtained using the split law as follows.

$$L_n = \begin{cases} M_0 \circ Q_0 \circ R_0 \ (n = 0) \\ M_n \circ Q_n \circ R_n \parallel L_{n-1} \ (n = 1, 2, ..N) \end{cases} \quad (19)$$

$$M_n = \bigodot_{k=0}^{n} P_k \ (n = 0, 1, 2) \quad (20)$$

Finally a recurrence expression for Figure 9 (c) is given by Formula (21) and Formula (22). The split law is used.

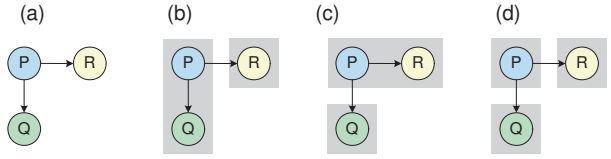


Figure 10. Mapping onto Processes

$$L_n = \begin{cases} M_0 \circ R_0 \ (n = 0) \\ M_n \circ R_n \parallel L_{n-1} \ (n = 1, 2, ..N) \end{cases} \quad (21)$$

$$M_n = \begin{cases} P_0 \circ Q_0 \ (n = 0) \\ (M_{n-1} \parallel P_n) \circ Q_n \ (n = 1, 2, ..N) \end{cases} \quad (22)$$

As this section shows, the number of explicit topology candidates based on the transitive closure depends on the recurrence formula expression. This stems from the granularity of operations represented by the algebraic expressions. The coarser the granularity, the narrower the explored design space is, while the finer the granularity is, the broader the space is.

4 Mapping

4.1 Partitioning Computational Blocks

Once a functional topology has been selected, the next step is mapping it onto computing resources. Mapping determines how computational block functionality will be partitioned onto computing resources (which are ultimately used for implementation). Mapping is based on two rules. First, when a topology has a concurrency relation, the nodes with the relation are partitioned and mapped to separated resources given by the concurrency relation. Second, when the topology has a dependency relation, the nodes with the relation are partitioned and mapped onto either shared or separated resources.

As an example, Figure 10 (a) has three computational blocks P, Q, and R, and two arcs between P and Q and between P and R. The functional topology has dependency relations between P and Q and between P and R, and a concurrency relation between Q and R. Following the rules, (a) is mapped onto two partitions as shown in (b) or (c), and then units with dependency relations are separated into three partitions as shown in (d). A gray zone in the figure illustrates a partition. As a result, three candidates ((b), (c) and (d)) are obtained. Notice that Q and R are **not** allowed in the same partition with these rules.

4.2 Mapping with Recurrence Formula

This section explains how to map a functional topology onto physical resources using the recurrence formula. The

following two relations are needed in order to create functional topologies: dependency relations within a granularity set and dependency relations between two granularity sets. In Section 3.2, the granularity in a recurrence formula is discussed to decide the size of the exploration space. In addition to the recurrence granularity, mapping uses another granularity. This granularity reflects the available resources of the potential implementation platform. For example, we have a recurrence Formula (23) by transforming Formula (16), and provided that the granularity for mapping is P_n and Q_n, a dependency relation in a granularity set is $P_n \circ Q_n$. A dependency relation between two granularity sets is given as $P_n \circ P_{n-1}$.

$$L_n = P_n \circ (Q_n \parallel L_{n-1}) \tag{23}$$

Similarly, when a granularity set is P_n, Q_n, P_{n-1} and Q_{n-1}, a dependency relation in a granularity set is $P_n \circ (Q_n \parallel P_{n-1} \circ Q_{n-1})$, and a relation between two sets is $P_{n-1} \circ P_{n-2}$ given by Formula (24).

$$L_n = P_n \circ (Q_n \parallel P_{n-1} \circ (Q_{n-1} \parallel L_{n-2})) \tag{24}$$

As a next step, the partitioning method explained in Section 4.1 is applied. Consequently, Formula (23) gives candidates of process network as shown in Figure 11 (a), and Formula (24) gives candidates as depicted in Figure 11 (b). A gray zone illustrates a partition. P_e and P_o are P with even and odd indices respectively. A self loop implies a repetition of units in a process.

A drawback of this method is that it is not capable of describing relations between nonadjacent granularity sets. Therefore, this method cannot give such a topology described in (25).

$$\bigodot_{n=0}^{8} P_n \circ \bigodot_{n=0}^{8} Q_n \tag{25}$$

5 Case Study

Functional model exploration of the deblocking filter from H.264 [12], [5] is reported in this section. The objective of the case study is to explore the optimal functional topology to minimize overall execution time.

5.1 Functional Behavior

We chose to explore the H.264 deblocking filter algorithm since it is responsible for a significant percentage **(approx. 33%)** of the total computational complexity of H.264 [6]. The deblocking filter function is applied to a block (4×4 pixels) border of an image for the luminance and chrominance components separately, except for the block borders at the

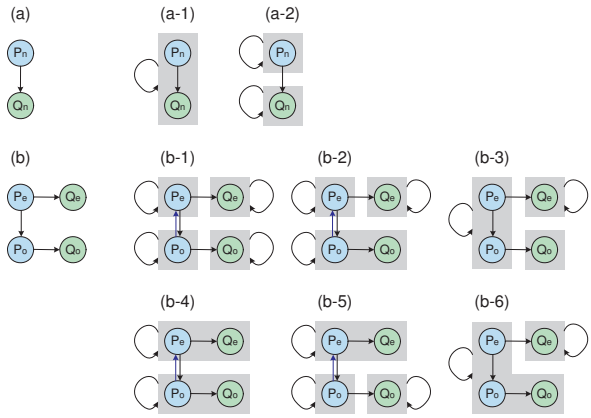


Figure 11. Mapping onto Partitions with Recurrence Formulas

boundary of the image. Note that the deblocking filter function is performed on a macroblock basis after the completion of the image construction function.

In a macroblock, a block border $V0$ is selected first, and then eight pixels denoted as a_i and b_i with $i = 0, \cdots, 3$ are filtered, and the other fifteen rows along $V0$ are also filtered. The filtering is applied to a set of eight samples across a block border (in the order of $V0$, $V1$, $V2$, $V3$, $H0$, $H1$, $H2$, and $H3$ in Figure 12). When $H1$ is selected, eight pixels denoted as c_i and d_i with $i = 0, \cdots, 3$ are filtered as well as the other fifteen pixel set along $H1$.

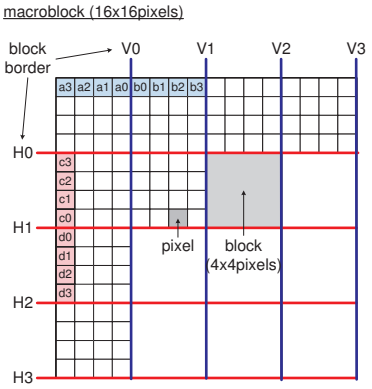


Figure 12. Macroblock Processing

Figure 13 is the pseudo code for the deblocking filter derived from H.264 reference software [12], [10]. $DeblockMB$ checks whether neighbor macroblocks (16 × 16 pixels) are available for a target macroblock. $GetStrength$ outputs a boundary strength ($str_{i,j,k}$) for the filter, and $EdgeLoop$ does filtering for eight samples depending on the strength. The boundary strength is in the range of 0 and 4 (integer number) and the number is decided depending on slice type, reference pictures, the number of reference pictures, and the transform coefficient level of ev-

ery block. Our exploration is carried out for the worst case among five boundary strengths. We observed the total cycle count is the worst (largest) when the boundary filtering strength is one. $GetStrength$ and $EdgeLoop$ will be the units of granularity for our exploration.

```
D : DeblockMB();
for i=0,1 do
   I_i :
   for j=0,···,3 do
      J_j :
      for k=0,···,15 do
         P_k : str_{i,j,k} = GetStrength(i, j, k);
      end for
      for k=0,···,15 do
         Q_k : EdgeLoop(i, j, k, str_{i,j,k});
      end for
   end for
end for
```

Figure 13. Deblocking Filter Pseudo Code

5.2 Functional Exploration

In the code shown in Figure 13, the data dependency is extracted in a similar manner to [9]. Labels D, P and Q indicate $DeblockMB$, $GetStrength$ and $EdgeLoop$ respectively. Labels I and J describe an iteration given by index i and j. Dependency relations appear between $GetStrength$ and $EdgeLoop$ in every iteration of index k, where a boundary strength is sent from $GetStrength$ to $EdgeLoop$. In addition, a dependency relation between two consecutive groups of sixteen $(k = 0, \cdots, 15)$ $EdgeLoop$ computational blocks appears because $EdgeLoop$ reads eight sets of pixel data from a memory and writes eight sets of filtered pixel data to the memory. Hence data written along a block border is supposed to be data read along the next block border. This data dependency can be represented as a graph shown in Figure 14, where the nodes labeled with P_k and Q_k denote the k-th executions of $GetStrength$ and $EdgeLoop$ respectively in the corresponding loops in the code.

This case has four iteration loops with 257 computational

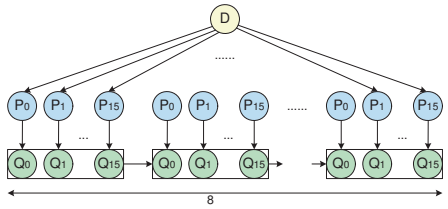


Figure 14. Topology of Deblocking Filter

blocks every macroblock, which results in explosive design space. To reduce the design space, we take a look at dependency relations for every level of nested loop structure. A dependency relation in the deepest level (iteration index k) is expressed algebraically as follows (26).

$$P_k \circ Q_k \ (k = 0, \cdots, 15) \tag{26}$$

A dependency relation for iteration index i and j is given as follows (27) (28).

$$I_0 \circ I_1 \tag{27}$$

$$J_j \circ J_{j+1} \ (j = 0, 1, 2) \tag{28}$$

The most outer loop has the following relation (29).

$$D \circ I_i \ (i = 0, 1) \tag{29}$$

Considering relations (27),(28), and (29), labels D, I and J are executed sequentially. We focus on relations of unit P and unit Q as the case study.

The following recurrence Formula (30) is obtained from relation (26) when the granularity is set as P_n and Q_n.

$$L_n = \begin{cases} P_0 \circ Q_0 \ (n = 0) \\ P_n \circ Q_n \parallel L_{n-1} \ (n = 1, \cdots, 15) \end{cases} \tag{30}$$

Table 1 shows recurrence formulas (second column) when the formula granularity is set as P_n and Q_n, and dependency relations in a granularity set (third column) and dependency relations between granularity sets (fourth column) when the mapping granularity is set P_n, Q_n, P_{n-1} and Q_{n-1}. Recurrence formula ID 1 has no relation between granularity sets.

Table 1. Functional Topology Candidates

ID	Recurrence	Relations in Set	Relations Between Sets
1	$P_n \circ Q_n \parallel L_{n-1}$	$P_n \circ Q_n \parallel P_{n-1} \circ Q_{n-1}$	-
2	$P_n \circ (Q_n \parallel L_{n-1})$	$P_n \circ (Q_n \parallel P_{n-1} \circ Q_{n-1})$	$P_{n-1} \circ P_{n-2}$
3	$(P_n \parallel Q_n) \circ L_{n-1}$	$(P_n \parallel P_{n-1} \circ Q_{n-1}) \circ Q_n$	$Q_{n-2} \circ Q_{n-1}$
4	$P_n \circ Q_n \circ L_{n-1}$	$P_n \circ Q_n \circ P_{n-1} \circ Q_{n-1}$	$Q_{n-1} \circ P_{n-2}$
5	$P_n \circ L_{n-1} \circ Q_n$	$P_n \circ P_{n-1} \circ Q_{n-1} \circ Q_n$	$Q_{n-2} \circ Q_{n-1}$ $P_{n-1} \circ P_{n-2}$
6	$L_{n-1} \circ P_n \circ Q_n$	$P_{n-1} \circ Q_{n-1} \circ P_n \circ Q_n$	$Q_{n-2} \circ P_{n-1}$

5.3 Metropolis Integration and Mapping

To capture this functional behavior we use METROPOLIS [1]. METROPOLIS is a system level design framework based on platform-based design (PBD) [8]. PBD explicitly separates the functional and architectural portions of a system. The METROPOLIS framework evaluates performance by mapping the functional model onto an architectural model for simulation. The architecture models for our flow are based on the Xilinx Virtex II Pro FPGA platform [13] created

in METROPOLIS. Specifically we will be examining architectures based on Microblaze soft-microprocessor cores and Fast Simplex Links (FSLs). An FSL is a FIFO-like communication channel, which connects two Microblazes in a point-to-point manner. Besides their general attractiveness, FPGAs can be characterized a priori thus giving us more confidence in the accuracy of the estimations that guide the optimization process. More details of the architectural modeling are reported in [2] and [4]

Once the functional topology has been created we must transform this into a METROPOLIS functional model. We are interested in investigating potential clock cycle counts when mapped and simulated with an METROPOLIS architecture model. METROPOLIS' higher abstraction level [8] allows functional model statements to be classified into three primitive functions: $read$, $write$, and $execute$. The total number of clock cycles required is found by accumulating cycles for $read$, $write$, and $execute$ functions. $GetStrength$ and $EdgeLoop$ are composed of these basic functions. The arguments to $read$, $write$, and $execute$ are translated by METROPOLIS characterizer databases. This translates into a cycle count for each operation [3]. We refer to a process with $GetStrength$ and $EdgeLoop$ as a filter process henceforth.

The mapping is carried out in such a way that a filter process and a communication channel in the functional model are mapped onto a Microblaze and an FSL in one-to-one manner respectively as shown in Figure 15.

We define a source process in the functional model as follows. A source process is a storage with stream data and baseband data. A source process communicates with filter processes in such a way that a filter process sends 32-bit wide data (a read/write flag, a target address, and a target data length in this order) and afterwords a source process sends or receives data in a burst transfer manner. The source process has in/out ports connected to all filter processes as shown in Figure 15 and receives requests from the filter processes in a first-come-first-served basis with non-blocking reads.

The source process is also mapped onto a Microblaze. The length of a FIFO connected between the source process and filter processes is large enough so that processes are not blocked on write operations. For this study, the source FIFOs have a depth of 16. The length of a FIFO between filter processes changes in this case study, and is given by N in Figure 15.

Provided that the number of MicroBlazes is three and under, 14 functional topology candidates are obtained from Table 1 as shown in Figure 16. A gray zone represents what will be executed on each Microblaze (a partition; mapping) and resource ID is denoted by PID in the figure. For example, (C) in Figure 16 implies that resource 1 (PID1) has computational block P and resource 2 (PID2) has computational block Q. ID 1 from Table 1 gives candidates (A), (B), (C) and (J). We skip sequential structured topologies 4, 5 and 6 from Table 1 to simplify.

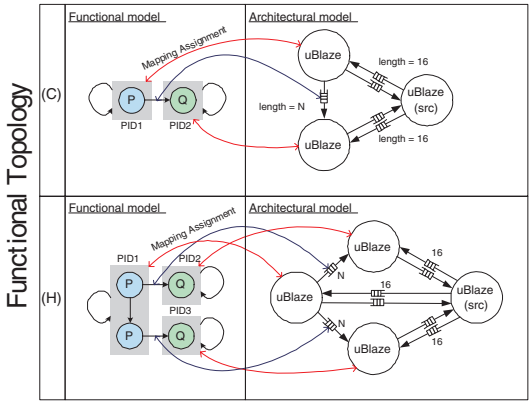


Figure 15. Mapping a Functional Model onto an Architectural Model

5.4 Results of Exploration

Execution cycle count results for the functional topology candidates explored (Figure 16) are discussed in this section first. Figure 17 shows the total execution cycle count breakdown (computation cycles, communication cycles with a source process, and waiting cycles) when the length of a FIFO between filter processes (N denoted in Figure 15) is one. The waiting cycles accumulate in two following cases: when a filter process waits for other filter processes to finish their transaction with a source process and when a filter process waits for data to come to a FIFO from other filter process.

The vertical axis in Figure 17 is the number of clock cycles required and horizontal axis shows topologies (A to N) as shown in Figure 16. B through G have two bars, where the first bar corresponds to process 1 denoted by $PID1$ in Figure 16 and the second is process 2 denoted by $PID2$. H to N have three bars, where the first bar corresponds to process 1 denoted by $PID1$ and the second and third bars are results of process 2 and process 3 denoted by $PID2$ and $PID3$ each.

The simulation results demonstrate that workload balance has a strong effect on execution time for a multiprocessor system. Case H is the best case in terms of workload balance and as a result, the total amount of cycles is the smallest. Compare case J with case L. L has more communication channels than case J. Nonetheless process 3 in L spends less time waiting than process 3 in J, which implies that the memory traffic of L is lighter than that of J due to synchronization between process 1 and process 3. Compare K with L. Their topologies are the same, but the process execution order differs. As a result, the completion times are different. Similar conclusions can be drawn for M and N.

We can draw several conclusions from these results.First, apparently small changes in the functional topology can actually have dramatic effects on execution time. Secondly, the

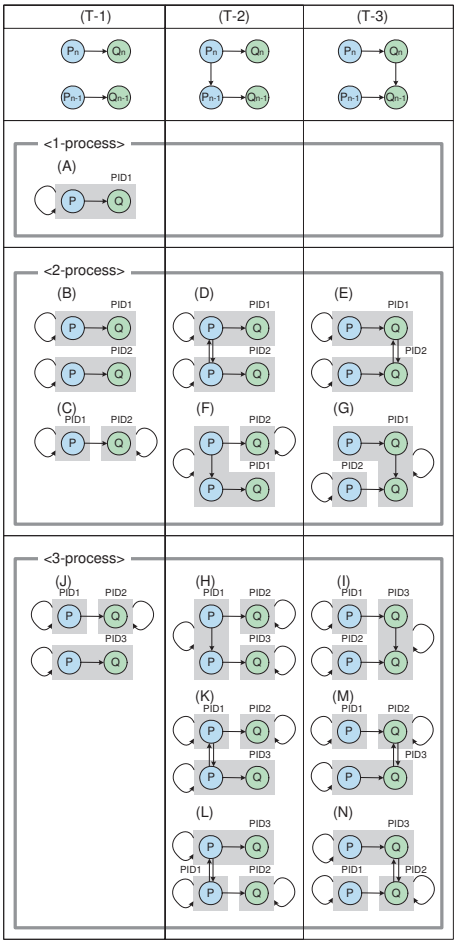


Figure 16. Candidates of Functional Topology Mappings

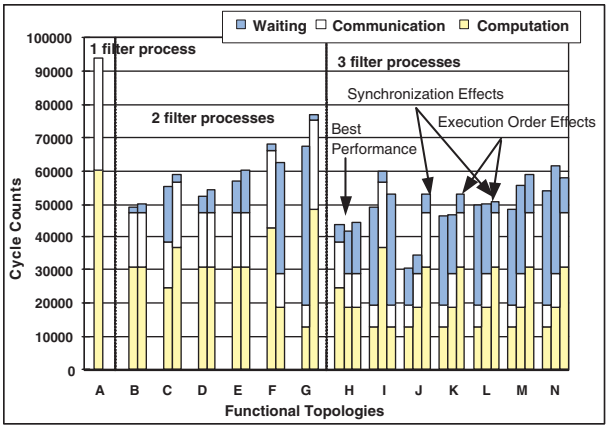


Figure 17. Metropolis Simulation Results for All Candidates

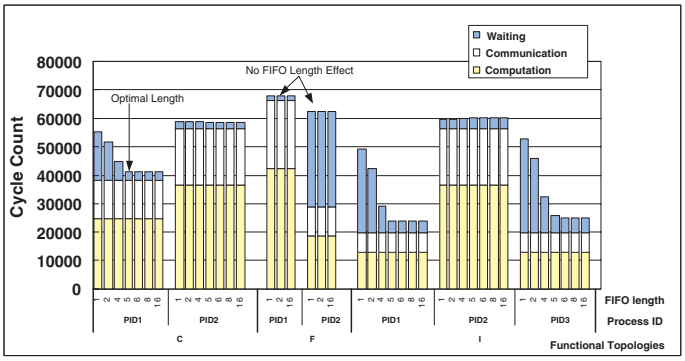


Figure 18. Metropolis Simulation Results for Various FIFO Sizes

breakdown of overall execution time is important to examine for these types of applications. Finally, METROPOLIS was able to perform efficient functional design space exploration with ease and with only minor changes to the functional and mapping models.

5.4.1 Optimal FIFO Size

In the second set of experiments we took a look at the effect of FIFO length. Figure 18 shows execution cycle counts of three cases: C, F and I when the length of a FIFO between filter processes changes (N in Figure 15). The results show the optimal length in terms of minimum cycle counts. Changing the length of a FIFO does not have an effect on the total cycle count, but rather on the cycle counts of individual processes.

Table 2 breaks down the performance further. Total clock cycle counts (second column) and the optimal FIFO length, which is the smallest with the lowest clock cycle counts (third column), and resource cost are shown. Resource cost is program binary code size (4th, 5th and 6th column for each process) and PQ is the result given by combining P and Q computational blocks. In the case where FIFO length does not make any difference for the counts, the optimal length is set to 1.

This simulation demonstrates that users can make a decision regarding the optimal functional model based on parameters related to performance and cost such as total cycle counts (workload balance), communication overhead, memory traffic, FIFO length, shared memory size, the number of processors, program code size, context switching overhead, register, cache, dedicated hardware logic size, and so forth. Again, METROPOLIS provides a easy-to-use framework for this type of functional exploration.

5.4.2 Simulation Accuracy

All of the previous results are meaningless unless METROPOLIS simulation accurately correlates to the actual implementation. Figure 19 illustrates how closely METROPOLIS' simulation compares to experimental results.

Table 2. Performance and Cost Results

Topology	Counts	Length	Proc1	Proc2	Proc3
A	94021	1	PQ	-	-
B	50188	1	PQ	PQ	-
C	58839	5	P	Q	-
D	54505	1	PQ	PQ	-
E	60124	1	PQ	PQ	-
F	67981	1	PQ	Q	-
G	76182	6	PQ	P	-
H	43932	1	P	Q	Q
I	60215	5	P	Q	P
J	52031	3	P	Q	PQ
K	52971	1	P	Q	PQ
L	50780	1	P	Q	PQ
M	58941	6	P	Q	PQ
N	61190	6	P	Q	PQ

Binary Data Size PQ: 47.9KB; P: 47.0KB; Q: 45.9KB

Each design was implemented on a Xilinx ML310 design board and the execution time was measured. The maximum difference between implementation and simulation is **7.3%**. This is a high correlation while maintaining a high level of abstraction in the Metropolis models. In addition, it **confirms that H has the lowest cycle count of any design** and demonstrates that making an absolute design decision based on Metropolis simulation would have been the correct choice.

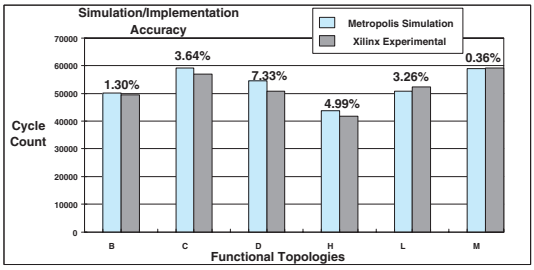


Figure 19. Metropolis Accuracy versus FPGA Implementation

6 Conclusions

This paper presented a methodology for performing functional design exploration with the goal of obtaining an efficient implementation on a given platform that was appropriately characterized in terms of performance. Beginning with an operational description of the function to be implemented, we use algebraic operators to obtain an efficient representation that is then manipulated using composition and decomposition laws to generate alternatives quickly. These alternatives are correct by construction as they are all functionally equivalent to the original description.

This procedure can be used to yield a method to transform a sequentially structured program in a multimedia application domain into a set of concurrent processes. Once this decomposition had been carried out, we proposed a mapping methodology to partition this design and assign those partitions to architectural resources. Finally, this entire process was demonstrated in the METROPOLIS design environment. The results of our H.264 deblocking filter example showed that not only do small changes in the functional topology result in nonintuitive, dramatic results, but also that with METROPOLIS, we are able to explore a number of design axis with a very high accuracy.

Balancing the structure of an application with the structure of an implementation architecture will be increasingly important as both functionality and available architectures grow in complexity. Using solid methods to explore the design space within the limits posed by the implementation platform, will give a strong push towards high-value system level design tools.

References

[1] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An Integrated Electronic System Design Environment. *IEEE Computer*, 36(4):45– 52, Apr. 2003.

[2] D. Densmore. Metropolis Architecture Refinement Styles and Methodology. Technical Report UCB/ERL M04/36, University of California, Berkeley, CA 94720, September 14, 2004.

[3] D. Densmore, A. Donlin, and A. L. Sangiovanni-Vincentelli. FPGA Architecture Characterization for System Level Performance Analysis. In *DATE*, March 2006.

[4] D. Densmore, S. Rekhi, and A. L. Sangiovanni-Vincentelli. Microarchitecture Development via Metropolis Successive Platform Refinement. In *DATE*, pages 346–351, 2004.

[5] Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification. (ITU-T Rec. H.264 — ISO/IEC 14496-10 AVC).

[6] M. Horowitz, A. Joch, F. Kossentini, and A. Hallapuro. H.264/AVC Baseline Profile Decoder Complexity Analysis. *IEEE Trans. Circuits Syst. Video Techn*, 13(7):704–716, 2003.

[7] A. Jantsch. *Modeling Embedded Systems and SOC's*. Morgan Kaufmann Publishers, 2004.

[8] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System Level Design: Orthogonolization of Concerns and Platform-Based Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12), December 2000.

[9] B. Kienhuis. Matparser: An Array Dataflow Analysis Compiler. Technical Report UCB/ERL M00/9, University of California, Berkeley, CA 94720, April 27th, 2000.

[10] MPEG4 AVC Reference Software JM92. http://www.m4if.org/index.php, MPEG Industry Forum.

[11] J. A. Webb. Steps Toward Architecture-Independent Image Processing. *IEEE Computer*, 25(2):21–31, 1992.

[12] T. Wiegand, G. J. Sullivan, G. Bjntegaard, and A. Luthra. Overview of the H.264/AVC Video Coding Standard. *IEEE Trans. Circuits Syst. Video Techn*, 13(7):560–576, 2003.

[13] Xilinx. http://www.xilinx.com.