A Multi-Level Transformation Approach to HW/SW Codesign: A Case Study

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

This reported work applies a transformational synthesis approach to hardware/software codesign. In this approach, the process of algorithm design is coupled early on with hardware design to allow for a complete design space exploration. Both the specification and the transformation mechanisms are encoded in a functional notation, called form*, which facilitates algorithmic derivation, structural transformation and verification. In the algorithmic derivation phase, possible computational schedules for a given application function are generated from a partial specification of the target architecture. At the hardware level, structural transformations are applied to explore possible datapath designs, where different designs yield different performance and cost. Other design metrics such as interface buffer size, software code size and data size etc. are also included to determine* **analytically** *a hardware/software partition.*

1. Introduction

The goal of our codesign process is to synthesize components from generic specifications, usually prescriptive in nature, and transform them to suitable structural forms which could be mapped either to software processes or hardware circuits. The specific objective of the codesign process is to find an implementation that satisfies all the system design constraints, and has least cost. In the codesign process, refinements of mixed hardware-software component behaviours are tightly coupled to afford a fine step traversal of the design space which enables us to avoid post-integration design optimisation typical of many other codesign processes. To support such an integrated design environment, we have a single high level notation that describes components in a manner independent of their final implementation technology. A single notation provides for codesign a unified system specification device and the possibility of joint hardware-software optimisation, which involves the analysis of different partitioning possibilities at the algorithmic level and the evaluation of design tradeoffs for different target implementations at the structural level.

This paper uses an example from a class of digital signal processing applications to illustrate the codesign process. The specification language is based on a functional notation, called a *form* [1,2]. Its declarative semantics facilitates transformational synthesis, symbolic manipulation and verification [3] which provide a basis for integrated design exploration during hardware/software partitioning. The design process starts with an executable specification in *form*, and proceeds through a series of symbolic manipulations and structural transformations. The symbolic manipulations are aimed at determining the best possible computational schedule that makes efficient use of the underlying resources (algorithmic derivation) and the structural transformations refine the behavior of a given hardware structure into one with lower cost. Its advantage is that a design synthesised with the transformation system is verifiably correct. Validation of its correctness is done by formal equivalence checking [3] which ensures that successive transformations preserve both the behaviour and functionality of the original design, while improving the performance or resources usage. The partitioned components are then mapped to a target embedded system architecture which consists of a programmable processor core (software module), an application specific coprocessor (hardware module) and a set of memory modules for data and instruction storage. To synchronise these communicating modules which are typically running at different rates, the inherent synchronisation associated with function application is mapped to an interface module. A typical interface module comprises control and data buffers with a specific invocation/interaction procedure (protocol) required for the exchange of data. The buffer size may vary with different partitions of hardware and software and, the amount and rate of data transferred between them. It is, therefore, best determined during hardware/software partitioning.

The remainder of this paper is organised as follows. Section 2 gives a brief introduction to the *form* notation. Section 3 presents the case study codesign of a two-dimensional discrete fourier transform operation, including its specification in *form*, the transformation into a more efficient schedule using divide-and-combine algorithmic derivation, the coprocessor design, the software generation and the hardware/software partitioning.

2. The form notation

The *form* is based on a variant of FP [4], with extensions to support multi-dimensional structured streams, delay functional and synchronized concurrent forms [1,2]. The main advantage of FP is its combinative property that allows function composition and construction by means of combining forms. It captures naturally function sequencing, concurrency and stream synchronization. The primary feature of a stream is its ability to model time-ordered flow of continuous data during function evaluation [5] and it supports modular interaction between entities implemented in hardware and/or software. This interaction between functions can be mapped to an implementation in many ways, such as shared buffers and common bus with an appropriate protocol to govern communication.

The main component of *form* is the set of combinators that describe sequential, concurrent, conditional and synchronizing behaviours. The application of a set of transforming algebraic rules allows trade off between resources and performance while maintaining behavioural and/or functional integrity [3]. The 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 the delay functional (denoted by Zf) which is used to construct loop/feedback structures [6]. 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 the sequence and (α+i) selects the (i+1)th element of the sequence in a finite multi-dimensional structure.) The intuitive meaning of each construct is stated below.

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. Thus, this computation is *non-speculative*. An *eager* evaluation of a conditional can be defined as:

$$p?[f_1, f_2] \triangleq ((\alpha+1, \omega)?\alpha) \cdot [p, f_1, f_2]$$

which applies the predicate and both branches of the conditional concurrently, then selects a result. Note that the two conditionals have different implications with respect to resource usage. The eager conditional consumes more resources for each stream data element than the non-speculative one. 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. Therefore, 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 same 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 sequences and treats them as first-class objects which can be referenced and passed as arguments to forms. Technically, the selector function can achieve the same objective as a named sequence, making naming redundant in the language. However, associating a name with a sequence can improve appreciably the readability of the notation. The DotProduct example below illustrates the use of the named sequences, a and b.

```
        hd :: α
        tail :: α+1... ω
DotProduct a b :: +· [ × (hd a) hd b, DotProduct (tail a) tail b]
```

3. The case study design

Many digital signal processing algorithms can be formulated naturally as divide-and-combine computations, such as the fourier transform and FIR filters etc. This class of computation exploits the potentially high degree of data parallelism by partitioning input data into separate, smaller subsets; performing the same computation on each subset; and then combining the results into a result for the whole. For appropriate data, the partitioning process may be applied recursively until the data size is small enough for the function to be applied directly. In this case study, where the objective is to codesign a system for computing the two dimensional fast fourier transform (2DFFT) operation, the function is implemented in a co-processor (hardware module) which contains many parallel resources for processing the partitioned data at high speed. The partitioning of the input data and arranging it in sequences into a data buffer, and the loading of instructions into the co-processor are accomplished by interleaving the execution of two software processes on a processor core (Figure 1). This divide-and-combine parallelisation method is used to generate a range of hardware/software partitions by varying the size of the data subset that can be processed by the coprocessor. To determine a suitable size for the hardware/software partition, an analytical model is developed to enable the performance evaluation of designs. The model is composed of a set of design parameters, such as software code size, hardware module size, buffer size at the interface and execution time of the computation etc. A heuristic search method may be used to locate a feasible solution that satisfies the design constraints. For each such partition, the behaviors of the hardware, software and interface modules are generated and the behavioral hardware module is then synthesised into an optimised structure.

Recently, Luk has also developed a hardware/software partitioning strategy for divide-and-combine algorithms. It is based on what he called a "divide by software and conquer by hardware" methodology [7,8]. Our approach is also based on this methodology. However, our work contrasts with his approach in that we have developed an analytical model for the performance evaluation of this class of algorithms, which takes into account the communication cost at the interface. This enables a more accurate determination of a hardware/software partition. The rest of this section presents the case study design.

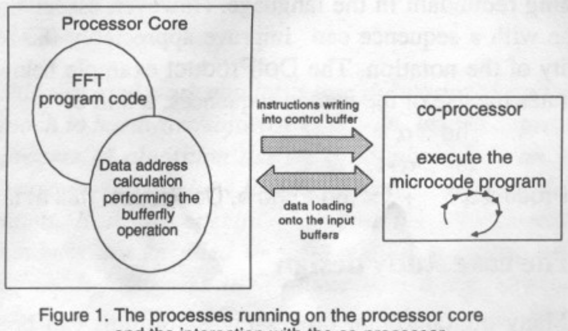

Processor Core
FFT program code
Data address calculation performing the bufferfly operation
instructions writing into control buffer
data loading onto the input buffers
co-processor
execute the microcode program

Figure 1. The processes running on the processor core and the interaction with the co-processor

3.1 Problem specification

The two dimensional discrete fourier transform (2DFT) of $x(n_1, n_2)$ is defined by:

$$X_{k_1k_2} = \sum_{n_1=0}^{N-1}\sum_{n_2=0}^{N-1} x_{n_1n_2}\,\omega_N^{n_1k_1}\omega_N^{n_2k_2}\ ,\ 0 \le k_1, k_2 \le N-1$$

$$\text{where}\quad \omega_N = e^{-\frac{2\pi i}{N}},\quad i = \sqrt{-1}$$

The *form* description of this mathematical definition, which is formulated in terms of a multidimensional map operator which is defined in the box below, follows.

The 2D input sequence $x(n_1,n_2)$ is expressed as:

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

and the coefficients matrix as:

$$W_m^{(2)} = [\,[\omega_N^{0},\ldots,\omega_N^{(m-1)k_2}],\ldots,[\omega_N^{(m-1)k_1},\ldots,\omega_N^{(m-1)k_2+(m-1)k_1}]\,]$$

where m=N. The innerproduct generator of the 2DFT function can be expressed as:

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

where the plus operator, + sums all of the elements in the input sequence. Finally, the 2DFT function is given by:

$$2DFT :: [\, X_{k_1k_2} \mid k_1 \leftarrow [0..N-1],\ k_2 \leftarrow [0..N-1]\,]$$

where the infix operator "←" successively instantiates the form for each of the indices.

3.2 Divide-and-combine algorithm derivation

To support algorithm design from specification, *form* provides a formalism for describing the symbolic manipulations of specifications. This formalism is based on meaning-preserving function transformations. A sequence of function transformations is applied to an initial *form* specification and derives a computational schedule with desired performance and resource usage. This requires a methodology to analyse the structure of a specification that is common to a class of algorithm, and to direct appropriate transformation mechanisms to derive the computation in that class. In this work, the derivation is carried out by hand (automatic approaches to structure analysis and the selection of transformation mechanisms to produce an efficient schedule are currently being explored).

This section gives the result of a derivation of 2D Fast Fourier Transform (2DFFT) from the 2DFT specification using the divide-and-combine parallelisation method. The resulting function F can be expressed as a general form:

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

① Multidimensional Map operator, $f^{\square}$:

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

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

where $A_n^{(m)}$, $B_n^{(m)}$ denote an 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. [1]

② Quadrant selectors:

first quadrant	$\alpha_q :: [\,[\,\alpha\ldots(\omega/2-1)]\cdot\alpha, \ldots, [\,\alpha\ldots(\omega/2-1)]\cdot(\omega/2-1)\,]$
second quadrant	$\alpha{+}1_q :: [\,[\,\omega/2\ldots\omega]\cdot\alpha, \ldots, [\,\omega/2\ldots\omega]\cdot(\omega/2-1)\,]$
third quadrant	$\alpha{+}2_q :: [\,[\,\alpha\ldots(\omega/2-1)]\cdot\omega/2, \ldots, [\,\alpha\ldots(\omega/2-1)]\cdot\omega\,]$
fourth quadrant	$\omega_q :: [\,[\,\omega/2\ldots\omega]\cdot\omega/2, \ldots, [\,\omega/2\ldots\omega]\cdot\omega\,]$

The quadrant selectors select all the elements in each quadrant from a 2D-sequence. Note that the row dimension is assumed to be the most significant dimension, which will be selected first before the column dimension.

where divide partitions the input data into subsets that are to be processed concurrently and independently by the function G, and the results are then combined using combine. In this derivation, the determination of the function divide requires knowledge of the underlying implementation technology which, in this case, is the coprocessor data path. For instance, in the case of the 2DFT operation, if an adequate number of multipliers is available in the data path, the divide function can partition an input 2D sequence into rows and then an entire row transform may be performed in a single operation. On the other hand, if only a few multipliers are available, an alternative schedule, such as transforming a small window at a time, is more appropriate and the divide function, in this case, may partition the input 2D sequence into subsequences. Assume here that the 2D sequence is divided into quadrants, and define:

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

Applying the divide-and-combine parallelisation, we get:

$$\begin{aligned} &F\ x_N^{(2)}\ W_N^{(2)} \\ &::\ +\cdot\times^{\square}\,(\,[\,F\ \alpha_q\cdot x_N^{(2)}\ \alpha_q\cdot W_N^{(2)},\ F\ \alpha{+}1_q\cdot x_N^{(2)}\ \alpha_q\cdot W_N^{(2)}, \\ &\qquad F\ \alpha{+}2_q\cdot x_N^{(2)}\ \alpha_q\cdot W_N^{(2)},\ F\ \omega_q\cdot x_N^{(2)}\ \alpha_q\cdot W_N^{(2)}\ ]\,) \\ &\qquad [\,\omega_N^{0},\ \omega_N^{(N/2)k_2},\ \omega_N^{(N/2)k_1},\ \omega_N^{(N/2)k_2+(N/2)k_1}\,] \qquad \text{—(1)} \end{aligned}$$

Equation (1) says that function F can be computed by partitioning the input 2D sequence into a sequence of four quadrants and applying F recursively and concurrently to each quadrant. The four results are then combined using the combine function "$+\cdot\times^{\square}$". The individual steps of this derivation are not included here for space reasons. Some of the transformation mechanisms that were applied are simple algebraic laws, such as law of associativity and distributivity, some are laws relating different multi-dimensional structures and, some are based on the unfold/fold transformations of Darlington [9].

The divide-and-combine parallelisation can be seen as a rooted balanced tree (fig 2) with leaf nodes representing the *least* task size supported by the implementation technology (the co-processor). A depth-first-traversal of this tree yields a *sequential* schedule (software code) for executing the co-processor operations. By expanding the tree into different levels, different sizes of leaf nodes are generated. Thus, the depth-first-traversal of this partially expanded tree enables different sizes of hardware/software partitions to be produced (Fig 2). The behavior of depth-first-traversal of this tree is expressed in a function S below: Let

$x'^{(2)}_N :: [\,\alpha_q,\ldots,\omega_q\,]\cdot x_N^{(2)}$ and $W'^{(2)}_N :: [\,\alpha_q,\ldots,\omega_q\,]\cdot W_N^{(2)}$

Define $S\ x'^{(2)}_N\ W'^{(2)}_N ::$

$$[\,F\ \alpha\cdot x'^{(2)}_N\ \alpha\cdot W'^{(2)}_N,\ ZS\ (\alpha{+}1\ldots\ \omega)\cdot x'^{(2)}_N\ W'^{(2)}_N\,]$$

which states that the concurrent applications of recursive function F in Equation (1), on the four quadrants, are sequentialized using the Z-operator in *form*. This sequential behavior represents a depth-first-traversal of the tree. Thus, from Equation (1),

$$F\ x_N^{(2)}\ W_N^{(2)}\ ::\ +\cdot\times^{\square}\,(\,S\ x'^{(2)}_N\ W'^{(2)}_N\,)$$
$$[\omega_N^{0},\ \omega_N^{(N/2)k_2},\ \omega_N^{(N/2)k_1},\ \omega_N^{(N/2)k_2+(N/2)k_1}]$$

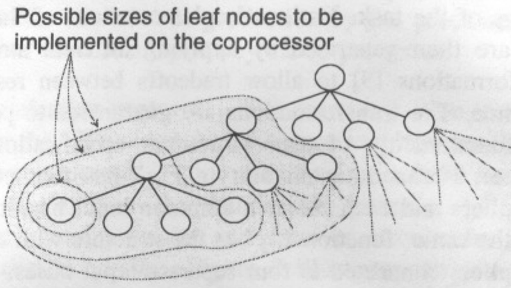

Possible sizes of leaf nodes to be
implemented on the coprocessor

Figure 2. Possible expansions of the divide-and-combine tree, with different sizes of leaf nodes to be implemented on the coprocessor as a piece of hardware or as a microcode.

3.3 Hardware co-processor architecture

The 2DFFT coprocessor may be realised as a semi-custom ASIC processor with the following synthesisable units:

- A programmable microcoded control unit which consists of a sequencer, a control store, and an address generator to access a coefficient ROM.
- An application specific data path which contains a shared register file, coefficient ROM, complex multiplier/accumulators and shared input/output buses.
- A buffered interface unit which uses a pre-defined R/W handshake protocol and supports non-blocking Read/Write communication. The main components of this unit are a FIFO controller and three FIFO buffers for control instructions, and input and output data.

The coprocessor begins its instruction execution when the processor core has filled up both the input data buffer and the control buffer. The coprocessor control unit then decodes the instruction at the top of the control buffer and executes the microcode segment that interprets the instruction. This process is halted once the control buffer is empty and resumed when the buffer is filled up again by the processor core. The processor core will read the results from the output buffer when the process terminates. An architectural model of the coprocessor is shown in Figure 3.

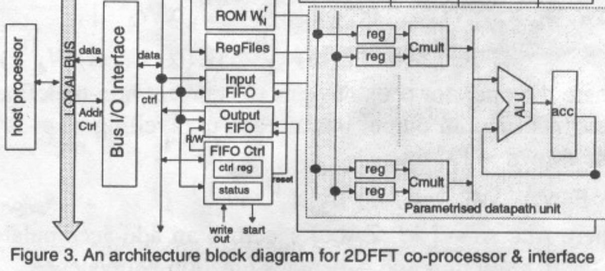


Figure 3. An architecture block diagram for 2DFFT co-processor & interface

Datapath – A partial expansion of the divide-and-combine tree enables the determination of a suitable size for the hardware/software partition. Leaf node of the partially expanded tree defines a task to be implemented on the hardware coprocessor. Its datapath is characterised by the behaviors of the task. Various implementations of the datapath are then generated by applying different structural transformations [3] to allow tradeoffs between resources and time. The transformations are guaranteed to preserve the functionality of the structural specification. For instance, if the datapath unit in Fig3 has four complex multipliers and each shares a common input bus, it would have the same "functionality" as the structure with the four multipliers connected to four separate input buses. On the other hand, different structures which are function-equivalent may yield very different performances. In the previous case, if a complex multiplication takes four clock cycles and an addition takes one cycle, the former structure requires nine cycles to complete while the latter one takes only five cycles.

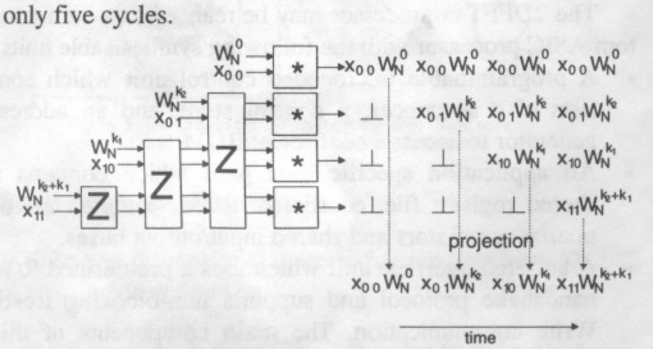


Figure4. The behavioral model of the datapath structure in Fig3.

The behavior of the datapath unit in Fig3 is specified by a linear recursive form Φ_k (defined below), which models the shared input bus structure of a set of multipliers by recursively delaying their input sequences using Z-operators so that data elements are fed into each multiplier at separate time steps (see Fig 4). Assume that the divide-and-combine tree of the function F in Equation 1 is expanded in full, the leaf node of the tree would define the base case of F, denoted by F_b below.

$F_b\, x_4^{(1)}\, W_4^{(1)} :: +\cdot \times^{\square}\, x_4^{(1)}\, W_4^{(1)}$

where $x_4^{(1)} = [x_{00}, x_{01}, x_{10}, x_{11}]$

and $W_4^{(1)} = [\omega_N^{\,0}, \omega_N^{\,k_2}, \omega_N^{\,k_1}, \omega_N^{\,k_1+k_2}]$

Now, $\Phi_k\, x_4^{(1)}\, W_4^{(1)} :: \mathrm{proj}_t\cdot [\, \times\cdot [\alpha\cdot x_4^{(1)}, \alpha\cdot W_4^{(1)}]\,,$

$\qquad \Phi_k\, (Z\cdot(\alpha+1\ldots\,\omega)\cdot x_4^{(1)})\, (Z\cdot(\alpha+1\ldots\,\omega)\cdot W_4^{(1)})\,]$

where the operator proj_t projects the current non-undefined elements onto an output bus at each unravelling step. It is also shown in [3] that:

$$F_b\, x_4^{(1)}\, W_4^{(1)} \cong_f \mathrm{Acc}\cdot \Phi_k\, x_4^{(1)}\, W_4^{(1)}$$

where Acc = +· [id, ZAcc] specifies an add-accumulator in the datpath unit and $\cong_f$ denotes function-equivalence.

Hardware multiplier – In the datapath description, the operator "×" represents a *functional abstraction* of the complex multiplication operation. The abstract function is progressively refined to a more detailed level until it can be directly mapped into library components which is the lowest level of a given implementation technology. The library functions may include normal logic and arithmetic functions, or even more complex floating point operations. *Form* supports the description of functions at different levels of refinement. An example *form* description of a serial multiplier is given below and is one of the library functions being used to implement the ×-operator.

```
/* One bit adder description */
sum :: ⊕· [ α... α+2]
carry :: or · [ and· [α+2, ⊕· [ α, α+1] ], and· [α, α+1] ]
onebitAdder :: [carry, sum]
/* n-bits adder description */
nbitAdd  a b carryin
   :: [ α·PrevAdd,  (nbitAdd  (α... ω-1)· a (α... ω-1)· b
                                  α·PrevAdd) ], ω·PrevAdd ]
      where PrevAdd :: onebitAdder·[ ω·a, ω·b, c]
nbitAdder :: [ ω/2, [ (ω/2)+1... ω] ] · nbitAdd
/* Add and Shift */
shifter :: [ α... ω-1]
AddShift  c  acc  multiplier  multiplicand
   :: [0, shifter· [(nbitAdder· [acc, multiplicand, 0],
                       [c,acc] )?ω·multiplier , multiplier] ]
/* nbit serial multiplier */
SMult :: (( [ AddShift· [α... ω-1], ω-1, dec· ω],
                       id )?gt0, ω ) · Z SMult
SMult@[0] a b = [0,0,a,b,n]

SerialMult a b :: [ α+1, α+2]· (SMult a b)
```

Software code generation – refers to the process of generating from a *form* specification an executable program for both the processor core and the coprocessor. In the algorithmic derivation phase, a *partial-order schedule* of the operations is derived. The derived specification may contain concurrent combinators $[f_1, \ldots, f_k]$, so invocations of the concurrent functions f_1 to f_k must be sequentialized in order to run on the processor core. Thus, at the program level, the partial order schedule is converted to a *total order schedule*. However, at the micro-program level, the partial order schedule remains in order to utilize the full parallelism offered by the coprocessor datapath. Each of the *form* combinators in the total order schedule is then translated into code in a standard programming language, for example, a linear recursive form or a map operator is translated to a for–loop.

Recall from Section 3.2 that function S expresses the sequential depth-first-traversal of the tree, which can be translated into nested-for-loops with the function F being invoked once in each loop. If the data size is assumed to be

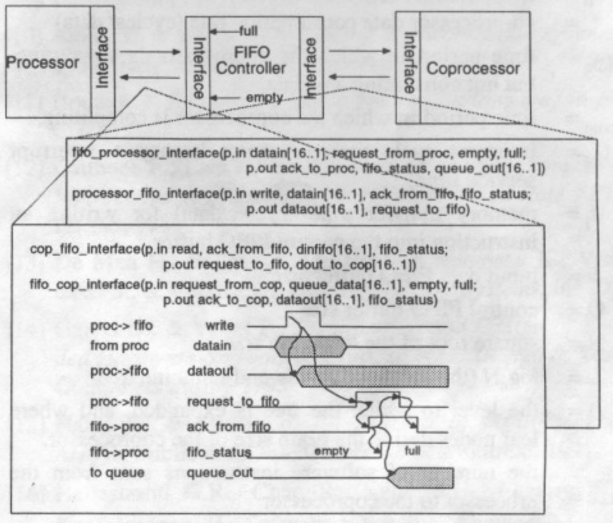
N×N where $N=2^L$ and there are L-1 levels of nested loops. The innermost loop computes the base case of F on the coprocessor, which reads its input sequence $x_N^{(2)}$ from the input data buffer. At each invocation of the function F, the combine function "+·×°", which uses the results of S, is also computed on the coprocessor. This requires the results of S to be stored in a coprocessor register-file and the required maximum number of registers is 4×(L-1), that is, four for each level of the tree. In the computation of the combine function, the coprocessor also reads the constant coefficient sequence $W_N^{(2)}$ from the coefficient ROM. The maximum ROM size for storing the coefficients is 4×L.

The result of the code generation for both the processor core and the coprocessor is shown in Fig 5. Note that the input sequence $x_N^{(2)}$ has already been rearranged by a software process AddressGenerationProcess(), so that the right elements are processed in each loop. This software process starts running when the control buffer is filled. When the buffer becomes empty, an empty signal is generated to interrupt the processor core which then immediately resumes another process (2DFFT process in Fig 5) to feed more instructions into the control buffer, and suspends the AddressGenerationProcess() process by saving its current context. This process can be characterized by a 4-way recursive function (Fig 6) where each recursion implements one of the four quadrant selector functions ($\alpha_q \cdots \omega_q$).

3.4 Hardware/software interface

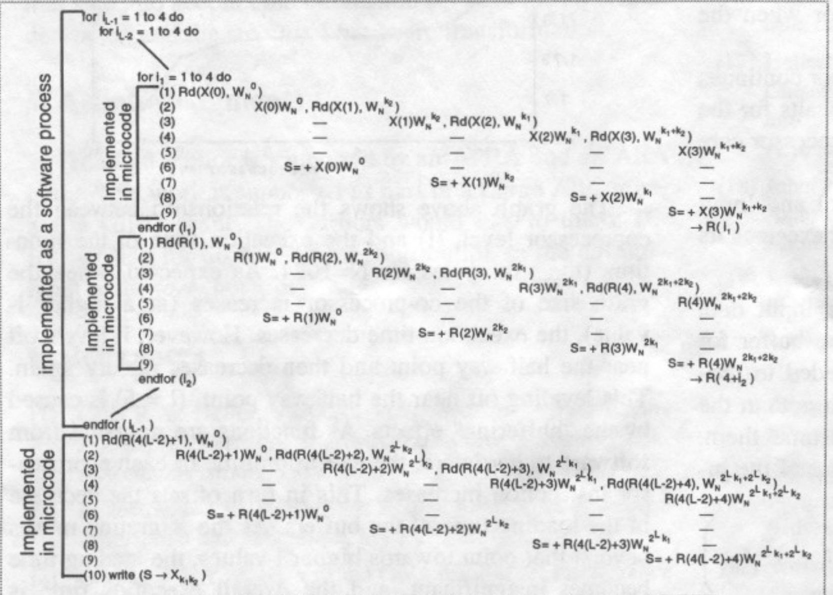
The interface module between the processor and the coprocessor consists of three FIFO buffers, a control buffer, and an input and an output data buffer. Each buffer entry can be accessed using memory-mapped I/O. When the buffers have been loaded, the processor asserts the start signal (by writing an appropriate control code to a control register) of the FIFO controller, which then starts a coprocessor execution cycle by sending it an instruction. The processor polls the results in the output buffer. Note that the size of the input buffer is determined by the differences between the rate of data production by the processor and the rate of data consumption by the co-processor.

Communication protocols – The communication interface between the FIFOs and the processor and between FIFOs and the co-processor is composed by a predefined template which implements the handshake protocol [5]. The template contains sender and receiver circuits for asynchronous data transfer. Figure 7 illustrates the interfaces specification and the Read/Write handshake protocol, that have been described in the MODAL HDL [2].



```
fifo_processor_interface(p.in datain[16..1], request_from_proc, empty, full;
                         p.out ack_to_proc, fifo_status, queue_out[16..1])
processor_fifo_interface(p.in write, datain[16..1], ack_from_fifo, fifo_status;
                         p.out dataout[16..1], request_to_fifo)
cop_fifo_interface(p.in read, ack_from_fifo, dinfifo[16..1], fifo_status;
                   p.out request_to_fifo, dout_to_cop[16..1])
fifo_cop_interface(p.in request_from_cop, queue_data[16..1], empty, full;
                   p.out ack_to_cop, dataout[16..1], fifo_status)
```

Figure7. Interface between the FIFO controller, processor and co-processor



implemented as a software process
Implemented in microcode
Implemented in microcode
Implemented in microcode
for i_{L-1} = 1 to 4 do
for i_{L-2} = 1 to 4 do
for i_1 = 1 to 4 do
(1) Rd(X(0), W_N^0)
(2) X(0)W_N^0, Rd(X(1), $W_N^{k_2}$)
(3) X(1)$W_N^{k_2}$, Rd(X(2), $W_N^{k_1}$)
(4) X(2)$W_N^{k_1}$, Rd(X(3), $W_N^{k_1+k_2}$)
(5) X(3)$W_N^{k_1+k_2}$
(6) S= + X(0)W_N^0
(7) S= + X(1)$W_N^{k_2}$
(8) S= + X(2)$W_N^{k_1}$
(9) S= + X(3)$W_N^{k_1+k_2}$ → R(i_1)
endfor (i_1)
(1) Rd(R(1), W_N^0)
(2) R(1)W_N^0, Rd(R(2), $W_N^{2k_2}$)
(3) R(2)$W_N^{2k_2}$, Rd(R(3), $W_N^{2k_1}$)
(4) R(3)$W_N^{2k_1}$, Rd(R(4), $W_N^{2k_1+2k_2}$)
(5) R(4)$W_N^{2k_1+2k_2}$
(6) S= + R(1)W_N^0
(7) S= + R(2)$W_N^{2k_2}$
(8) S= + R(3)$W_N^{2k_1}$
(9) S= + R(4)$W_N^{2k_1+2k_2}$ → R(4+i_2)
endfor (i_2)
endfor (i_{L-1})
(1) Rd(R(4(L-2)+1), W_N^0)
(2) R(4(L-2)+1)W_N^0, Rd(R(4(L-2)+2), $W_N^{2^L k_2}$)
(3) R(4(L-2)+2)$W_N^{2^L k_2}$, Rd(R(4(L-2)+3), $W_N^{2^L k_1}$)
(4) R(4(L-2)+3)$W_N^{2^L k_1}$, Rd(R(4(L-2)+4), $W_N^{2^L k_1+2^L k_2}$)
(5) R(4(L-2)+4)$W_N^{2^L k_1+2^L k_2}$
(6) S= + R(4(L-2)+1)W_N^0
(7) S= + R(4(L-2)+2)$W_N^{2^L k_2}$
(8) S= + R(4(L-2)+3)$W_N^{2^L k_1}$
(9) S= + R(4(L-2)+4)$W_N^{2^L k_1+2^L k_2}$
(10) write (S → $X_{k_1 k_2}$)

Figure 5. The software code generated for the 2DFFT

```
AddressGenerationProcess()
{
   IndexCalculation(N,0,0);
   exit()
}

IndexCalculation (m, i, j)
{
   If (m ≥ 2) {
      IndexCalculation(m/2, i, j);
      IndexCalculation(m/2, i+m/2, j);
      IndexCalculation(m/2, i, j+m/2);
      IndexCalculation(m/2, i+m/2, j+m/2);
   } else
      Transfer X_{N*i+j} to the input data
FIFO.
      /* assume linear memory array */
}
```

Figure 6. The Address Generation Process

3.5 Performance Evaluation – HW/SW partitioning

Given a divide-and-combine tree derived from an initial *form* specification, various hw/sw partitions can be created by expanding the tree into different levels, with the leaf nodes to be implemented in hardware (see Fig 2). The size of the leaf nodes defines the granularity of the hardware module where the larger the grain size, the lesser the interaction between hardware and software, and the greater the concurrency and the higher the performance. However, a larger grain size also means a higher hardware cost. This section examines various design metrics to assist in the hardware/software partitioning. To estimate the overall system performance of this case study design, the following parameters of the architectural model were considered:

t_p =	processor data production rate (cycles/data)
t_c =	co-processor data consumption rate (cycles/ data)
t_{idle} =	time period in which the coprocessor is computing, but not consuming any data.
t_{comp} =	time period in which the coprocessor is computing.
t_{int} =	interrupt cycle time (interrupt latency + interrupt service time)
t_i =	memory transfer time (cycles/data) for writing an instruction into the control FIFO buffer.
S =	input data FIFO buffer size
Q =	control FIFO buffer size
N =	square root of the total data size
L =	$\log_4 N$ (the height of divide-and-combine tree)
l =	the level to which the tree is expanded, and where leaf nodes define the grain size of the coprocessor.
code size =	the number of software instructions sent from the processor to the coprocessor

The following assumptions for the model are made:

1. The FIFO controller interrupts the processor when the control buffer is empty
2. Once an interrupt is signaled, the coprocessor continues to complete its current instruction. It then waits for the control buffer to be filled up again by the processor core before resuming its instruction execution.
3. During the initialization cycles, the control and input buffers are being filled up. The coprocessor executes its first instruction once they have been filled.

To enable the coprocessor to consume its input data *continuously* (i.e. data is always present in the buffer for the coprocessor to process), the total time needed to produce all the data from the processor must be less than the total time taken for the co-processor to consume them. With this requirement, an inequality for the size of the input buffer is established (proof is omitted here):

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

where codesize = $1+4+\ldots+4^{L-l}$ with $l \leq L$.

The size of the data buffer at the interface is crucial to the overall performance of the function. For instance, if S is below the value determined by the inequality, the potential parallelism between the processor and coprocessor will be severely reduced as the coprocessor has to stop and wait for the processor to write data into the buffers. By selecting the lower bound for S as the size of the data buffer in a hardware/software partition, the execution time of the function can be evaluated.

Case ①: $t_{comp} \geq t_{int}$ — this states that during the interrupt cycles, if the control buffer has been filled up but before the coprocessor finishes its current instruction (the one that causes the interrupt), the coprocessor can continue to execute its next instruction without stopping. The overall execution time, t_{exec} is equal to the time needed to initialize the control and data buffers, t_{init} plus the total coprocessor execution time, t_{cop}.

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

t_{cop} = codesize × (coprocessor microcode execution time)

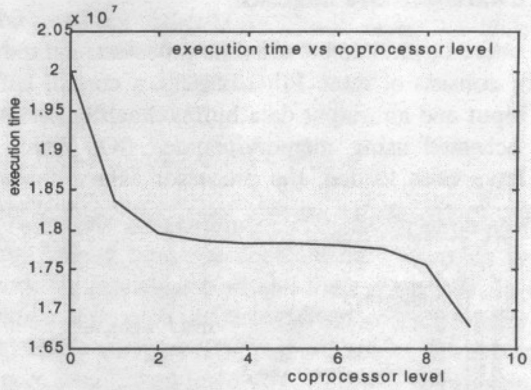
$= (1+4+\ldots+4^{L-l}) \times (4^l \times 9 + 4^{l-1} \times 9 + \ldots + 4^0 \times 9)$

$t_{exec} = t_{init} + t_{cop}$

Case ②: $t_{comp} < t_{int}$ — this states the reverse, which suggests that the coprocessor must idle until the control buffer has been refilled.

$t_{exec} = t_{init} + t_{cop}$ + (refilling time of the control buffer)

$= t_{init} + t_{cop} + (t_{int} - t_{comp}) \times$ (codesize/Q – 1)



The graph above shows the relationship between the coprocessor level, (l) and the execution time of the function, (t_{exec}) in cycles for N=1024. As expected, when the grain size of the co-processor increases (at a higher l-value), the execution time decreases. However, it levels off near the half-way point and then decreases rapidly again. This leveling off near the half-way point, (l = 5) is caused by the "buffering" effects. As functions are migrated from software to hardware, the execution time of each coprocessor instruction increases. This in turn offsets the decrease in the loading time of the buffers. As the migration moves beyond that point towards higher l-values, the loading time becomes insignificant, and the overall execution time is dominated by the coprocessor execution time.

4. Concluding remarks

This paper presents a codesign framework based on a functional notation, *form* which supports unified specification of system structures, joint design exploration, verification, and system performance evaluation. The main objective is to find an appropriate partition of hardware and software to efficiently compute a function. The methodology here does not provide solutions to every class of computational paradigms, but it does provide the means of encoding rules and transformation mechanisms to solve a particular class of problem, thereby lifting the level of traditional hardware synthesis (behavioral level) to allow the early integration of algorithm design (function level) with hardware design. The particular class of computations being considered are the divide-and-combine data parallel algorithms. The codesign flow starts with an executable specification in *form*, and applies a divide-and-combine algorithm derivation. This generates a set of possible partial-order schedules (divide-and-combine tree) to run on a partially defined coprocessor model. At the hardware level, a set of structural hardware transformations can be applied to yield a performance effective datapath. Each of the transformation mechanisms is formally verified to preserve the functionality of the initial structure. From the partial-order schedule, an analytical performance evaluation technique is used to locate the level of partitioning, that satisfies some performance-cost constraints. Once the level of partitioning is determined, the partial-order schedule can be converted to a total-order-schedule to be implemented in software, and the hw/sw interface is then generated. This case study demonstrates this design process. Future work will include the use of intelligent rules selection to assist in the algorithm derivation and possible mixed-level transformations that take into account the interaction between the algorithm derivation and the structural hardware transformation.

5. Acknowledgments

The first author is supported by an APRA and an AEA grant. This work is supported as part of a Large ARC grant and a GIRD grant. The authors would like to thank Dr. Ricky Chan for his significant contribution in the development of the *form* notation.

6. References

[1] Chan R. & Hellestand G.R., *VLSI Realisation from Recursive Expressions*, in 9th Australian Microelectronics Conf., Adelaide, Australia. 1990.

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

[3] Cheung T.K.Y. & Hellestrand G.R., *Multi-level Equivalence in Design Transformation.* in CHDL'95.

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

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

[6] Sheeran M., *μFP an algebraic VLSI design language*, in ACM symp. on LISP and functional programming, 1984

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

[8] Luk W., Lok V. and Page I., *Hardware acceleration of divide-and-conquer paradigms: a case study*, in Proc. IEEE Workshop on FPGAs for Custom Computing Machines 1993

[9] Darlington, J. *Program Transformation*, in Functional programming and its application: an advanced course, Ed. by J. Darlington, P. Henderson and D.A. Turner, 1982.

[10] Amon T. & Borriello G., *Sizing Sychronization Queues: A Case Study in High Level Synthesis*, in DAC, p690-3 1991.

[11] Boute R.T. *Declarative languages - still a long way to go*, in CHDL'91.

[12] Catthoor F., Lanneer D., De Man H., *Application Specific Microcoded Architectures for Efficient Fixed-Rate FFT.* in ISCAS, 1989.

[13] De Man H., Bolsens I., Lin B., Van Rompaey K., Vercauteren S., & Verkest D., *Codesign of DSP Systems*, in [20].

[14] Gajski D., & Vahid F., *Specification and Design of Embedded Hardware-Software Systems*, in IEEE Design & Tests of Computers Spring 1995.

[15] Hains G. & Mullin L. *An Algebra of Multidimensional Arrays*, Publication 783, Universite de Montreal, 1991.

[16] Hellestrand G.R., Chan R., Kam M.C., Cheung T.K.Y, Kanthamanon P. *Software-Hardware Engineering: Functional Specification* $\rightarrow$ *Structural Synthesis and Simulation*, in 2nd Int. Workshop on HW/SW Codesign. 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*, in Asia Pacific Conf. on HDL 1993.

[18] Johnson S.-D. *Digital Design in a Functional Calculus*, S.D. Johnson, in Formal Aspects of VLSI Design, Ed. by G. J. Milne and P. A. Subrahmanyam. 1986

[19] Jones G. *Deriving the fast Fourier algorithm by calculation,* in Functional Programming Languages and Computer Architecture, J. Hughes (Ed) LNCS. 1991

[20] Nato Advanced Study Institute, *Hardware/Software Co-Design Lecture Notes* Vol 1 & 2. 1995. Directors: De Micheli G & Sami M.G.

[21] Smith D.R. *Automating the Design of Algorithms*, in Formal Program Development, Moller B., Partsch H., & Schuman S. (Eds) LNCS 1993.

[22] Wolf W.H. *Hardware-Software Codesign of Embedded Systems*, in Proceedings of IEEE, Vol 82. No 7. 1994.