A Transformational Codesign Methodology

Tommy King-Yin Cheung, Graham Hellestrand and Prasert Kanthamanon

VLSI and Systems Technology Laboratory
School of Computer Science and Engineering
University of New South Wales
Kensington 2052 Australia

Abstract —We present a hardware/software codesign methodology using formal transformations. The goal is to refine a given function specification of a task to an operational structure involving both hardware and software components. The refinement process is separated into two levels, the algorithmic and the structural. Within each level, refinement is accomplished by applying sequences of transformations that preserve the functionality of an initial specification. This allows various 'correct' design alternatives to be generated and their costs analyzed. At the algorithmic level, different algorithm designs are explored, each producing a computational schedule that has a different performance cost. At the structural level, different spatial structures with different resources and performance costs are explored. These costs which characterize the designs are used to assist in the hardware/software partitioning. An example is used throughout to illustrate this methodology.

I. INTRODUCTION

The goal of hardware/software codesign is to synthesize efficient implementations consisting of mixed hardware/ software components from initial function specifications. A function specification is a description of an input/output relation, which is 'abstract' in the sense of being independent of any specific implementation or partitioning. The central idea behind our codesign process is to incrementally refine the high-level function specification until an implementation is derived which is as a mix of procedural software processes and applicative hardware modules. The process involves the exploration of hardware/software tradeoffs such as the optimization of the hardware/software interface or the movement of processing functions from one domain to the other. The process is based on a transformational synthesis approach which constructs the implementation by repeatedly applying a set of correctness-preserving transformation rules. The transformation is said to preserve correctness if the resulting implementation is functionally equivalent to its initial specification [1]. This transformational approach has practical as well as theoretical advantages in codesign. The practical advantage is that applying different sequences of correct transformations can generate alternative designs for performance and cost-benefit analysis. The theoretical advantage is that verification of the resulting design is achieved by the application of a sequence of correct transformation steps. That is, since each transformation step preserves the correctness of the original function, the resulting design is guaranteed to be correct. Intuitively, the transformation steps can be viewed as the primitive operations of a more general design-synthesis process, and hence many design automation techniques can be applied to automate the transformation steps. In this paper, the transformations are applied at two different levels within the codesign process: the development of a high-level algorithms and the optimization of low level structural implementations [2]. In practice, these two levels of transformation are coupled together and applied iteratively. The objective is to integrate the process of algorithm design, early on, with lower level structural design. This allows a broader design space to be explored than the traditional high level synthesis methodology which typically assumes a fixed algorithmic specification of the problem [3,4,5].

The specification is written in a single high level functional notation, called *form* [6,7,8] which provides for codesign a unified system specification device. The notation is aimed at describing various forms of control compositions and synchronisations over a set of functions and forms. *Form* has three key features. First, it has a semantic model [9] that captures the causal ordering of function applications on a stream of data possibly in a multidimensional structure [6]. The notion of continuity in streams [10,11] provides a temporal abstraction of data. The resulting model specifies a reactive behavior of a system which relates outputs to inputs in time. Second, parametrised function or form can be used as a *type* which defines a context to be instantiated by other functions [7]. Composite types can be built using the set of composition laws for functions. The binding of composite function types is particularly important for building systems hierarchically. Third, it supports syntactic transformation and verification [1]. The transformation mechanisms themselves are encoded using the notation. This supports a formal validation of both the design and its transformation mechanisms, thus supporting the concept of correct-by-construction in our approach.

Another important component in a codesign system is a measure of design cost. Given a basic function specification, several transformations which lead to different correct hardware/software implementations may be applied. Each of the resulting implementations usually has a different design cost. A major task is to select an implementation that best

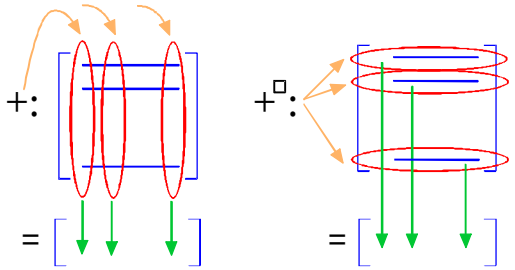
ASP-DAC '97
0-89791-851-7/$5.00 ©1997 IEEE

fits the cost requirements. We have established a system performance model in terms of various design metrics to be used to estimate the cost and to determine a feasible solution that satisfies the requirements. Some of the design metrics include software code size, throughput of processors, memory size, access time, interrupt cycle time, costs of function units, and computation time etc.

This paper is organized as follows. Section 2 gives a brief introduction to the *form* notation. Section 3 defines the two levels of transformations, algorithmic level and structural level, and their correctness properties. Section 4 uses a case study to illustrate the transformational codesign process, including specification, algorithm derivation, hardware/software partitioning, structural refinements and structure mappings. Section 5 concludes the work.

II. The Form Notation

The *form* is based on a variant of FP [12], with extensions to support multi-dimensional structured streams, delay functional and synchronized concurrent forms [6,13]. The main advantage of FP is its combinative property that allows function composition and construction by means of combining forms. The primary data structure in *form* is the *stream* which is used to model the time-ordered flow of continuous data during function evaluation. However, the data structuring of each stream element is operational, that is, its data structure is represented by the composition and decomposition operators used to construct them. Function application on a multi-dimensional data structure is distributive and concurrent over the elements in the default *least significant dimension*, for example, applying a *sum* operator to a two dimensional structure will sum its columns (the least significant dimension) concurrently to produce a one dimensional structure (Fig. 1a). A higher order operator could be defined to alter the default operating dimension of a function, for example, a multidimensional map operator (denoted by $f^{\square}$) which has been defined in [2] operates on the most significant dimension. As an example, the application of a map-sequence-sum, $+^{\square}$ operator to a two dimensional structure is illustrated in Fig. 1b.



(a) (b)

Fig. 1. The applications of the *sum* and *map-sequence-sum* operators to a two dimensional data structure.

The set of primitive combinators include serial composition (denoted by $f_1 \cdot f_2$), conditional composition (denoted by $(f_1,f_2)?p$), concurrent composition (denoted by $[f_1,f_2]$) and delay functional (denoted by Zf) which is used to construct loop/feedback structures [14]. There are some other primitive functions, such as the identity function (id) and selectors (α and ω – where α selects the head of a sequence, ω selects the last element of a sequence and $(\alpha+i)$ selects the $(i+1)$th element of a sequence in a finite multi-dimensional structure). The intuitive meaning of each construct is to follow.

Serial composition $f_1 \cdot f_2$ means that the result of applying f_2 to a stream is a stream which is passed to f_1, where both functions operate at a rate correlated to the input stream. Conditional composition $(f_1,f_2)?p$ means that for each data stream, the predicate p is computed first, then either f_1 or f_2 is applied depending on the truth value of p. Concurrent composition $[f_1,f_2]$ means that the input stream is passed to both functions and the applications are performed concurrently. The resulting structured stream is then passed out at a rate correlated to the input stream. Thus, it induces synchronization on concurrent activities, even though they may have vastly different execution time. Finally, the delay functional Zf returns the previous stream element of the result of f applied to the original input stream, where the initial output function is denoted by $f@[0]$. The default value of Zf is # (an undefined function) which always returns the undefined object $\perp$.

Form also supports named streams and treats them as first class objects which can be referenced or passed as arguments to forms. Technically, multiple named streams can be grouped into one sequence to be passed as a single argument to a form, where the argument is then decomposed to the streams using the selector functions. Thus, there is already the possibility of representing sequences by means of composition and decomposition operators, so that naming is simply a convenience in the notation. However, associating names with streams does improve appreciably the readability of the notation.

As an example, here is a simple counter that counts the number of clicks between a ***start*** and a ***stop*** signal (two named input streams) for a mouse-like device.

Pulse *start stop* :: (0, (1, ZPulse)?*start*)?*stop*.
ClickCounter *click start stop*
:: ((inc·P, P)?*click*, (P, 0)?*stop*)?(Pulse *start stop*)
where P = ZClickCounter.

The function Pulse produces a period of ones between the ***start*** and the following ***stop*** signals and the function ClickCounter which makes use of an incrementer (inc) counts the number of ***click***s during that period.

III. Multi-Level Transformations and Correctness

Form describes systems as a set of continuous stream processing functions which are possibly recursive. The output of the system for a given input stream is determined by the least fixpoint of the functions [10,15]. This input-output relation defines the functionality of the system. Generally, for a given specification function *f*, there exists many possible implementation functions that have the same functionality as *f*, but they differ by their ways of structuring the input-output data

stream [1]. Each implementation function could be obtained from f through a series of correct transformations that preserve the functionality of f. If g is a correct transformation of f, then the following equation holds:

$$\varphi_O \cdot f = g \cdot \varphi_I$$

where the functions φ_I and φ_O are structuring functions that map the structures of all objects in the domain of f into their corresponding object structures in the domain of g.

Despite the generality of this property, it is only suitable for the refinement of functions at the algorithmic level where the structuring of the input-output data is not fixed yet. At the structural level, spatial refinement is generally needed to explore the variety of spatial structures appropriate to realize a given implementation function. Each realization could be obtained from the implementation function through a series of spatial transformations that preserve not only the functionality of the function, but also the structuring of its input-output data. As an example, a pipelined realization is structurally different to a non-pipeline one, but they both exhibit the same input-output structuring of data and have the same functionality. Some spatial transformations include grouping common conditional expressions, pipelining, serializing components in a concurrent form, and many of the simple algebraic laws [1]. By integrating both functional and spatial refinements during the design process, derivation of input-output behavior and optimization of structure yielding the best design under some constraints, become possible.

IV. Transformational Codesign

The codesign process proceeds from the top-down as shown in Figure 2. It starts with a *form* specification of a task and constructs an algorithm that computes the desired function. The construction of algorithms requires analysis of the structure of the specification that matches a desired class of computational strategy, such as divide-and-conquer or dynamic programming etc, and to direct appropriate transformation mechanisms to derive an algorithm using the computational strategy [16]. The resulting implementation function is to be partitioned into hardware and software parts which satisfy the cost constraints. During the partitioning process, structural exploration of the hardware and software components is carried out iteratively to obtain optimistic and achievable cost estimates of the overall system. A performance model of the system which assumes a fixed architecture is used to evaluate this cost. The final structural forms are then mapped into the target implementation technologies to produce high level pseudo-code for the software components and applicative descriptions for the digital hardware components. The rest of this section illustrates this codesign process with an example.

A. An Example of the Codesign Process

In this case study, the objective is to codesign a system for computing the two dimensional fourier transform operation (2DFT). The target system has three components:

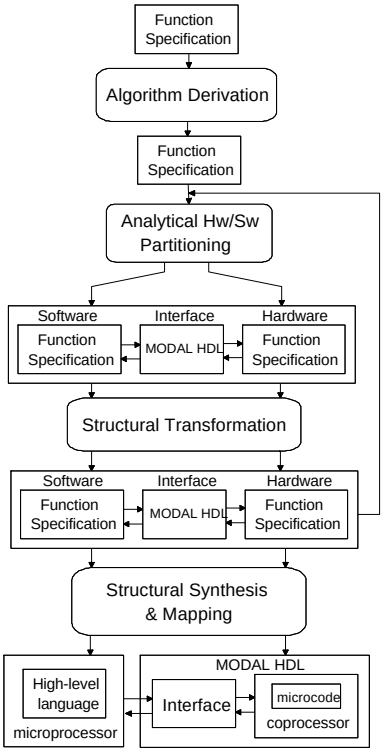


Fig. 2: The transformational codesign process

(1) a processor core for interpreting software processes, (2) a coprocessor which contains many parallel function units for processing the partitioned data at a high speed and (3) a buffered interface unit for synchronizing the interaction between the software processes and the hardware coprocessor.

The structure and effect of the three components is described below:

Processor Core: There are two software processes running on the processor core, one for the partitioning of input data and arranging it in sequence into the data buffer, and the other for the loading of appropriate instructions into the control buffer, where the instructions in the buffer are ready to be executed by the coprocessor.

Buffered Interface Unit: It contains three FIFO buffers, a control buffer, and an input and an output data buffer. When the control buffer has been loaded, it starts a coprocessor execution cycle by sending it an instruction. When the buffer becomes empty, it signals an interrupt to the processor core which will then suspend its current process and invoke another one that fills up the buffer.

Coprocessor: The coprocessor begins its instruction execution when the processor core has filled up both the input and the control buffers. It stops when the buffer becomes empty and resumes when the buffer has been refilled. When the coprocessor completes its computation, it sends the data to the output buffer which will be periodically read by the processor core.

The overall system structure is shown in Fig 3. Details of the design steps that lead to an implementation of the 2DFT operation are described next.

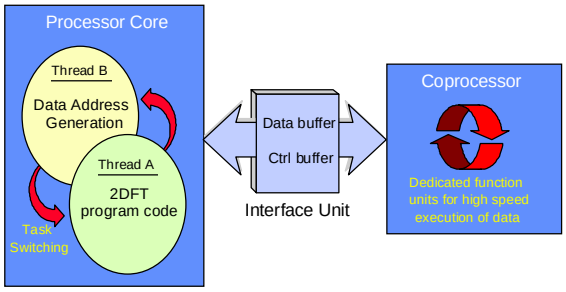


Fig. 3: The embedded system structure for computing the 2DFT operation

Now the specification, algorithm derivation, partitioning, transformation and synthesis activities are delineated.

Specification: The 2DFT operation of an input matrix $x(n_1, n_2)$ is expressed in the *form* notation as shown below.

$$X_{k_1k_2} = \sum_{n_1=0}^{N-1}\sum_{n_2=0}^{N-1} x_{n_1n_2} W_N^{n_1k_1} W_N^{n_2k_2}$$

mathematical formulation of 2DFT

$X_{k_1k_2} :: +\cdot +^{\square}\cdot (\times^{\square}\ x_N^{(2)}\ W_N^{(2)})$

where the 2D input sequence $x_{n_1n_2}$, $\forall\ 0 \le n_1, n_2 \le (N-1)$, is expressed as:

$x_N^{(2)} = [\,[x_{00},\ldots,x_{0(N-1)}],\ldots,[x_{(N-1)0},\ldots,x_{(N-1)(N-1)}]\,]$

and the coefficient matrix as:

$W_N^{(2)} = [\,[W_N^{\,0},\ldots,W_N^{(N-1)k_2}],\ldots,[W_N^{(N-1)k_1},\ldots,W_N^{(N-1)k_2+(N-1)k_1}]\,]$

The *form* description of 2DFT

The above *form* description of the 2DFT operation is formulated in terms of a multidimensional map operator which is defined in the box below.

Multidimensional Map operator $f^{\square}$:

$f^{\square}\ A_n^{(m)}\ B_n^{(m)} :: [\,(f^{\square}\ \alpha A_n^{(m)}\ \alpha B_n^{(m)}), \ldots, (f^{\square}\ \omega A_n^{(m)}\ \omega B_n^{(m)})\,]$, $m > 1$

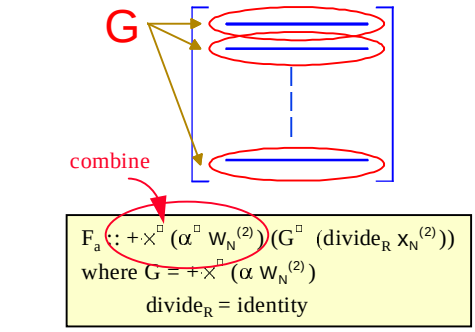
$f^{\square}\ A_n^{(1)}\ B_n^{(1)} :: [\,f^{\square}\ \alpha A_n^{(1)}\ \alpha B_n^{(1)}, f^{\square}\ (\alpha+1\ldots\omega) A_n^{(1)}\ (\alpha+1\ldots\omega) B_n^{(1)}\,]$, $m \le 1$

where $A_n^{(m)}$, $B_n^{(m)}$ denote a m-dimensional block with n elements in each dimension. The function $f^{\square}$ is defined recursively by applying itself to each element in the most significant dimension. Note that the elements in $[\alpha A_n^{(m)}, \ldots, \omega A_n^{(m)}]$ have one smaller dimension than $A_n^{(m)}$, so that this recursive definition terminates. The base case (second line) states that f is applied to each element in a single-dimensional sequence.

Algorithm Derivation: Our method of algorithm derivation is by successive modifications to the initial *form* specification using a set of transformation rules that satisfy the commuting property. The application of transformation rules is directional, that is, an output scheme is provided in the derivation as part of the design strategy. For instance, the chosen strategy for designing the 2DFT operation is *divide-and-combine* scheme which can be expressed in a general form:

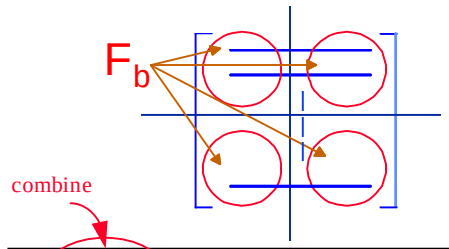
$$F :: \text{combine} \cdot G^{\square} \cdot \text{divide}$$

where the function divide partitions the input data into subsets that are processed independently by the function G, and the results are then combined using the combine function. There are many possible derivations of the functions divide, combine and G, for an initial function specification. The



$F_a :: +\cdot\times^{\square}\ (\alpha^{\square}\ W_N^{(2)})\ (G^{\square}\ (\text{divide}_R\ x_N^{(2)}))$
where $G = +\cdot\times^{\square}\ (\alpha\ W_N^{(2)})$
divide_R = identity

(a)



$F_b :: +\cdot\times^{\square}\ W_4^{(1)}\ (F_b^{\square}\ (\text{divide}_Q\ x_N^{(2)})\ W_4^{(3)})$
where $W_4^{(1)} = [\omega_N^{\,0}, \omega_N^{(m/2)k_2}, \omega_N^{(m/2)k_1}, \omega_N^{(m/2)k_2+(m/2)k_1}]$
$W_4^{(3)} = [\alpha_q,\ldots,\alpha_q]\ W_N^{(2)}$
$\text{divide}_Q = [\alpha_q,\ldots,\omega_q]$
and α_q, ω_q are quadrant selectors that select all elements of the first and last quadrants respectively in a 2D-sequence.

(b)

Fig. 4. Two possible implementation functions of the 2DFT specification obtained by partitioning the 2D input sequence into (a) rows and (b) quadrants.

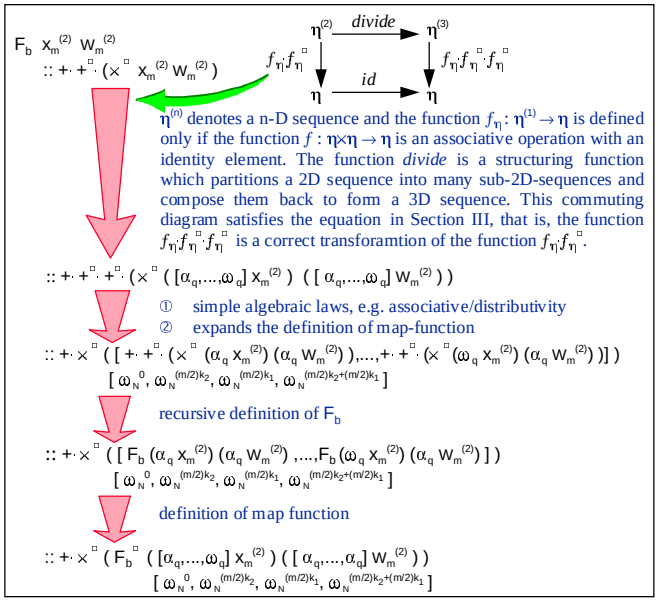


Fig. 5: Derivation steps of the function F_b from the 2DFT specification

selection is dependent on the final cost metrics defined as part of the performance analysis of the implementations. Figure 4 illustrates two possible derivations from the 2DFT specification, and Figure 5 details the sequence of derivation steps for the implementation function, F_b in Fig. 4b.

Hardware/Software Partitioning: Given a divide-and-combine implementation function, various hardware/software partitions can be created by unfolding the function into an evaluation tree whose leaf nodes represent the *least* task size supported by the hardware module. A depth-first-traversal of this tree produces a *sequential schedule* of software instructions for the hardware module. The size of each hardware/software partition is governed by the granularity of the hardware module. The larger the grain size, the smaller will be the level of interaction between hardware and software, and the greater the concurrency, the higher the performance the resulting system will have. However, a larger grain size also means a higher hardware cost. Figure 6 shows the evaluation trees for the two derived implementation functions, F_a and F_b (Fig. 4) of the 2DFT specification and their possible grain sizes. By varying the grain sizes, different hardware/software partitions are generated (see Fig. 6). To assist in the performance evaluation of each partition, many design metrics were considered. The set of metrics includes:

t_p =	processor data production rate (cycles/data)
t_c =	co-processor data consumption rate (cycles/data)
t_{comp} =	the coprocessor processing time.
t_{idle} =	the time period in which the coprocessor is computing, but not taking any data stored in the input buffer.
t_{int} =	interrupt cycle time (interrupt latency + service time)
t_i =	memory transfer time (cycles/data) for writing an instruction into the control buffer (FIFO).
S =	input data buffer (FIFO) size
Q =	control buffer size
l =	the coprocessor grain size.
N =	square root of the total data size
code size =	the number of software instructions sent from the processor core to the coprocessor

This takes into account the sizes of data and control buffers at the interface unit which is critical to the overall performance of the implementation function. To enable the coprocessor to consume its input data *continuously* (i.e. data is *always* present in the buffer for the coprocessor), the total time needed to produce all the data from the processor core must be less than the total time taken for the coprocessor to consume it. With this requirement, an inequality for the size of the input buffer is established (proof omitted):

$$S > \frac{-1}{t_p}\left((t_c - t_p)N^2 - \left(\frac{\text{codesize}}{Q} - 1 \right) \times \min(t_{int}, t_{comp}) - t_{idle} \right)$$

If S is below the lower bound determined by the inequality, the potential parallelism between the processor core and the coprocessor will be reduced as the coprocessor has to stop and wait for the processor core to fill up the buffers. Thus, the best choice for S is its lower bound value plus one. Similarly, the best choice for the control buffer size, Q is the code size. If Q < codesize, an interrupt will be raised to the processor core when the buffer becomes empty; the processor core then invokes a process (an interrupt routine) to fill up the control buffer. Thus, the interrupt cycle time t_{int} which is the time needed to service an interrupt is directly proportional to the control buffer size Q.

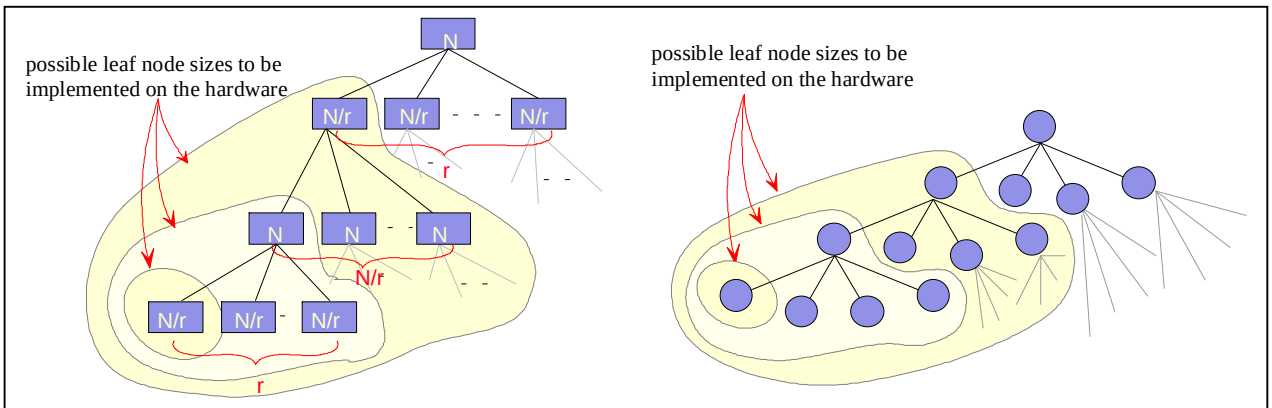
The calculation of the overall execution time, t_{exec} follows:

t_{exec} = the time needed to initialize the control and data buffers, t_{init} + total coprocessor execution time, t_{cop} + total interrupt service time, t_{ser}

where $t_{init} = t_i \times Q + t_p \times S$

$t_{cop} = \text{codesize} \times t_{comp}$

$t_{ser} = t_{int} \times (\text{number of interrupts})$

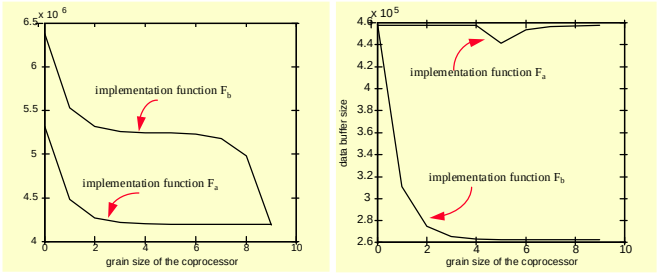


The divide-and-combine function, F_a of Fig 4a can be expanded into a two or three level tree where the leaf nodes represents the size of the hardware modules. The above diagram shows a three level tree. The bottom two levels represent the processing of a block of (N/r) rows with each row containing N elements. Each row is further subdivided into r sections of size (N/r) each. The first level represents the processing of a column which is subdivided into r sections with each one containing (N/r) rows of elements. Different sizes of hardware/software partitions can be generated by varying r and number of levels in the tree.

The recursive divide-and-combine function, F_b of Fig 4b can be expanded into a four way rooted balanced tree with the recursion depth indicating the *least* task size supported by the hardware module. Each level represents the processing of a quadrant which is obtained by sub-dividing the quadrant at the level above. That is, if the root node processes an (N×N) input sequences, then each of its child nodes at the first level will process an (N/2 ×N/2) input sequence and so on. Generally, each node at the rth level processes an ($N/2^r \times N/2^r$) input sequence, and by varying r, different sizes of hardware/software partitions are generated.

Fig. 6. Two possible expansions of the evaluation trees for the two divide-and-combine implementation functions in Fig. 4.

To determine a suitable hardware/software partition, given the performance constraint, the execution time t_{exec} must be calculated for each possible partitioning size, and is defined in terms of the granularity of the hardware module. As for the example implementation functions F_a and F_b, each of their relationships between the execution time (in cycles) and the grain sizes of the coprocessor is shown in Figure 7a. The calculation assumes the best choices for both S and Q, thus ignoring interrupts. The grain size is a function of the data partition size, r, which is defined in Figure 6. As the grain size increases, more functions are migrated from software to hardware and the execution time decreases as expected. Note that the implementation function F_a has a lower time cost than that of F_b, but it uses a much larger buffer size (see Figure 7b) than F_b does, in order to sustain its continuous consumption of data without interruption. However, if Q is set less than the code size (Q < codesize), the effect of interrupts must be taken into account in the performance calculation. Figure 8 shows the effect of incurring various numbers of interrupts on the execution time curve. Initially, when the grain size is small, the interrupt cycle time is larger than the coprocessor computation time. This causes the coprocessor to stop and wait for the buffers to be refilled, and hence increases the overall execution time. As the grain size increases, the coprocessor computation time dominates the interrupt cycle time. This means that the control buffer has been filled up long before the coprocessor completes its current instruction (the one that causes the interrupt), allowing the coprocessor to continue its next instruction without stopping. Thus, the interrupts have no effect on the execution time. This type of analysis permits the examination of various hardware/software partitions and enables a suitable selection to be made which observes some desired cost constraint.



(a) (b)

Fig. 7. (a) The total execution time, t_{exec} and (b) the buffer size, S of the two implementation functions F_a and F_b as a function of the hardware grain size.

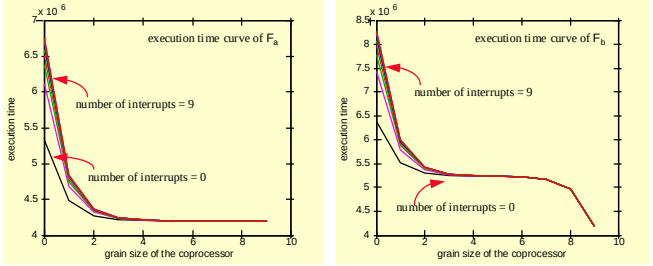


Figure 8: The relationship between the number of interrupts and the execution time of the two implementation functions F_a and F_b.

Structural Transformation: Different structural forms usually have different performance costs. The process of structural transformation refines a given structural form into one that has lower cost and higher performance. This refinement process is coupled with the hardware/software partitioning, because system performance generally varies with the size and structure of the hardware module. For each hardware/software partition, there are many possible structural implementations of the hardware module that have the same functionality and input-output structuring of data, but different performance costs. They can be generated by successive spatial transformations of a given implementation function. The resulting function is guaranteed by the transformations to preserve its structuring of the input-output data and the original functionality. Details of the transformation mechanisms can be found in [1].

As an example, consider a hardware/software partition of the function, F_b that has the smallest grain size, that is, the hardware module computes four complex multiplications followed by a complex add-accumulation for each instruction cycle. The structure of the hardware module is shown in Figure 9a, and it accepts four pairs of input sequences and produces one output sequence. This structure has the same input-output structuring of data as the structure shown in Figure 9b, where the four complex multipliers are connected to a shared bus and data is fed to the accumulator sequentially. Although the latter structure requires a smaller accumulator unit, it takes more clock cycles to complete the whole computation. By applying different spatial transformations to a form, many different structures could be explored, allowing design tradeoffs and selections.

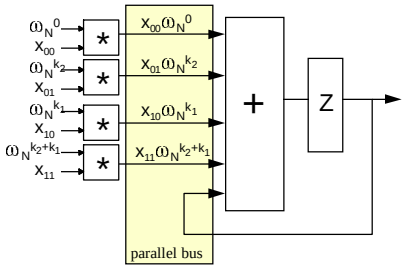


Fig. 9a : A parallel bus implementation of a product-accumulator, where data is fed to the accumulator in parallel.

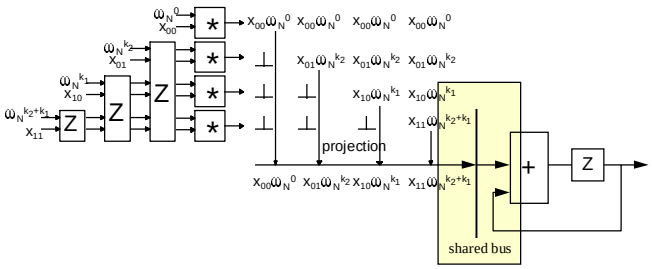


Figure 9b: A shared bus implementation of a product-accumulator, where data is fed to the accumulator sequentially.

Structural Synthesis and Mapping: This step maps the partitioned structural forms into physical components determined by some given implementation technology. The physical components could be software codes in a specific

executable language, or hardware library modules which may include typical logic and arithmetic functions. The mapping from a structural form to the software code is essentially syntax-directed, for example, a concurrent combinator $[f_1, \ldots, f_k]$ is translated into a set of sequential statements which invoke the functions one by one. A linear recursive form or a map operator is translated into a forall-loop. However, the mapping from a structural form to a hardware component is *exact*, that is, there is a direct correspondence between the elements in forms and the hardware elements. For instance, combinators in forms are treated as interconnections, and arguments are treated as ports.

V. Conclusion

This paper presents a formal transformational codesign methodology which supports the concepts of "design-by-transformation" and "correct-by-construction". It is based on a functional notation, *form* which allows a unified specification of system components and joint design transformations. By applying different sequences of "correct" transformations to a form, various implementations and component structures can be explored using both qualitative and quantitative analysis. The results of the analyze permit design selections to be made during hardware/software partitioning and structural optimization. Although the example presented here targets only one class of problem solution strategies, the methodology does provide means of encoding rules and transformation mechanisms to solve problems in other computational strategy classes, thereby lifting the level of traditional hardware synthesis (behavioral level) to a level that allows early integration of algorithm design and hardware and software synthesis.

Acknowledgements

The first author is supported by an APRA and an AEA grant. The authors would like to thank Dr. Ricky Chan for his significant contribution in the development of the *form* notation. This work has been materially assisted by Australian Government GIRD contracts 16044 and 17028.

References

[1] Cheung T.K.Y. & Hellestrand G.R., "Multi-level Equivalence in Design Transformation," *Computer Hardware Description Languages Conference*, Chiba, Japan. Aug. 1995.

[2] Cheung T.K.Y., Hellestrand G.R. and Kanthamanon P., "A Multi-level Transformation Approach to HW/SW Codesign: A Case Study," *the 4th International Workshop on Hardware/Software Codesign*, Pittsburgh, Mar. 1996.

[3] Amellal S. & Kaminska B., "Functional Synthesis of Digital Systems with TASS," *IEEE Transactions on CAD*, Vol 13, No 5., May 1994.

[4] Gajski D.D. & Ramachandran L., "Introduction to High-Level Synthesis," *IEEE Design & Test of Computers*, p44-54, Winter 1994.

[5] Bergamaschi R.A., O'Connor R.A., Stok L., Moricz M.Z., Prakash S., Kuehlmann A., Rao D.S., "High-Level synthesis in an industrial environment," *IBM Journal of Research & Development*, Vol 39, No. 1/2 Jan/Mar. 1995.

[6] Chan R. & Hellestrand G.R., "VLSI Realisation from Recursive Expressions," *9th Australian Microelectronics Conf.*, Adelaide, Australia. 1990.

[7] Hellestrand G.R. "MODAL: A System for Digital Hardware Description and Simulation," *CHDL'79*, 1979.

[8] Hellestrand G.R., Chan R., Kam M.C., Cheung T.K.Y, Kanthamanon P. "Software-Hardware Engineering: Functional Specification → Structural Synthesis and Simulation," *2nd Int. Workshop on HW/SW Codesign*. 1993

[9] Cheung T.K.Y. & Hellestrand G.R., "Synchronisation Graph: Semantics and Applications," *Asia Pacific Conference on Computer Hardware Description Languages*, India. Jan 1996.

[10] Kahn G., "The semantics of a simple language for parallel programming," *Information Processing 74, Proc. IFIP Congress*, J.L. Rosenfeld (ed.) Amsterdam: p471-475, 1974.

[11] Johnson S.D. "Circuits and Systems: Implementing Communication with streams," *Parallel and Large scale computers: performance, architecture and applications*, Ed. by M. Ruschizka et al. 1983

[12] Backus J., "Can Programming be liberated from von Neumann style? A functional style and its algebra of programs," *Comm of ACM*, Vol 21 1978

[13] Chan R., "VLSI synthesis from Abstract Recursive Expressions," *PhD thesis*, School of EE & CS, University of New South Wales, 1990.

[14] Sheeran M., "μFP an algebraic VLSI design language," *Proc. ACM symp. on LISP and functional programming*, p104-112. 1984

[15] Chen M., "Space-time Algorithms: Semantics and Methodology," *PhD dissertation*, California Institute of Technology, may 1983.

[16] Smith D.R. "Automating the Design of Algorithms," *Formal Program Development*, Moller B., Partsch H., & Schuman S. (Eds) Lecture Notes in Computer Science 1993.

[17] Hellestrand G.R., Kanthamanon P., Chan R., Kam M.C., Cheung T.K.Y., "Functional Specification of Concurrency and Sequencing: Synthesisable Codesign Specification," *Asia Pacific Conf. on HDL* 1993.

[18] Hellestrand G.R., "The Unified Specification of Mixed Technology Systems," Invited paper, *SASIMI'95*, Japan, 1995.

[19] Darlington, P. (1982) "Program Transformation," *Functional programming and its application: an advanced course*, Ed. by J. Darlington, P. Henderson and D.A. Turner, Cambridge University Press.

[20] Luk W., Wu T., & Page I., "Hardware-Software Codesign of Multi-dimensional Programs," *Proc. IEEE Workshop on FPGA for Custom Computing Machines*. D.A. Buell & K.L.Pocek (eds). 1994.