System Synthesis via Hardware-Software Co-design

Rajesh K. Gupta Giovanni De Micheli

Technical Report CSL-TR-92-548
October 1992

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University, Stanford, CA 94305-4055.

Abstract

Synthesis of circuits containing application-specific as well as re-programmable components such as off-the-shelf microprocessors provides a promising approach to realization of complex systems using a minimal amount of application-specific hardware while still meeting the required performance constraints. We formulate the synthesis problem of complex behavioral descriptions with performance constraints as a hardware-software co-design problem. The target system architecture consists of a software component as a program running on a re-programmable processor assisted by application-specific hardware components. System synthesis is performed by first partitioning the input system description into hardware and software portions and then by implementing each of them separately. We consider the problem of identifying potential hardware and software components of a system described in a high-level modeling language. Partitioning approaches are presented based on decoupling of data and control flow, and based on communication/synchronization requirements of the resulting system design.

Synchronization between various elements of a *mixed* system design is one of the key issues that any synthesis system must address. We present software and interface synchronization schemes that facilitate communication between system components. We explore the relationship between the non-determinism in the system models and the associated synchronization schemes needed in system implementations.

The synthesis of dedicated hardware is achieved by hardware synthesis tools [1], while the software component is generated using software compiling techniques. We present tools to perform synthesis of a system description into hardware and software components. The resulting software component is assumed to be implemented for the DLX machine, a load/store microprocessor. We present design of an ethernet based network coprocessor to demonstrate the feasibility of mixed system synthesis.

Key Words and Phrases: System-level synthesis, High-level Synthesis, System Partitioning, Hardware-Software Co-design, Multiple Chip Modules (MCMs)

Copyright © 1992
by
Rajesh K. Gupta and Giovanni De Micheli

Contents

1 Introduction **1**
- 1.1 Motivations for hardware-software partitioning . . . 1
- 1.2 The problem of system synthesis . . . 4
- 1.3 Applications . . . 5

2 System Architectures Based on Hardware-Software Components **6**
- 2.1 Target System Architecture . . . 8

3 Specification and Modeling of Hardware-Software Systems **9**
- 3.1 System specification using *HardwareC* . . . 10
  - 3.1.1 Memory and Communication . . . 11
  - 3.1.2 Nondeterminism in System Specifications . . . 12
- 3.2 System Model . . . 12
  - 3.2.1 Communication . . . 17
- 3.3 Specification of Timing Constraints . . . 19
- 3.4 Data Rate Constraints . . . 20

4 The Problem of Hardware-Software Partitioning **22**
- 4.1 Processor Model . . . 22
- 4.2 Modeling of Software Performance . . . 24
- 4.3 Partitioning Feasibility . . . 27
- 4.4 Algorithms for System Partitioning . . . 28
- 4.5 System partitioning based on system non-determinism . . . 28
- 4.6 Partitioning based on decoupling of control and execution . . . 31

5 Implementation of Hardware Components **32**
- 5.1 Hardware Timing and Resource Constraints . . . 33
- 5.2 Constrained Hardware Partitioning . . . 33

6 Implementation of Software Components **34**
- 6.1 Rate constraints and software performance . . . 37
- 6.2 Representation of Inter-thread dependencies . . . 40
- 6.3 Control Flow in the Software Component . . . 40
- 6.4 Concurrency in Software Through Interleaving . . . 41
- 6.5 Issues in Code Generation from Program Routines . . . 42
  - 6.5.1 Memory allocation . . . 43
  - 6.5.2 Data types . . . 43
  - 6.5.3 The C Standard Library . . . 43
  - 6.5.4 Linking and loading compiled C-programs . . . 44
  - 6.5.5 Interface to assembly routines . . . 45

7 System Synchronization **46**
7.1 Hardware-Software Interface Architecture 49
7.2 Example 51

8 Example of System-level Synthesis: Network Coprocessor **53**
8.1 Host CPU-Coprocessor Interface 54
8.2 Coprocessor Operation 54
8.3 Coprocessor Architecture 54
8.4 Network Coprocessor Implementation Results 56

9 Summary **59**

10 Acknowledgments **60**

11 Appendix A: Processor Characterization in Vulcan-II **63**

List of Figures

1 *Example of a Mixed System Implementation* . . . 2
2 *DES Encryption Scheme* . . . 3
3 *System Synthesis Procedure* . . . 6
4 *System Classification Based on HW/SW Components* . . . 7
5 *Target System Architecture* . . . 8
6 *Linear Code versus Data-Flow Graph Representations* . . . 10
7 *Example of a sequencing graph model* . . . 15
8 *The Constraint Graph Model* . . . 19
9 *Specification of rate constraint as a min/max timing constraint* . . . 21
10 *Determination of minimum static storage for single execution thread* . . . 27
11 *Partitioning into Hardware Control and Software Execute Processes* . . . 31
12 *Partitioned Hardware Model* . . . 32
13 *Steps in generation of the software component* . . . 35
14 *Example of a graph model containing unknown delay operations* . . . 36
15 *Generating fixed addresses from C-programs* . . . 44
16 *Control FIFO schematic* . . . 47
17 *FIFO control state transition diagram* . . . 47
18 *Hardware and Software Interface Architecture* . . . 48
19 *Hardware and Software Interface Model* . . . 49
20 *Graphics Coprocessor Block Diagram* . . . 50
21 *Graphics Coprocessor Implementation* . . . 50
22 *Graphics Coprocessor Simulation* . . . 51
23 *Graphics Controller Software Component* . . . 52
24 *Network Coprocessor Block Diagram* . . . 55
25 *Network Coprocessor Implementation* . . . 56
26 *Network Coprocessor Simulation* . . . 58

List of Tables

1 *Sequencing graph operation vertices* 16
2 *Addressing Modes* 23
3 *Comparison of program thread implementation schemes* 42
4 *A comparison of control FIFO implementation schemes* 52
5 *Network Coprocessor Instruction Set* 54
6 *Network Coprocessor Synthesis Results using LSI LCA10K Gates* 57
7 *Network Coprocessor Synthesis Results using Actel Gates* 57
8 *Network Coprocessor Software Component* 58

System Synthesis via Hardware-Software Co-design

Rajesh K. Gupta Giovanni De Micheli

Technical Report CSL-TR-92-548
October 1992

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University, Stanford, CA 94305-4055.

1 Introduction

Existing high-level synthesis techniques attempt to generate a purely hardware implementation of a system design either as a single chip or as an interconnection of multiple chips each of which is individually synthesized [1] [2] [3] [4]. A common objection to such an approach to ASIC design is the cost-effectiveness of an *application-specific* hardware implementation versus a corresponding software solution using standard *re-programmable* components, such as off-the-shelf microprocessors. Often system design requires a *mixed* implementation, that blends ASIC chips with processors, memory and other special purpose modules like multimedia, transducer and DSP modules. Important examples are embedded controllers and telecommunication systems. In practice most such systems consist of hardware and software components - hence the term *firmware* is often used to describe these systems. When considering the problem of firmware synthesis, an important issue is the definition of boundaries between the hardware and the software components. In some cases, this boundary can be dictated by issues such as analog interfaces that require a specialized hardware implementation. In this report we consider instead the problem in which implementations are sought for synchronous digital systems, and where the choice between dedicated hardware and software solutions are driven by system performance and cost requirements.

1.1 Motivations for hardware-software partitioning

Indeed, most digital functions can be implemented by software programs. The major reasons for building dedicated ASIC hardware is the satisfaction of performance constraints. These performance constraints can be on the overall time (latency) to perform a given task, or more specifically on the timing to perform a subtask and/or on the ability to sustain specified input/output data rates over multiple executions of the system model. The hardware performance depends on the results of scheduling and binding and on basic performance characteristics of individual hardware blocks. Whereas the number of cycles that it takes a general re-programmable processor to execute a routine depends on the number of instructions it

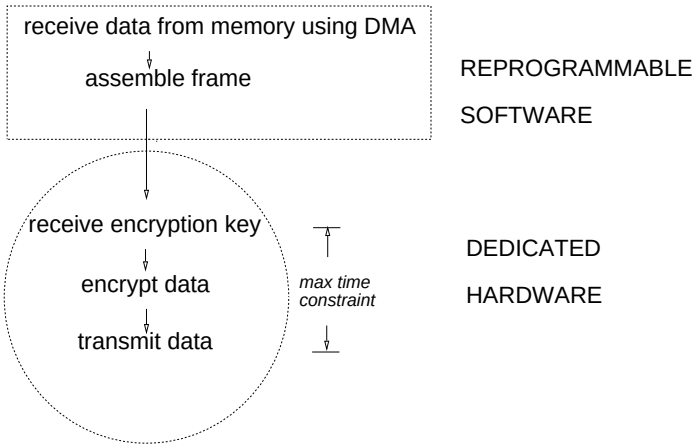


Figure 1: *Example of a Mixed System Implementation*

must execute and the cycle-per-instruction (CPI) metric of the processor. In general, application-specific hardware implementations tend to be faster since the underlying hardware is tailored and optimized for the specific set of tasks. However in absence of stringent performance constraints, for a given behavioral description of an ASIC machine, some parts (subroutines) of it may be well suited to a commonly available re-programmable processor (like 6502, 68HC11, 8051, 8096 etc) while others may take too long to execute. For instance, most general purpose CPU's deal with byte operands whereas many ASIC controllers contain bit oriented operations resulting in unnecessary overheads when implemented entirely in software. However, the software implementations do provide the ease and flexibility of reprogramming for the possible price of loss of performance.

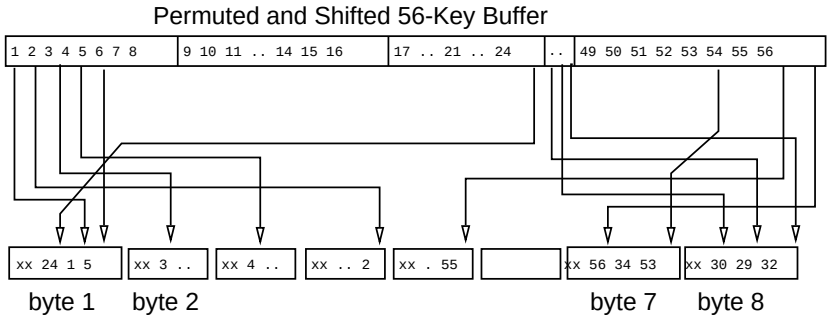
Example 1.1.

To be specific, consider design of a **data encryption/protocol controller chip**, such as DES (Data Encryption Standard) used by commercial banks or AES (Audio Engineering Society) protocol used for communication between digital audio devices and computers. In Figure 1, the DES transmitter takes data from memory using a DMA controller, assembles the frame for transmission, it encrypts the data after it receives the key and transmits the encrypted data. Encryption protocol requires that the encrypted data be transmitted within a certain time duration of receiving the encryption key. In the DES protocol, a 56-bit encryption key is used to transform 64 bits of 'plaintext'. Software implementations of the encryption algorithm shown in Figure 2 vary from 300 to 3000 instructions depending on the level of bit-oriented operations supported. The hardware implementation on the other hand can be implemented to work in 16 cycles times of most digital systems.

It is possible to implement the DES controller in Figure 1 either as a program on a general purpose re-programmable component or as dedicated ASIC chips [5]. However, as shown in Figure 2, most encryption/decryption is a long, iterative process of rotations, XOR operations, bit permutations and table lookups. Further, these protocols often use bit-reversal operations as a part of overall encoding strategy. A bit-permutation operation can be implemented easily in dedicated hardware while it may take too long to execute as a sequence of instructions on most processors. While implementing the complete protocol controller on dedicated hardware may be too expensive, an implementation which uses a re-programmable component may satisfy performance requirements and at the same time provide the ease and flexibility of reprogramming in software.

```
clear 64-bit output buffer

for each bit i = 1 .. 48 of keyed buffer do {

   isolate bit i of the keyed buffer

   if (bit i = 1)
      set output buffer bit f(i) using map table, f

}
```

SOFTWARE: 300 TO 3000 INSTRUCTIONS



HARDWARE: MAXIMUM OF 16 CYCLES

Figure 2: *DES Encryption Scheme*

□

Example 1.2.

While bit-wise shifting and xor operations lead to a slower software implementations, corresponding byte-wise implementations was considerably faster. Given extra memory for table-storage, the program implementations can be speeded-up even more using table-lookup methods. Such implementations are often competitive with corresponding hardware implementations.

Consider a **16-bit CRC-CCITT computation** using polynomial $x^{16} + x^{12} + x^5 + 1$ below.

With the addition of every byte of data, the new CRC is clearly is a function of 8-bits of old CRC and the new byte of data. This function is precomputed and stored in an 256-entry table. Thus, a byte-wise implementation using two 256-byte tables, as described by the following pseudo-code, when coded in assembly can achieve 16-bit CRC computation in 7 instructions per byte.

```
typedef byte char;

byte Table_low[256], Table_high[256];
byte Temp, data, CRC_low, CRC_high;

Temp = data xor CRC_low;
CRC_low = Table_low[Temp] xor CRC_high;
CRC_high = Table_high[Temp];
```

The actual latency of computation is a strongly dependent on the instruction-set architecture (ISA) of the target processor. The best implementation of the above pseudo-code on an Intel 8086 processor computes 16-bit CRCs in 9 instructions, a Motorola 68K implementation in 11 instructions and a RISC-based implementation in 14 instructions. □

1.2 The problem of system synthesis

The problem of mixed system synthesis is complex, and todate there are no available CAD tools to support it. This report addresses the hardware-software co-design issue by formulating it as a system partitioning problem into application-specific and re-programmable components. As explained in Section 4, we can also view it as an extension of high-level synthesis techniques to systems with generic re-programmable resources. Nevertheless, the overall problem is much more complex and it involves, among others, solving the following subproblems:

1. Modeling the system functionality and performance constraints.

   System modeling refers to the *specification* problem of capturing important aspects of system functionality and constraints to facilitate design implementation and evaluation. Most hardware description languages attempt to describe a system functionality as a set of computations performed by a computing element and as interactions among computing elements. Among the important issues relevant to mixed system designs are:

   - explicit or implicit concurrency specification
   - communication model used: shared memory versus message-passing based
   - control flow specification or *scheduling* information

   There is a relationship between concurrency specification and the *natural* partitions in the system descriptions. Typically, languages that contain explicit partitioning via control flow breaks, find it difficult to specify concurrency explicitly. Concurrency information is then obtained by performing dependency analysis whose complexity depends on the communication model used. We consider the relevant modeling issues in Section 3.

2. Choosing granularity of the hardware-software partition.

   The system functionality can be partitioned either at the functional abstraction level where a certain set of *high-level* operations is partitioned or at the process communication level where a system model composed of interacting process models is mapped onto either hardware or software at the process description level. The former attempts fine grain partitioning while the latter attempts a *high-level library binding* through coarse-grain partitioning.

3. Determination of feasible partitions into application-specific and re-programmable components.

   The so-called problem of *hardware-software partitioning*. This delineation is influenced by issues such as analog interfaces that require a specialized hardware interfaces. However, for operations that can be implemented either in hardware or in software, the problem requires a careful analysis of flow of data and control in the system model.

4. Specification and synthesis of the hardware-software interface.

5. Implementation of software routines to provide real-time response to concurrently executing hardware modules.

6. Synchronization mechanisms for software routines and synchronization between hardware and software portions of the system.

This report attempts to outline major issues and suggests approaches to solving them. This report is organised as follows. In Section 2.1 we present a description of different system architectures based on types of hardware, software components used. We then describe features and limitations of the system architecture that is target of current approach towards system synthesis. Section 3 describes the modeling of system functionality. In particular, we describe how the input to our synthesis system is described in a hardware description language and its model based on flow graphs. Section 4 defines the problem of system partitioning. We discuss issues relating to performance characterization of hardware and software components and partitioning cost metrices based on which a partitioning algorithm is presented. Sections 5 and 6 present problems and solutions in implementation of hardware and software components, respectively. We introduce the notion of *threads* as a linearized set of operations. The software component is composed a set of concurrent and hierarchical threads. Section 7 discusses issues in system synchronization, how synchronization is achieved between heterogeneous components of system design. In section 8 we present design network coprocessor and summarize the results of hardware, software tradeoffs. Section 9 presents a summary of the main issues in system synthesis.

1.3 Applications

Among the potential applications of the techniques presented in this report are:

1. **Design of cost-effective systems:** The overall *cost* of a system implementation can be reduced by the ability to use already available general purpose re-programmable components while reducing the number of application-specific components.

2. **Rapid prototyping of complex system designs** - a complete hardware prototype of a complex system is often too big to be implemented except in a semi-custom implementation. With the identification of time critical hardware section, the total amount of hardware to be synthesized may be reduced significantly, thus making it feasible for rapid prototyping. A feasible partition that shifts the non performance-critical tasks to software programs can be used to quickly evaluate the design.

3. **Speedup of hardware emulation software** - During their development phase, many system designs are often modeled and emulated in software for test and debugging purposes. Such an emulation can be assisted by dedicated hardware components which provides a speedup on the emulation time.

Rapid prototyping and hardware emulation are two opposite ends of the system synthesis objective. Rapid prototyping attempts to minimize the application-specific component to reduce design time whereas hardware emulation attempts to maximize the application-specific component to realize maximum speed-up.

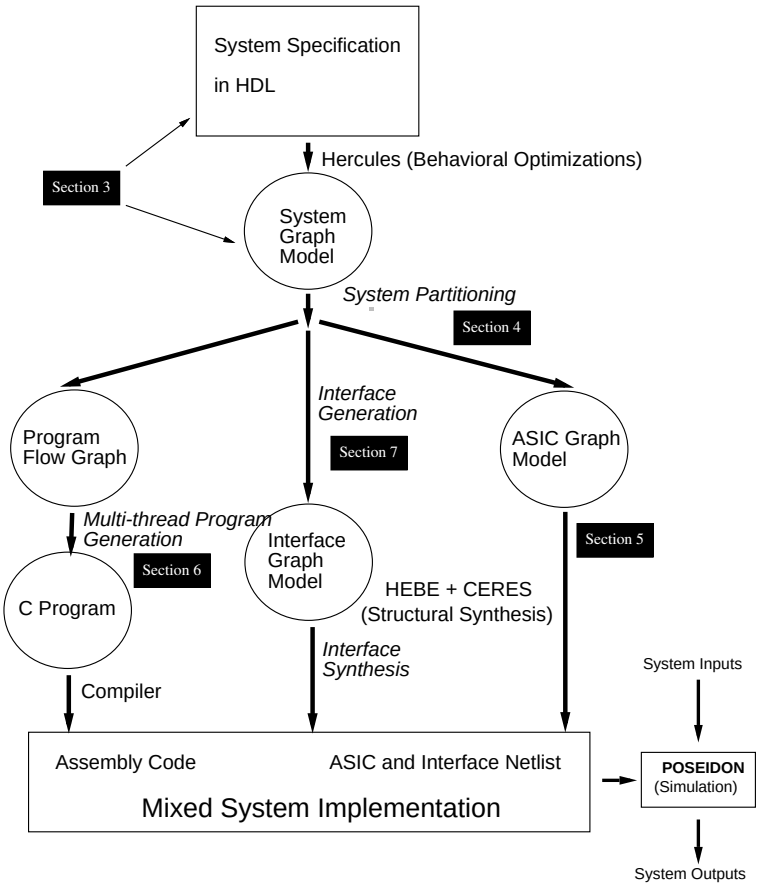


Figure 3: *System Synthesis Procedure*

Figure 3 shows organization of the CAD design system used for synthesis of mixed system designs. The input to our synthesis system is an algorithmic description of system functionality described in a hardware description language (HDL). The HDL description is compiled into a **system graph model** based on data-flow graphs described in Section 3. The system graph model is subject to system partitioning and hardware and software generation schemes as described in sections 4 through 6. Section 7 discusses mechanisms for synchronization between hardware and software. The resulting mixed system design consists of an assembly code for the software component, and a gate-level description of the hardware and hardware-software interface. This heterogenous description can be simulated by a program *Poseidon* that is described elsewhere [6].

2 System Architectures Based on Hardware-Software Components

In colloquial terms most digital systems can be classified as being either *reprogrammable* or *embedded*. Reprogrammable digital systems contain some form of storage that can be altered (reprogrammed) by the user under software control. On the other hand, the embedded systems are usually hardwired for certain

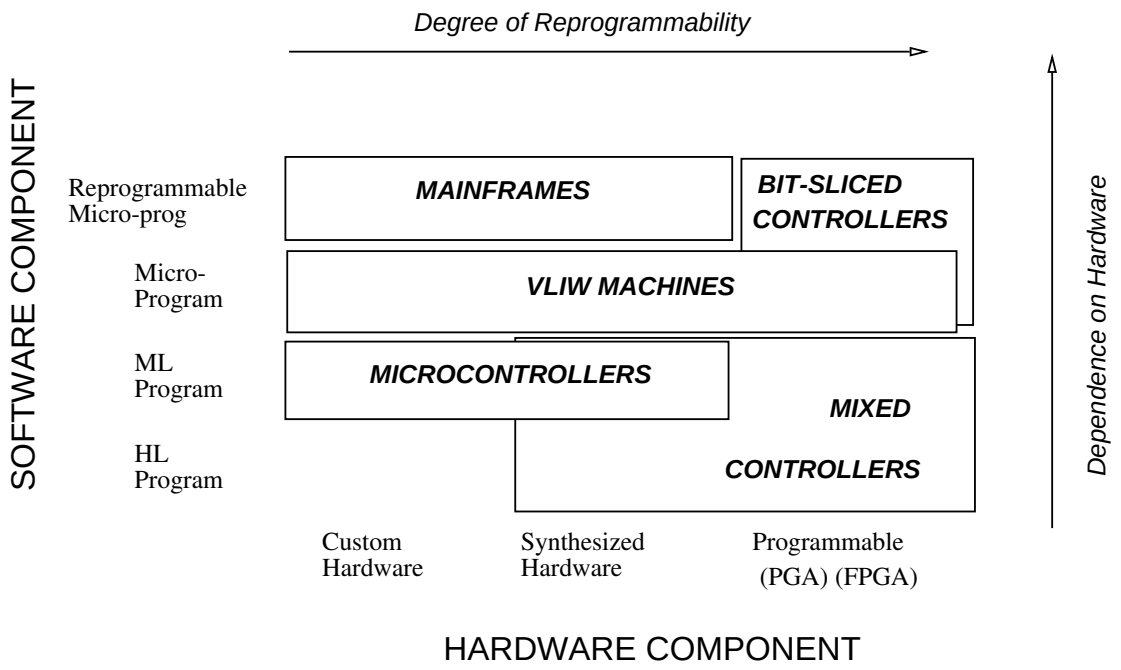


Figure 4: *System Classification Based on HW/SW Components*

specific tasks which can not be altered without changing the underlying hardware. Most reprogrammable digital systems contain one (or more) general-purpose microprocessor and structured memory components. Since embedded systems are optimized for certain specific tasks, the degree of 'reprogrammability' varies from none to changing parameters of some existing sequential control. An embedded system may have a dedicated controller (a sequencer) or a microcontroller programmed to sequence operations. Most of these systems contain storage (program or data) which is relatively small and can not be easily altered. Microcontrollers are essentially general purpose microprocessors with on-board memory for program and data storage. The ability to reprogram a digital system is related to the versatility of primitive operations, or the instruction-set of the microprocessor or microcomputer used in the system. In our terminology we refer to a microprocessor or a microcontroller as a *reprogrammable component* or simply as a *processor*. The specific set of instructions needed for a particular application to be executed by the reprogrammable component is referred to as the software component. Thus, in broad terms, a digital system can be thought of consisting of two components: *software* as a program in an on-board RAM or ROM and *hardware* as the underlying interconnection of special-purpose blocks. Based on this distinction, Figure 4 shows compositions of some familiar systems. The hardware component in a system design may be custom-designed as in most general purpose machines, or program-generated (programmed), or programmable as in programmable gate array designs. The software component of a system may consist of microcoded routines, or machine-level programs used in embedded control systems or high-level programming used in special-purpose machines. It is important to note that some system designs use microcoding simply as a technique for implementation of hardware control. For example, general purpose microprocessors

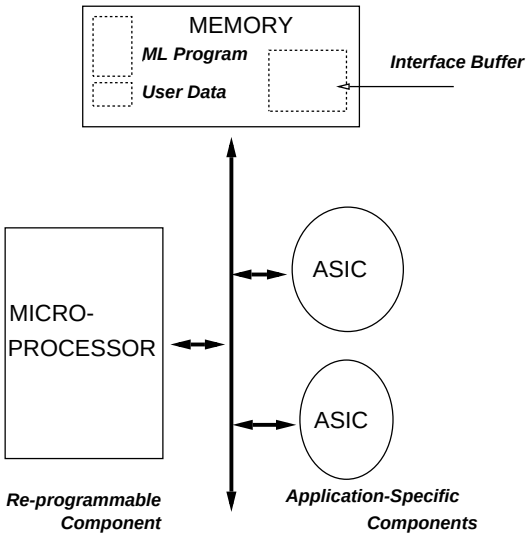


Figure 5: *Target System Architecture*

use microcoding simply as a design technique. This is different from the software *necessary* to achieve system functionality as in microprogramming of functional algorithms in case of mainframe machines.

Conventionally, machine-level and high-level programs manipulate user data-structures while micro-programs manipulate hardware resources. In case of mixed controller designs proposed in this paper, however, we use machine-level programs to perform both activities. The objective of this research is design of mixed controllers shown in Figure 4. These systems use a reprogrammable compoent to achieve part of the system functionality which may be time-constrained as in the case of embedded systems.

2.1 Target System Architecture

We choose a target architecture that contains the essential elements of hardware-software systems. In Figure 5, the target architecture consists of a general-purpose processor assisted by application-specific hardware components. The following lists the relevant assumptions relating to the target architecture.

- We restrict ourselves to use of a *single re-programmable component* because presence of multiple re-programmable components requires additional software synchronization and memory protection considerations to facilitate safe multiprocessing. Multiprocessor implementations also increase the system cost due to requirements for additional system bus bandwidth to facilitate inter-processor communications. We make this simplifying assumption in order to make the synthesis tasks manageable.

- The memory used for program and data-storage may be on-board the processor. However, the interface buffer memory needs to be accessible to the hardware modules directly. Because of the complexities associated with modeling hierarchical memory design, so far we considered the case where all memory accesses are to a single level memory, i.e., outside the re-programmable

component. The hardware modules are connected to the system address and data busses. Thus all the communication between the processor and different hardware modules takes place over a shared medium.

- The re-programmable component is always the bus master. Almost all re-programmable components come with facilities for bus control. On the other hand, inclusion of such functionality on the application-specific component would greatly increase the total hardware cost.
- All the communication between the re-programmable component and the ASIC components is done over named channels whose width (i.e. number of bits) is same as the corresponding port widths used by read and write instructions in the software component. The physical communication takes place over a shared bus. The problem of encoding and sharing multiple virtual channels over a physical bus is a subject of continuing research at Stanford [7].
- The re-programmable component contains a 'sufficient' number of maskable interrupt input signals. For purposes of simplicity, we assume that these interrupts are unvectored and there exists a predefined destination address associated with each interrupt signal.
- The application-specific components have a well-defined *RESET* state that is achieved through system initialization sequence.

It is important to note that the final system implementation may or may not be a single-chip system design depending on availability of the re-programmable component either as a macro-cell or as a separate chip. Further, the approach outlined in this report can also be used for alternative target architectures.

3 Specification and Modeling of Hardware-Software Systems

Currently most behavioral system specifications are derived from the corresponding algorithmic descriptions of the system functionality. The algorithmic descriptions are usually described in a procedural language like C or Pascal. Consequently, the hardware behavioral descriptions tend to use a procedural language like VHDL, Verilog etc. However, when describing hardware in a procedural language (that is, as a program), one is often faced with the difficulty of representing an essentially concurrently-executing set of operations in a linear code The linear-code representation inherently assumes existence of a single thread of control and static data storage. However, hardware execution is usually multi-threaded and is driven by availability of appropriate data. In contrast to *instruction-driven* single-threaded linear-code representation, data-flow graphs provide a *data-driven* representation that can model multiple-threads of execution (Figure 6). Therefore, the hardware for embedded controllers and non-recursive DSP algorithms is more appropriately represented by data-flow (DF) graphs instead of linear code used for algorithmic description. To avoid this dichotomy of behavioral representation, most high-level synthesis algorithms operate on an intermediate form that accurately reflects the concurrent nature of hardware. Most hardware intermediate forms used for high-level synthesis tend to be similar to data-flow graphs [8] [9] [10] [11].

Generally, any sequence of machine instructions can be represented by a machine-level data-flow graph. Indeed the expression-evaluation trees generated by compilers (before the code-generation stage)

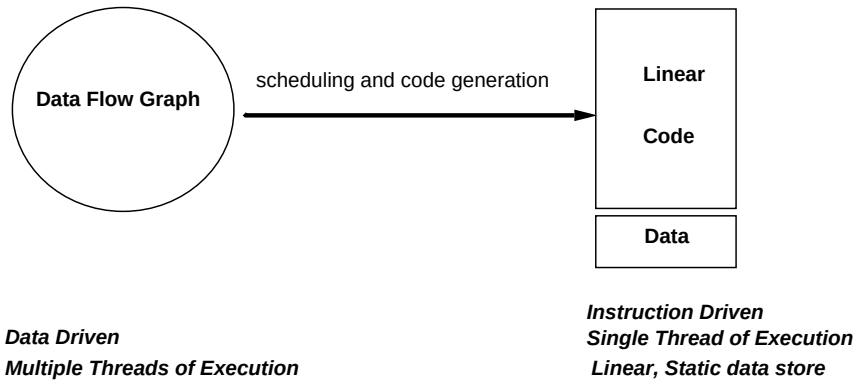


Figure 6: *Linear Code versus Data-Flow Graph Representations*

are a form of data-flow graphs. However, these data-flow graphs consisting of operations described at the level of machine instructions decrease the specification granularity significantly to make them useful for system partitioning in to hardware and software components. Therefore, data-flow graphs in our context are described using operations available at the language specification level. These have also been referred to as macro-data flow graphs elsewhere [12].

From data-flow representations we can generate an equivalent sequence of instructions by scheduling various operation vertices in the data-flow graph. Operation scheduling techniques are important even in case of a single thread of execution where static memory requirements are affected by scheduling even though all schedules result in same overall latency (see Section 4.2) . Latency optimality in scheduling is realized by exploiting parallelism in instruction stream which requires multiple execution threads. We consider the algorithms for evaluating data-flow graphs and their equivalent linear-code representations in section 4.2.

3.1 System specification using *HardwareC*

We specify system functionality in an hardware description language called, *HardwareC* [13]. *HardwareC* follows much of the syntax and semantics of the programming language, C with modifications necessary for correct and unambiguous hardware modeling. Like C, the primitive operation in *HardwareC* consists of an assignment operation with a procedural call being the means of abstraction of sub-specifications. Procedural calls correspond to modular specification of different components of the hardware. No recursive calls of any form are allowed. A *HardwareC* specification consists of *blocks* of statements which are identified by enclosing parentheses. The blocks are structured, thus no two blocks are overlapped partially. That is, given any two blocks, they are either disjoint or one is contained by the other block. Like C, no nested procedure declarations are allowed. Therefore, any variable that is non-local to any procedure is non-local to all procedures. Local variables are scoped *lexically* with the most-closed nested rule for structured blocks.

A process in *HardwareC* executes concurrently with other processes in the system specification. A process restarts itself on completion of the last operation in the process body. Thus there exists an

implied outer-most loop that contains the body of the process model. In other languages, this loop can be specified by an explicit outer loop statement. Operations within a process body need not be executed sequentially (as is the case in a process specification in VHDL, for example). A process body can be specified with varying degrees of parallelism such as parallel (<>), data-parallel ({}) or sequential ([]).

In addition, *HardwareC* allows specification of *declarative* model calls as blocks that describe physical connections and structural relationships between called models. For hardware modeling purposes, both timing and resources constraints are allowed in the input specifications. Timing constraints are specified as min/max delay attributes between labeled statements where as resource constraints are specified as user-specified bindings of process and procedure calls to specific hardware model instances.

Example 3.1 describes a simple process specification in *HardwareC*.

Example 3.1. Example of a simple *HardwareC* process

```
process simple (a, b, c)
  in port a[8], b[8] ;
  out port c[8] ;
{
  boolean x[8], y[8], z[8] ;

  <
  x = read(a);
  y = read(b);
  >

  z = fun(x , y);
  write c = z;
}
```

This process performs two synchronous read operations in the same cycle, followed by a function evaluation and a write operation. Note that specification of explicit parallelism by (<>) delimiters is redundant here since there exists an implicit parallelism between the two read operations, thus a data-parallel grouping ({}) would yield the same execution results. □

There is no explicit delay associated with individual assignment statements (except in case of explicit register/port load operations as mentioned later). An assignment may take zero or non-zero delay time. However, multiple assignments to same variable can either be interpreted as (a) last assignment or (b) an assignment after some delay. Resolution of which policy (a) or (b) to be used is performed by a reference stack [14]. Reference stack performs variable propagation by instantiating values of the variables in the right-hand side of the assignments. In case of identified storage elements (b) is adopted where 'some delay' corresponds to delay of 'at least' one cycle time. In addition, this policy can also be enforced on some assignments by an explicit 'load' prefix that assigns a delay of precisely one cycle time to the respective assignment operation.

3.1.1 Memory and Communication

HardwareC allows specification of shared memory within a process model. All the communication within a process model is based on the shared memory specified within the model, because it is relatively straight-forward to ensure ordering of operations within a given process model to ensure integrity of

memory shared between operations in the model. However, consistency of memory shared across concurrently executing models must be ensured by the models themselves. *HardwareC* allows specification of *blocking* inter-model communications based on message-passing operations. As with the shared memory variables, the only data-types available for channel is a fixed-width bit-vector. Integers are coded using 2's complement representation.

Use of message-passing operations simplifies the specification of inter-model communications. It should be noted, however, that it is easy to implement a message-passing communication using memory shared between respective models (the converse is not true, however). Indeed, during system partitioning, reductions in communication overheads are realized by simplifying the inter-model communication as discussed in later sections.

3.1.2 Nondeterminism in System Specifications

Non-determinism in our system models is caused either by *external* synchronization operations or by *internal* data-dependent delay operations, like conditionals and data-dependent loops. External synchronization operations are related to blocking communication operations, whereas operations likes data-dependent loops present variable and unknown execution delays. Example 3.2 below shows a *HardwareC* process description containing 3 unbounded/unknown delay operations: message-passing receive operation, conditional and loop.

Example 3.2. Example of a *HardwareC* process with unbounded delay operations

```
process example (a, b, c)
  in port a[8] ;
  in channel b[8] ;
  out port c ;
{
  boolean x[8], y[8], z[8] ;

  x = read(a);
  y = receive(b);
  if (x > y)
        z = x - y ;
  else
        z = x * y ;
  while (z >= 0) {
        write c = y ;
        z = z - 1 ; }
}
```

`read` refers to a synchronous port read operation that is executed unconditionally as a value assignment operation from the wire or register associated with the port `a`. `receive` is a message-passing based read operation where the channel `b` carries additional request and acknowledge control signals that facilitate a *blocking* read operation on based on availability of data on channel `b`. □

3.2 System Model

Broadly speaking, there are two major ways of modeling and analyzing the system behavior:

process-based modeling where logical and temporal properties of processes and their constituent events define the behavior of a system, and axiomatic techniques, similar to theorem-proving in proof systems, are applied to verify correctness of system transformations [15]. Relevant logical properties of the system behavior are expressed by assertions about liveness and safety. Each liveness property indicates that the assertion (about state of the system) will eventually hold. Each safety property states that the assertion (on the system state) will always be true. Most common method to specify and verify such assertions is by adding time variables to the computation model [16] [17]. Common examples of this approach are proof-systems for CSP programs [18], distributed programs [19].

graph-based modeling which uses techniques from graph theory to build the system model. The main difference with the process-based modeling is in explicit expression of dependencies between processes and constituent operations.

We model system behavior using a graph representation based on flow graphs [20]. As described later in this section, timing and resource constraints are represented on graphs compatible with the operation graph models.

Definition 3.1 *A system model, $\mathcal{M}$, is represented by a 3-tuple consisting of operation graph Φ, timing constraint , $\mathcal{T}$, and resource constraint, $\mathcal{R}$.*

$$\mathcal{M} = (\Phi, \mathcal{T}, \mathcal{R})$$

The operation graph models capture system functionality as a set of control-data-flow graphs. Timing constraints are specified using *compatible* weighted graph models. Resource constraints are used to specify bounds on types and number of data-path resources available for synthesis of $\mathcal{M}$ into hardware as well as the amount of static storage available for synthesis of $\mathcal{M}$ into software. Hardware resource constraints are important for hardware synthesis and are briefly mentioned in Section 5.

Definition 3.2 *An operation graph model, Φ, consists of a set of acyclic sequencing graph models:*

$$\Phi = \{G_1, G_2, \ldots, G_n\}$$

where a sequencing graph model, G_i, represents body of an iterative construct in the hardware description language model.

The iterative constructs are of two types:

- **Counting loops** have an explicit or an implicit *repeat count*, r.
- **Non-counting loops** wait on some *external* conditions on a wire or a channel. For example, a *HardwareC* statement like

  `while(wire_name);`

  is semantically equivalent to `wait(!wire_name);`. These loops model some asynchronous event in the behavior model and are required for correct modeling of reactive hardware behavior. The corresponding hardware for such loops is implemented by means of asynchronous set/reset inputs to the storage elements. In software, however, such operations can be implemented either as interrupts to on-going computation or as polled operations.

All sequencing graph models are assumed to model counting loop operations with a finite or infinite repeat count. Non-counting loop operations are modeled as individual wait operations. We make further distinctions between characteristics of counting loop constructs when considering issues in performance characterization of the software component.

A sequencing graph model, G, that models a *process* specification in *HardwareC* is a model with infinite repeat count, that is, on completion of its last operation (sink), it restarts itself unconditionally. Among the sequencing graph model of an operation graph model, Φ, there are two kinds of hierarchies induced:

- **structural hierarchy** denoted by relation, S, that denotes structural relationship between any two sequencing graph models,
- **calling hierarchy** G^* of a sequencing graph model, G, denotes the set of sequencing graph models that are on the control-flow hierarchy of G, that is, models that are called or used by G.

An operation graph model may consist of structural connection of one or more than one sequencing graph models, each of which contains a called hierarchy. A sequencing graph model that is common to two calling hierarchies is considered a **shared model** or a shared resource.

Definition 3.3 *A sequencing graph model is a polar acyclic graph* $G = (V, E, \delta, \chi, S)$ *where* $V = \{v_0, v_1, \ldots, v_N\}$ *represent operations with* v_0 *and* v_N *being the source and sink operations respectively. The edge set,* $E = \{(v_i, v_j)\}$ *represents dependencies between operation vertices. An integer weight* $\delta(v_i), \ \forall\, v_i \in V$ *represents execution delay of operation associated with vertex,* v_i*. Function,* $\chi : E \mapsto Z$ *defines condition index of a given edge. In case of edges incident from a condition vertex or incident on a join vertex, these a condition index refers to case value associated with the evaluated condition.* S *defines the storage common to operations in the graph model* G*.*

For sake of simplicity, a sequencing graph model, G, is often expressed as $G = (V, E)$. An edge, $(v_i, v_j) \in E(G)$ induces a precedence relationship between vertices v_i and v_j and it is also indicated by $v_i > v_j$. Relation $>^*$ indicates transitive closure of the precedence relation. The transitive closure of a sequencing graph model, G, under precedence relation is denoted by $G^>$. Note that the sequencing model defined here are similar to the the SIF model defined in [14]. There are, however, some differences in representation of conditional and wait operations.

The sequencing graph model captures the operation concurrency and data-dependent delay operations. Overall, the sequencing graph model consists of concurrent data-flow sections which are ordered by control flow. The graph edges represent dependencies while branches indicate parallelism between operations. The data-flow sections preserve the parallelism while control constructs like conditionals and loops obviate the need for a separate description of the system control flow. The control operations like loops are specified as separate subgraphs by means of hierarchy. The computational semantics of the sequencing graph model is as follows: an operation in the data-flow graph is enabled for executions once all the input data are available. We maintain the strict FIFO order of operations during successive invocations of the graph model by imposing the additional constraint that a source vertex is reinvokes only after the corresponding sink vertex has been executed. This requirement avoids need for conventional

token-matching schemes needed for execution of pure data-flow graphs [21]. Figure 7 shows an example of the graph model corresponding to `process simple` described in Example 3.2.

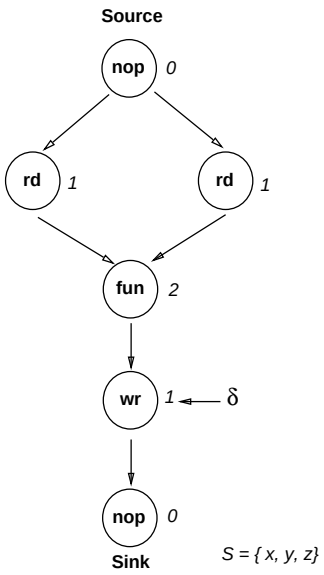


Figure 7: *Example of a sequencing graph model*

Remark 3.1 *Storage, S, is defined for correct behavioral interpretation of the graph model, G. S is independent of cycle-time of the clock used to implement the corresponding synchronous circuitry and does not include storage specific to structural implementation of G (for example, control latches). Further, S, need not be the* minimum *storage required for correct behavioral interpretation of a sequencing graph model.*

An operation vertex is classified as a *simple* or *complex* vertex depending on the operation performed by the vertex. Simple vertices consist of a single operation whereas complex vertices consists of a set of operations that are represented by a called sequencing graph model. Thus complex vertices induce hierarchical relationships between sequencing graph models. A call vertex enables execution of the sequencing graph corresponding to the procedure call. A loop vertex iterates over the graph body of the loop until its exit condition is satisfied. Table 1 lists operation vertices used for describing the sequencing graph models.

Remark 3.2 *The sequencing graph is* **acyclic** *because the* HardwareC *descriptions are required to be structured and looping constructs are represented as separate graphs.*

Data-dependent and synchronization operations introduce uncertainty over the precise delay and order of operations in the system model and thus make its execution *non-deterministic* [22]. We refer to a vertex with data-dependent delay as a *point of non-determinism* in the system graph model.

Type	*Operation*	*Description*
Simple	no-op	No operation
	load	Load register
	cond	Conditional fork
	join	Conditional join
	wait	Wait on a signal
	op-logic	Logical operations
	op-arithmetic	Arithmetic operations
	op-relational	Relational operations
	op-io	I/O operations
Complex	call	Procedural call
	loop	Iteration
	block	Declarative block

Table 1: *Sequencing graph operation vertices*

Definition 3.4 *An operation vertex in G with an unbounded execution delay is known as an* **anchor** *vertex.*

A vertex representing **wait** operation is example of an anchor vertex. The execution delay, $\delta(v)$ of an anchor vertex may not be known statically and may take any nonnegative integer value from 0 to ∞. By definition, source vertex, v_0, of a sequencing graph is considered an anchor vertex.

Definition 3.5 *A sequencing graph* **delay function**, d, *returns a non-negative delay of a graph model, G following a bottom-up computation as follows:*

1. *Delay of a non-anchor vertex is the execution delay of the operation, $d(v_i) = \delta(v_i)$,*
2. *Delay of an anchor vertex is set to zero,*
3. *Delay of a sequencing graph model, G, is the delay of the longest path in G,*

$$d(G) = \max d(p_{0N}) = \sum_{v_i \in longest_path(G)} d(v_i)$$

   where a path,p_{ij}, in G consists of an ordered set of vertices in $V(G)$, $p_{ij} = (v_i, v_{i+1}, \ldots, v_j)$.
4. *Delay of a complex vertex is set to zero, that is,*

$$d(v_{loop}) = d(v_{call}) = d(v_{block}) = 0$$

5. *Delay of a conditional vertex is maximum of delay over each of its branches. A conditional branch is defined by a directed path from the condition vertex to the corresponding join vertex.*

Definition 3.6 *A sequencing graph* **latency function**, λ, *returns a non-negative latency of a graph model,* G *following a bottom-up computation as follows:*

1. *Latency of a vertex is the execution delay of the operation represented by vertex,*
2. *Latency of a sequencing graph model is execution delay of the sequencing graph model, that is, the time period from execution of its source vertex to the execution of its sink vertex.*
3. *Latency of complex vertices is the latency of the corresponding called sequencing graph models, that is,*

$$\begin{aligned} \lambda(v_{loop}) &= r_l \cdot \lambda(G_{loop}) \\ \lambda(v_{call}) &= \lambda(G_{call}) \\ \lambda(v_{block}) &= \lambda(G_{block}) \end{aligned}$$

where r_l *is the repeat-count of the loop operation* v_{loop}.

Note that latency of a sequencing graph is a function of operation scheduling. On the other hand, the delay function represents the longest path delay of sequencing graph.

3.2.1 Communication

For all operations with in in a graph model, $G = (V,\ E,\ \delta,\ S)$ all the communication is based on shared storage, S. Inter-model communications are represented by I/O operation vertices which, on execution, may alter the model storage, S. An I/O operation vertex may encapsulate a sequence of operations which is referred as a *communication protocol*. A communication protocol may be *blocking* or *non-blocking*. A non-blocking protocol may also be finitely *buffered*.

A blocking communication protocol is is expressed as a sequence of simpler operations on ports and additional control signal to implement the necessary handshake. For example, to implement a blocking read operation on a channel '`c`' additional control signals '`c_rq`' and '`c_ak`' would be needed as shown in the Example below.

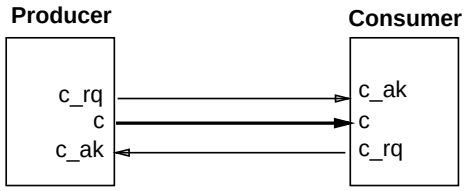
Example 3.3. A blocking read operation.

```
bread(c)          =>      [
                          write c_rq = 1;
                          wait(c_ak);
                          < read(c);
                          write c_rq = 0; >
                          ]
```

□

While it is easy to connect two blocking or two non-blocking read-write operations, connection of two disjoint read/write operations on a channel requires handling of special cases. For example, consider a connection between blocking read and non-blocking write operation below.

Example 3.4. Blocking/Non-blocking channel connections.



Blocking read and non-blocking write

```
     Blocking read                    Non-blocking write

[                                     [
write c_rq = 1;                       write c_rq = 1;
wait(c_ak);                           < write c = value; write c_rq = 0; >
< read(c); write c_rq = 0; >          ]
]
```

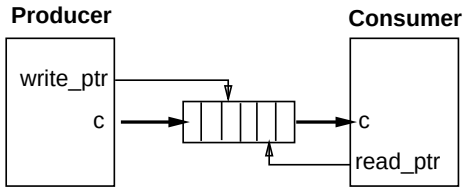
Blocking write and non-blocking read

```
      Non-blocking read               Blocking write

[                                     [
write c_rq = 1;                       write c_rq = 1;
< read(c); write c_rq = 0; >          wait(c_ak);
]                                     < write c = value; write c_rq = 0; >
                                      ]
```

A non-blocking/non-blocking read/write connection results in one cycle read and write operations. However, a blocking/non-blocking connection requires two clock cycles for the non-blocking operation. □

A **buffered communication** is facilitated by a finite-depth interface buffer with corresponding read and write pointers. The communication protocol consists of I/O operation as well as manipulation of the read, write pointers as shown by the example below.

Example 3.5. Buffered communication protocol.




```
[                                     [
read (buff[read_ptr]);                write buff[write_ptr] = value;
read_ptr++ modulo N;                  write_ptr++ modulo N;
]                                     ]
```

Under normal operation, `read_ptr` $\neq$ `write_ptr`. Violation of this condition indicates either a buffer is full or empty depending on whether the increment of `write_ptr` causes violation or the increment of `read_ptr` causes the violation. □

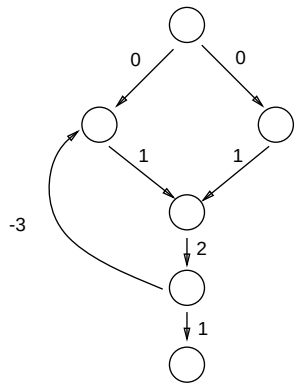


Figure 8: *The Constraint Graph Model*

3.3 Specification of Timing Constraints

Timing constraints are of two types: (a) minimum/maximum timing separation between pairs of operations and (b) system input-output rate constraints. Timing constraints between operations are indicated by tagging the corresponding operations. The input(output) rate constraints refer to the rates at which the data is required to be consumed (produced). The rate constraints refer to min/max time constraints on multiple executions of the same input or output operation.

Let us first consider the timing constraints of the first type, that is, the min/max timing constraints. Min/max timing constraints between operations are specified by tagging the corresponding assignment operations in the *HardwareC* specifications. Example 3.6 shows an example of a min/max timing constraint.

Example 3.6. Specification of min/max timing constraints by statement tagging

```
process simple (a, b, c)
  in port a[8], b[8] ;
  out port c[8] ;
{
  boolean x[8], y[8], z[8] ;
  tag A, B;

  <
A: x = read(a);
  y = read(b);
  >

  z = fun(x , y);
B: write c = z;

  constraint maxtime from A to B = 3 cycles;
}
```

□

These timing constraints are abstracted in a constraint graph model [14] shown in Figure 8. In the constraint graph model vertices represent operations and edges indicate timing constraints between

vertices. The edges are weighted by the timing constraint value. A positive value implies a minimum timing constraint, a negative value implies a maximum timing constraint.

Let $T(v_i)$ represent *start time* of operation v_i. A *minimum timing constraint*, $l_{ij} \geq 0$ from operation vertex v_i to v_j is defined by the following relation between the start times of the respective vertices:

$$T(v_j) \geq T(v_i) + l_{ij}$$

Similarly a *maximum timing constraint*, $u_{ij} \geq 0$ from v_i to v_j is defined by the following inequality:

$$T(v_j) \leq T(v_i) + u_{ij}$$

Definition 3.7 *The timing constraint graph model, G_{T0} is defined as $G_{To} = (V, E, \Omega)$ where the set of edges consists of* forward *and* backward *edges, $E = E_f \cup E_b$ and $\omega_{ij} \in \Omega$ defines the weights on edges such that $T(v_i) + \omega_{ij} \leq T(v_j)$.*

3.4 Data Rate Constraints

Each execution of an input(output) operation consumes(produces) a *sample* of data. An input/output data interval is defined in terms of *cycles/sample* as the interval between successive input/output operations. Corresponding data rates are defined by inverse of the data interval. Input/output data rates are a function of time.

A *minimum data rate constraint*, ρ_m, on an input/output operation defines the lower bound on the interval between any successive executions of the corresponding operation. Similarly, a *maximum data rate constraint*, ρ_M, on an I/O operation defines the upper bound on the time interval between successive executions of the operation. Thus, the rate constraints refer to time constraints on multiple executions of the same input or output operation. These constraints can be expressed as min/max timing constraints on *unrolled* constraint graph models. As an example, constraint graph model G_{T1} shown in Figure 9 consists of two sequential executions of the sequencing graph model G. The rate constraint on consecutive read operations is shown as a maximum timing constraint between two read operations in G_{T1}.

For an I/O operation $v \in V(G)$, a data-rate constraint of ρ_M samples/cycle can be translated into a constraint on latency of G. Graph G either models a process or body of a loop operation. If G models a process then execution of G restarts itself on completion. Since polarity of G guarantees that there is one execution of v for every execution of G. Therefore, latency, $\lambda(G) \leq \frac{1}{\rho_M}$ cycles. If G models body of a loop operation $v_l \in V(G')$ then

$$\lambda(G') = \lambda(v_l) + d' = r_l \cdot \lambda(G) + d'$$

where d' (≥ 0) is the difference in latencies of G' and v_l. Thus a constraint on $\lambda(G)$ can be translated into a constraint on latency of the corresponding process graph model, G', if the loop repeat count, r_l, can be constrained.

In case of nested loop operations, rate constraints are indexed by corresponding loop operations. The loops are indexed by increasing integer numbers. The inner-most loop is indexed 0. In the Example 3.7 below there are two rate constraints on the read operation with respect to the two while statements.

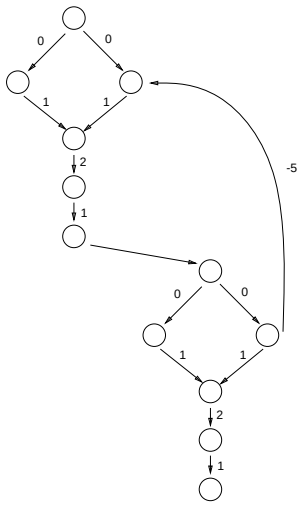
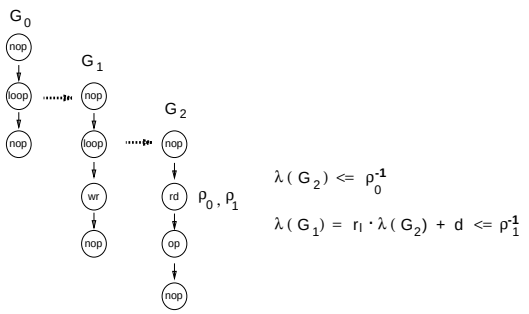


Figure 9: *Specification of rate constraint as a min/max timing constraint*

Example 3.7. Specification of rate constraints in presence of nested loop operations.

```
process example (frameEN, bitEN, bit, word)
   in port frameEN, bitEN, bit;
   out port word[8];
{
   boolean store[8], temp;
   tag A;

   while (frameEN)
   {
       while (bitEN)
       {
A:             temp = read(bit);
               store[7:0] = store[6:0] @ temp;
       }
       write word = store;
    }
    attribute "constraint maxrate 0 of A =   1 cycles/sample";
    attribute "constraint maxrate 1 of A =  10 cycles/sample";
}
```



r_l denotes repeat-count of loop modeled by G_2 and d is the difference in latencies of G_1 and operation v_{loop} in G_1. □

4 The Problem of Hardware-Software Partitioning

We restrict our attention to partitions that are functional in nature so that the sequencing graph models are at the granularity of operations rather than machine instructions. Use of functional partitions helps in operation scheduling since an operation can only be scheduled when all its inputs are available. We make a distinction between *homogenous* and *heterogenous* partitions of a given system model based on partitioning objectives. The objective of homogenous partitioning is to partition a given hardware description into minimal number of hardware blocks each of which is smaller than a given size constraint. This partitioning is performed under overall area, pinout and latency constraints. The homogenous partitioning problem is attacked and solved by previous research [23]. The objective of heterogenous partitioning problem is to partition the system model for implementation into hardware and software components. Heterogenous partitioning can be thought of as a generalization of the homogenous partitioning problem with re-programmable components as being 'generalized resources'. However, there are inherent differences in the model of computation used for implementation of hardware and software models. The software component implements model functionality as an *instruction-driven* computation with a statically allocated memory space. On the other hand, hardware components essentially operate as *data-driven* reactive components. Further, due differences in primitive operations in hardware and software components, the two computations proceed at very different instantaneous rates. Because of these differences in the models and rates of computation used by hardware and software components, it is necessary to allow multiple executions of individual hardware and software models with respect to each other to achieve high system throughput. Further, the difference in rates of computations causes variations in the rates of communication between hardware and software components and thus entail a higher communication overhead due to necessary handshake and buffering mechanisms.

Given two sets of operations, their execution is termed *single-rate* if for each execution of an operation in one set there is only one execution of all operations in the other set. Correspondingly, an execution is termed *multi-rate* if for each execution of an operation in one set, there are more than one executions of an operation in the other set. Note that operations in a sequencing graph model, G, execute at a single rate. On the other hand, executions of sequencing graph models may be multi-rate. For the reasons described before, we would like to achieve a multi-rate execution of hardware and software models. Thus a partition of a sequencing graph model must be transformed to allow multi-rate executions of the partitioned graphs by choice of suitable inter-parition communication mechanisms (bufferring, for example). In this context, the problem of hardware-software partitioning is formulated as a problem of partitioning of operation graph models, Φ, instead of partitioning of the sequencing graph models, G. Because system partitioning is strongly influenced by the choice of the target processor and program implementation techniques, we first present a model of the processor and our approach to software characterization.

4.1 Processor Model

The target processor is represented by a cost model, $\mathit{\Pi} = (\tau_{op},\ \tau_{ea},\ t_m,\ t_i)$ where

- execution time function, τ_{op}, represents assembly instruction delay times in cycles,
- address calculation delay function, τ_{ea}, represents effective address calculation delay times in cycles,

Mode	*Notation*	*Explanation*
immediate	#4	value = 4
register	R1	value = [R1]
direct	(100)	value = mem[100]
register indirect	(R1)	value = mem[[R1]]
memory indirect	@(R1)	value = mem[mem[[R1]]]
indexed	100(R1)(R2)	value = mem[100+[R1]+d*[R2]]

Table 2: *Addressing Modes*

- memory access times t_m in cycles is the time for a memory access,
- interrupt response time, t_i, is the maximum time between activation of an external interrupt and beginning of execution of the corresponding interrupt service routine.

The execution time function, τ_{op}, maps *assembly instructions* to positive integer delays. The assembly instructions are generated by the high-level language compiler. These instructions usually correspond to instructions supported by the processor instruction set. However, some assembly instruction may refer to a group of processor instructions. These *pseudo-assembly* instructions are sometimes needed for compilation efficiency and to preserve *atomicity* of certain operation in the sequencing graph model. Effect of internal hardware pipelining in microprocessors is modeled as follows. The function, τ_{op} represents pipelined operation delays (which is usually 1 cycle for operations with non-pipelined execution delays of less than number of pipestages, p). A penalty of $p-1$ cycles is added to the delay of the overall program. In addition, additional *pipeline stall* penalty is added for instructions with latencies greater than p (such as floating point instructions). The address calculation function, τ_{ea}, maps a memory addressing mode to integral delay (in cycles) encountered by the processor in computing the effective address. An addressing mode specifies an immediate data, register or memory address location. In the last case, the actual address used to access the memory is called the effective address.

Table 2 lists common addressing modes. Square brackets ([]) indicate contents, for example, [R1] indicates contents of register R1, mem[10] indicates memory contents at address 10. Not all the addressing modes may be supported by a given processor. For example, the DLX processor supports only immediate and register addressing modes, while the x86 instruction set supports all mentioned addressing modes (though with restrictions on which registers can be used in a certain addressing mode). The interrupt response time, t_i, is the time that processor takes to become aware of an external hardware interrupt in a single interrupt system (that is, when there is no other maskable interrupt is running).

A software implementation of a sequencing graph model, $G = (V, E)$ is characterised by a software size function, $\mathcal{S}_{\Pi}$, that refers to the size of program and static data necessary to implement the corresponding program on a given processor, Π. For a operation graph model, Φ, $\mathcal{S}_{\Pi}(\Phi) = \sum_{G_i \in \Phi} \mathcal{S}_{\Pi}(G_i)$.

Now the problem of system partitioning is stated as follows.

Problem P1: *Given a system model, $\mathcal{M} = (\Phi, \mathcal{T}, \mathcal{R})$, static storage constraint, $\overline{\mathcal{S}}$, and a processor cost model, Π, find a partition of operation graph model, $\Phi = \Phi_h \cup \Phi_s$ such that:*

1. *Φ_s satisfies the timing constraints $\mathcal{T}$,*
2. *software size, $\mathcal{S}_{\Pi}(\Phi_s) \leq \overline{\mathcal{S}}$ and*
3. *the number of sequencing graph models in Φ_s is maximized.*

4.2 Modeling of Software Performance

Software performance is characterized by two metrices: software delay and program/data size. Software delay can be computed by bottom-up computation of operation delays in the sequencing graph model. Data size is determined by the size of static storage required for correct execution of the program.

In order to make effective tradeoffs during partitioning, it is necessary to be able to make good estimates about software and hardware performance. Such estimations often make simplifying assumptions that tradeoff modeling accuracy against speed of estimations. In estimating software performance, we make the following assumptions.

1. The system bus is always available for instruction/data reads and writes.
2. All memory accesses are aligned. Misaligned memory accesses add additional cycles to memory access time, t_m.
3. All memory accesses are to a single-level memory.

Each operation, v, in the graph model is characterised by number of read accesses, n_r, number of write accesses, n_w and the number of assembly-level operations, n_o. Typically, n_w is 1. The software operation delay function, η, is computed as follows:

$$\eta(v) = \sum_{i=1}^{n_o} t_{op_i} + \sum_{i=1}^{n_r} m_i + \sum_{i=1}^{n_w} m_i$$

where the operand access time, m_i, is the sum of effective address computation time and memory access time for memory operands. Due to non-orthogonality of most common instruction set architectures, the execution time function of some operations is often slightly overestimated from real execution delays. The number of read and write accesses is related to the amount and allocation of static storage, $\mathcal{S}(G)$. Wait operations in a graph model induce a synchronization operation in the corresponding software model. Thus, the software delay of wait operations is estimated by the *synchronization overhead* which is related to the program implementation scheme being used. A synchronization operation causes a *context switch* in which the waiting program is switched out in favor of another program. It is assumed that the software component is computation intensive and thus the wait time of a program can be overlapped by active execution another program. As mentioned earlier, a wait operation can be implemented either as an interrupt operation or a polled operation. In case of an *interrupt-based* implementation, the synchronization delay is computed as follows:

$$\eta_{intr}(v) = t_i + t_s + t_o$$

where t_i is interrupt response time, t_s is interrupt service time, which is typically the delay of the service routine that performs input read operation and t_o is concurrency overhead (Section 6). In case of a *polled* implementation of a wait operation, the delay due to a wait operation is the delay in performing the corresponding read operation. However, the program implementation scheme enforces additional constraint on the *minimum polling time interval*, t_p, at which any port can be polled. t_p is a function of the size of Φ_s.

The determination of static storage, S, required for software implementation of a graph model is more complex. By static storage here we are chiefly concerned with the storage required to hold data transfers across assembly operations. One approach to determine the minimum set of variables required to implement a graph model, G, would be to serialize the graph model, G based on a scheduling of operations. Due a single processor target architecture, the cumulative operation delay of G would be constant under any schedule. However, the number of variables required to implement program of G would vary according to scheduling technique used. Most popular heuristics for code generation use a specific order of execution of successor nodes in order to reduce the size of S [24]. A variable *interference graph* can be built from the serialized model G. The minimum number of colors required to color vertices in the interference graph such that no two adjacent vertices have the same color gives the minimum number of variables required to implement G in software. The set of variables in $\min S(G)$ can be mapped to specific memory locations or the on-chip registers, since no aliasing of data items is allowed in input *HardwareC* descriptions. (Register storage of an aliased variable will lead to incorrect behavior due to possible inconsistency in values stored in the register and the value stored at the aliased location, memory). Unfortunately, the computational complexity of the problem of coloring an interference graph is in the class of NP-problems. Thus, heuristics are required to determine $\min |S(G)|$. We use the following heuristics to determine minimum static data storage required for a given sequencing graph model. In this formulation we do not consider internal pipelining and storage requirements within the functional units. We assume that each operation vertex requires at least one cycle and hence any data transfer across operation vertices in the sequencing graph requires a *holding* register. With each edge in the sequencing graph we associate an integer weight, δ, representing the size of data transfer between corresponding vertices. Weight of an edge that represents a control dependency is set to zero. We assume all such data transfers are synchronous and, therefore, require corresponding storage elements. In case of single execution stream, we use the following algorithm, *single_thread_static_storage*, to identify minimum static storage required for execution of its corresponding linear code.

We produce a single-thread execution schedule of operations using a depth-first search to produce a topological order of vertices in the directed graph [25]. Topological sorting produces a complete order (schedule) of operations that is compatible with the partial order imposed by the sequencing graph. Topological sorting ensures that all edges are directed in only one direction (forward), in order words there are no backward edges [25]. This scheduling operation takes O($|\mathrm{V}| + |\mathrm{E}|$) time. Figure 10 shows an example.

Algorithm to determine $\min |S(G)|$

```
Input: sequencing graph model, G(V,E)
Output: S(G), static storage for a linear code implementation of G
single_thread_static_storage(G)
{
   H = topologically_order(G)
   count = storage = 0 ;
   ∀ u ∈ V(H)
   {
      ∀ v ∈ succ(u)
         count = count + δ(u > v) ;
      ∀ v ∈ pred(u)
         count = count - δ(v > u) ;
      storage = max(count, storage) ;
   }
   return storage
}
```

Linearization using Topological Sorting

```
Input: data-flow graph model, G(V,E)
Output: a list of topologically ordered vertices
topologically_order(G) {
   Q = stack = {} ;
   ∀ u ∈ V(G)
      u = white ;
   source_vertex(G) = gray ;
   push(source_vertex(G)) ;
   while stack ≠ {} {
      ∀ v ∈ Adj(Top(stack)) {
         if v = white {
            v = gray ;
            push(v) }
      }
      Q = Q + pop(stack) }
   return Q }
```

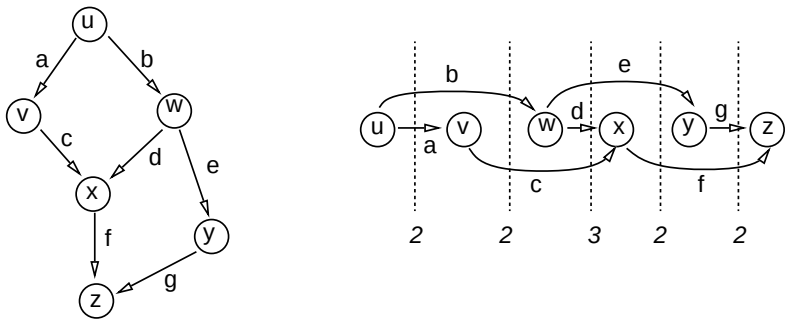


Figure 10: *Determination of minimum static storage for single execution thread*

4.3 Partitioning Feasibility

A system partition into an application-specific and a re-programmable component is considered **feasible** when it implements the original specifications and it satisfies the performance and interface constraints. We assume that the hardware and software compilation, done using standard tools, preserves the functionality. We, therefore, concentrate on constraints. In particular, timing constraints are of special interest. As noted earlier, timing constraints are of two types: those related to min/max time separation between different operations and rate constraints on I/O operations. We assume that the operations subject to a min/max timing constraint belong to the same process model. A timing constraint between operation belonging to two different processes is equivalent to a synchronization constraint between the processes and it can be specified by a blocking inter-process communication in functional specification of the processes. Rate constraints are translated as constraints on latencies of affected sequencing graph models and thus verified by comparing actual hardware and software latencies against imposed bounds. Satisfiability of min/max timing constraints is checked by analyzing the corresponding constraint graph models. For graphs with no unbounded delay operations, constraint satisfiability is related to absence of any positive cycles in the constraint graph model. However, in presence of unbounded delay operations, some constraints may be *ill-posed* [26], that is, constraints that can not satisfied for some *possible* values of delays of unbounded delay operations. Note that a minimum timing constraint is never ill-posed. For an ill-posed maximum timing constraint between vertices v_i and v_j there are two cases:

1. operations v_i and v_j are transitively related, i.e., $v_i >^* v_j$,
2. conversely, operations are concurrent, that is, $\neg(v_i >^* v_j)$ & $\neg(v_j >^* v_i)$.

In the first case, there exists a path, p_{ij}, that contains an unbounded delay operation. The unbounded delay operation may be an external *wait* operation or an internal *loop* operation. For the software implementation of the graph model, a wait operation is made deterministic by performing a *context switch* to a other operations. Thus, satisfaction of the timing constraint by the software component can be verified by assigning appropriate context switch delay (as mentioned in the previous section) to the wait operation. In presence of general unconstrained data-dependent loop operations, determination of satisfiability of

timing constraint is undecidable. However, under special conditions constrained loop operations can be made deterministic by choosing an appropriate policy of loop computation (as shown in the following sections). In the second case, where the constrained operations are concurrent, the constraint can be made well-posed by selective serializations between the two paths of computation.

As mentioned earlier, when partitioning system model into hardware and software components the data rates may not be uniform across models. The discrepancy in data-rates is caused by the fact that the application-specific hardware and re-programmable components may be operated off different clocks and the system execution model supports multi-rate executions that makes it possible to produce data at a rate faster than it can be consumed by the software component when using a using a finite sized buffer. In presence of multi-rate data transfers, feasibility of hardware-software partition is determined by the fact that for all data transfers across a partition, the production and consumption data rates are compatible with a finite and size-constrained interface buffer. That is, for any data transfer across partition, data consumption rate is at least as high as the data production rate. The size of the actual buffer needed may then be determined by using the scheme proposed in [27]. In addition, since the target architecture as shown in Figure 5 contains a single system bus over which data transfer to and from the re-programmable component takes place. Therefore, the net effect of all data-transfers over this bus should not exceed the pre-specified system bus bandwidth. Available bus bandwidth is a function of bus/processor clock rate and memory latency.

4.4 Algorithms for System Partitioning

The partition problem of hardware and software components requires first finding a feasible partition. Among data-rate feasible solutions, a *cost function* of overall hardware size, program and data storage cost, bus bandwidth and synchronization overhead cost is used to determine the quality of a solution. We explore two approaches to obtain a partition of system model into hardware and software components.

4.5 System partitioning based on system non-determinism

We consider approaches to system partitioning in the order of increasing complexity of the system model. Let us first consider a system graph model with no unbounded delay operations and with single-rate execution model. We then look for a partition of a system model driven by satisfaction of the imposed timing constraints. Consider an algorithm that is summarized as follows: starting with an initial solution with all operations in hardware, we select operations for move into the software component based on a cost criterion of communication overheads. Movement of operations to software requires a serialization of operations in accordance with the partial order imposed by the system model. With this serialization and analysis of the corresponding assembly code for a given re-programmable processor, we derive delays through the software component. The movement of operations is then constrained by satisfaction of the imposed timing constraints. Such a partitioning algorithm would strive to achieve maximal number of operations in the software component.

In presence of unbounded delay operations, we can still apply the algorithm described before. Note that unbounded delay operations can not be subject to any maximum timing constraints. Therefore, we

transfer all such operations into the software component and then identify deterministic delay operations for move into the software component such that all timing constraints are satisfied.

However, in systems with *multi-rate* execution model, the data-dependent delay operations makes it difficult to predict actual data-rates of production and consumption across partitions. Further, non-deterministic delays in the system model makes it difficult to statically schedule operations in any implementation of the system design. When considering a mixed implementation of the system design, it is possible to use dynamic scheduling of operations either or in both hardware and software components. Dynamic scheduling of operations in hardware or software requires both area and time overheads that may sometimes render a hardware-software co-design solution difficult or even infeasible. On the other hand, use of static scheduling requires a careful analysis of data-transfer rates across hardware and software portions in order to make sure that possible data-rates can indeed be supported by the interface implementation.

Due to non-determinism in system models, the most general implementation of hardware and software components requires a control generation scheme that supports data-driven *dynamic* scheduling of various operations. Since the software component is implemented on a processor that physically supports only single thread of control, realization of concurrency in software entails both storage area and execution time overheads. On the other hand, in absence of any point of non-determinism from the software, all the operations in the software can be scheduled *statically*. However, such a software model may be too restrictive by requiring the control flow to be entirely in hardware. In our model of software implementation, we take an intermediate approach to scheduling of various operations as described below. First, we make following assumptions about the implementation model:

- The system has an application-specific hardware component that handles all external synchronization operations. (External non-determinism points).
- All the data dependent delay operations (internal non-determinism points) are implemented by software fragments running on re-programmable components.

The software component is thought to consist of a set of concurrently executing routines, called *threads*. A thread consists of a linearly ordered set of operations. The serialization of the operations is imposed by the control flow in the corresponding graph model. Concurrent sets of operations are implemented as separate threads to preserve concurrency specified in the system graph model. All the threads begin with a point of non-determinism and as such these are scheduled dynamically. However, within each thread of execution all operations are statically scheduled. As an example, data-dependent loops in software are implemented as a single thread with a data-dependent repeat count. In this way, we take an intermediate approach between dynamic and static scheduling of software operations. Instead of scheduling every operation dynamically, we create statically known deterministic threads of execution which are scheduled in a *cyclo-static* manner depending on availability of data. Thus, an individual operation in software has a fixed schedule in its thread, however, the time and the number of times the thread may be invoked is data-driven. Therefore, for a given re-programmable processor, the latency,λ, of each thread is known statically. For a given data-input operation in a thread, i, with latency, λ_i, the data consumption rate, ρ_i is bounded as: $\frac{1}{\lambda_{max}} \leq \rho_i \leq \frac{1}{\lambda_i}$ where λ_{max} refers to the latency of the longest thread. It is assumed that the latency includes any synchronization overhead that may be required to implement multiple threads

Partition

Input: System graph model, G = (V, E)
Output: Partitioned system graph model, $V = V_H \cup V_S$
partition(V):

```
V = V^d ∪ V^n                                  /* identify points of non-determinism */
V^n = V^{n_e} ∪ V^{n_i}                        /* external vs internal non-determinism */
V_H = { V^{n_e}, V^d }                         /* the initial hardware component */
V_S = { V^{n_i} }                              /* the initial software component */
create software threads (V^{n_i})              /* create |V^{n_i}| routines */
compute data rates (processor)
if not(feasible(V_H, V_S)) exit                /* No feasible solution exists */
f_min = f(V_H, V_S)                            /* initialize cost function */
repeat
    foreach v^d_i ∈ V^d ∩ V_H                  /* select a deterministic delay operation */
        move(v^d_i)                            /* recursively move operations to sw */
until no improvement in cost function
return(V_H, V_S)
```

move(v^d_i): /* considers a vertex for move from V_H to V_S */

```
if feasible(V_H − {v^d_i}, V_S + {v^d_i})
    if f(V_H − {v^d_i}, V_S + {v^d_i}) < f_min
        V_H = V_H - { v^d_i }                  /* move this operation to sw */
        V_S = V_S + { v^d_i }
        f_min = f(V_H,V_S)
        update software threads
        update data rates (processor)
        foreach v^d_j ∈ succ(v^d_i) ∩ V_H      /* identify successor for move */
            move(v^d_j)
return
```

of execution on a single-thread re-programmable processor. The lower bound on ρ_i is obtained by implementing a software scheduling scheme that reschedules a repeating thread for execution at the end of every iteration.

The system partitioning across hardware and software components is performed by decoupling the external and internal points of non-determinism in the system model. It is assumed that for all external points of non-determinism, the corresponding data-rates are externally specified. Thus, through this decoupling we are able to determine all the data-rates for all the inputs to the re-programmable component. The production data-rates of the re-programmable component are determined by the software synchronization scheme used. We consider the issue of software implementation in Section 6.

From externally specified data rates we compute data rates for data flow edges in the system graph model. The vertex set, V, consists of two sets of vertices, $V = \{V^d, V^n\}$, where V^d denotes the set

of operations whose delay is bounded and known at compile time, and V^n refers to non-deterministic delay vertices. With a data-rate annotated system model as an input, we first isolate its points of non-determinism, V^n, into two groups: V^{n_e}, those caused by external input/output operations, and V^{n_i}, those caused by internal data-dependent operations. The external points of non-determinism, V^{n_e}, are solely assigned to the hardware while the internal points of non-determinism, V^{n_i}, are assigned solely to the software component. With this initial partition we determine the feasibility of data transfers across the partition. If this initial partition is not feasible, then the algorithm fails since no feasible partition exists *under* the proposed hardware-software interface and software implementation scheme. If the initial partition is feasible, then it is refined by migrating operations from hardware to software (i.e., moving vertices from V_H to V_S) in the search for a lower cost feasible partition.

Associated with each internal point of non-determinism (e.g. data-dependent loop bodies) we create a program fragment or a *thread of execution*. Each thread of execution corresponds to a software routine by creating corresponding C code from *HardwareC* description. For various threads of execution in the software component, we derive latency and static storage measures by analyzing the corresponding assembly code. The assembly code is obtained by compiling the corresponding C descriptions. We have considered todate two off-the-shelf components, the R3000 and the 8086, and used existing compilers to evaluate the performance of the corresponding implementation. The algorithm uses a cost function, f = f(size(V_H), size(V_S), Synch_cost(V_H, V_S), $\sum_{interface} data\ rates$) that is a weighted sum of its arguments. The algorithm uses a greedy approach to selection of vertices for move into V_S. There is no backtracking since a vertex moved into V_S stays in that set throughout rest of the algorithm. Therefore, the resulting partition is a local optimum with respect single vertex moves. The overall complexity of the algorithm is quadratic in the number of vertices.

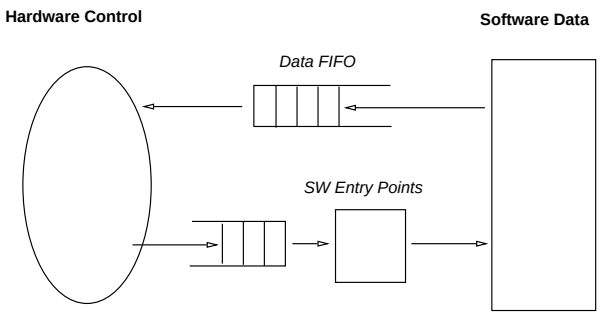


Figure 11: *Partitioning into Hardware Control and Software Execute Processes*

4.6 Partitioning based on decoupling of control and execution

In this section, we briefly mention some of the alternatives ways of partitioning system models in to hardware and software. The partitioning problem can be formulated as the problem of decoupling of control and execution processes. We think of a system model as consisting of interacting control and execution procedures. The execution procedures perform data manipulation where as the control procedures direct the flow of execution and data. There are two possibilities:

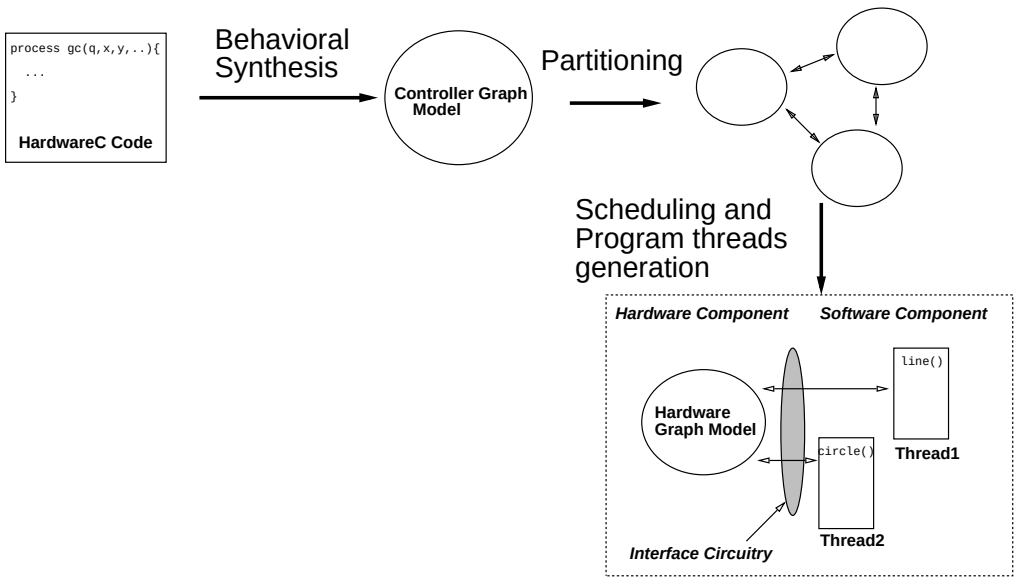


Figure 12: *Partitioned Hardware Model*

1. The hardware generates the address and data values for software execute process to start executing. In Figure 11 the software consists of a set of looping routines. The input data and loop counts are provided to the software addressing unit by the hardware.
2. The software control provides a mechanism to dynamically schedule hardware execute resources. This case is similar to microcoded machines where microcode uses different hardware resources to control flow of execution.

Exploration of alternative partitioning schemes forms a part of the continuing research.

As a result of partitioning of a system model, we have two sets of sequencing graph models representing functionality hardware and software components (Figure 12). These models are translated into dedicated hardware and software as explained in following sections.

5 Implementation of Hardware Components

Hardware implementation consists of synthesis of application-specific components from system model. Timing and resource constraints may be specified in the system model as well as during the design exploration phase of the synthesis process. Application-specific hardware synthesis under resource and timing constraints has been addressed in detail elsewhere [14]. When generating the hardware component attention must be paid to the determination and reachability of a know reset state. Typically this is achieved by use of an extra reset input signal that on assertion steers the ASIC hardware into reset state from any other state. Such a reachability to the reset state from any other state may sometimes be an overkill and expensive since it is obtained at the extra hardware cost of requiring every storage element in the ASIC to be 'reset-able'. It is possible to reduce this overhead by requiring resetability on a selected few storage

elements. The primary idea being that the system can be driven to a known reset state by keeping the primary inputs low and asserting the reset signal for a finite number of clocks needed to traverse the longest circuit path in the asic.

5.1 Hardware Timing and Resource Constraints

Hardware timing constraints were described in Section 3.3. A resource refers to a data-path element, that is a hardware module, which implements one or more *HardwareC* operations. Resource binding of a model refers to mapping of operations in the sequencing graph model to a set of hardware modules. On binding, a hardware module is *instantiated* in order to perform operations to which it is bound to. In general, a greater number of resource constraints results in a larger overall hardware size. However, to few resource instances also increase the hardware size by increasing the size of *control* circuitry needed to share resources among operations. Resource constraints refer to upper bounds on number and types of hardware modules and instances. *HardwareC* supports specification of total number of hardware modules and instances available for synthesis as well as specification of partial binding of operations to specific resources and resource instances. Example 5.1below shows an example of *HardwareC* specification with resource and timing constraints.

Example 5.1. Example of a *HardwareC* process with timing and resource constraints

```
 process example (a, b, c)
   in port a[8] ;
   in channel b[8] ;
   out port c ;
 {
   boolean x[8], y[8], z[8] ;
   tag A, B;
   instance multiply mpyA;

A: x = read(a);
   y = receive(b);
B: if (x > y)
         z = x - y ;
   else
         z = x * y ;
   while (z >= 0) {
         write c = y ;
         z = z - 1 ; }

   constraint maxtime from A to B = 1 cycles;
   constraint resource_usage multiply 1;
 }
```

□

5.2 Constrained Hardware Partitioning

Often the size of application-specific hardware component may be too big to be implemented in a single chip. This is especially the case when using programmable logic devices for the hardware component. Typical field-programmable gate array (FPGA) devices support approximately 1000 equivalent gates where as standard cell ASIC implementations provide up to 50,000 gates. In addition to size of hardware

implementation, structural synthesis of large hardware components itself becomes computationally difficult. For this reason, partitioning of hardware graph models in order to satisfy eventual area, pinout and delay limitations provides an effective means of hardware implementation. Our approach to hardware partitioning is formulated as an *hypergraph partitioning* problem and is described in [23].

6 Implementation of Software Components

In this section we focus on the problem of synthesis of the software component of system design. We consider the software portion to be small and mapped to real memory so that the issues related to virtual memory management are not relevant to this problem. The objective of software implementation is to generate a sequence of processor assembly instructions, or the program, from the set of sequencing graph models, Φ_s, obtained via system partitioning. The generated program is required to satisfy the timing constraints on the sequencing graph model. System partitioning ensures that size constraints on the software component will be observed. This task is accomplished in following four steps (Figure 13).

Step 1: Generation of linearized sets of operations or *program threads* **from** Φ_s. This requires selective serializations to ensure *convexity* of the subgraphs of graph models that are targeted for program threads. A subgraph is considered convex if all paths between any pair of vertices in the subgraph are completely contained in the subgraph. Selective serialization is followed by scheduling of operations in subgraphs of the sequencing graph models. Each maximal set of completely ordered (i.e., linearized) operations represents a potential program thread.

Step 2: Generation of program routines from program threads. In addition to operations in the program threads, a program routine also contains operations that make it possible to achieve concurrency and synchronization between program threads. It may also contain operations that are required to observed imposed timing constraints on Φ_s. Recall the essential problem here is how do we implement various program threads for execution on a processor that supports only sequential execution of operations. Since the processor is completely dedicated to the implementation of the system model and all the program threads are known statically, the final program can be generated in one of following two ways.

1. generate a single program routine that incorporates all the program threads, or
2. provide for multiple-thread executions by means of operation interleaving

In the first case, we attempt to merge different routines and schedule all the operations in a single routine. The unbounded delay operations are explicitly handled either by busy-waiting or by executing specific context-switch operations. In the second case, concurrency between threads is achieved by interleaved execution on a single processor. In principle, operation interleaving can be as "finer-grained" as the primitive operations performed by the processor, that is the assembly instructions. Here we make a further assumption that interleaving is performed at the level of operations used in the sequencing graph model. This assumption is made to avoid otherwise excessive overheads due to implementation of concurrency at processor instruction level. Multiple routines may be implemented using a subroutine relationships to a global routine scheduler. The

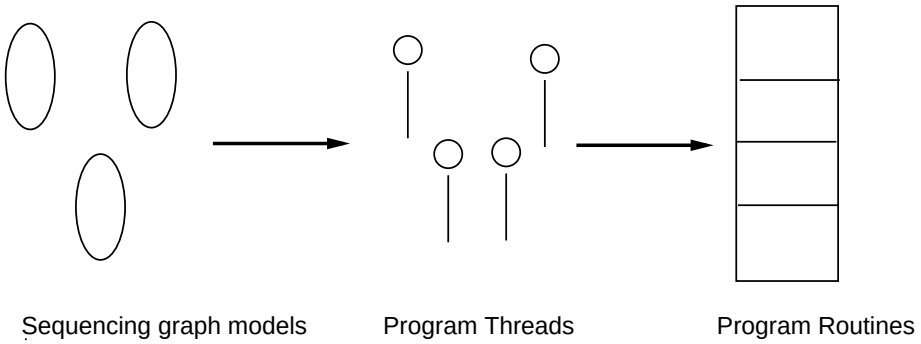


Figure 13: *Steps in generation of the software component*

cost of development of such an scheduler may be high in terms of code space and speed of response. This cost increases substantially in case an attempt is made to avoid possibility of *starvation* of some routines by implementing some kind of *fairness* in scheduling. An alternative would be to implement various routines as a set of *co-operating routines* instead of hierarchical relationship imposed by subroutine call/return. Such an implementation is particularly attractive in our case since it requires relatively little overheads to manage different routines on a single processor. Section 6.4 presents a comparison of different concurrency implementation schemes.

Step 3: Code generation from program routines. For purposes of retargetability, we generate C-code from program routines. Code generation requires translation of operations defined in the sequencing graph model into corresponding operations in C, a high-level programming language, identification of memory locations, binding of variables to memory addresses.

Step 4: Compilation of program routines into processor assembly and object code. C-programs are compiled using existing software compiler for the target processors. Some issues related to interface of the object code to the underlying processor and ASIC hardware must be resolved at this level. These are discussed later in this section.

In this section, we discuss important issues related to generation of the software component. Let us first consider the step of generation of program threads from sequencing graph models. Figure 14 shows the hierarchical graph model for the process `example` described in Example 3.2. The system graph model consists of two graphs, labeled G_0 and G_{loop}. The double-circles indicate operations with unbounded execution delays. Depending on the points of synchronization in a model, the graph can be implemented as a single or multiple program threads. A program thread is so called due to the complete serialization of operations required for the control flow in a single thread of execution. In absence of any points of synchronization, a simple graph model can be translated into a single program thread by ordering all the operations of the graph model. On the other hand, a hierarchical system model is implemented as a set of program threads where each thread corresponds to a graph in the model hierarchy. Thus, the software component consists of a set of program threads. The program threads may be hierarchically related. In addition, some program threads may need to be executed concurrently based on the concurrency among the corresponding graph models. Concurrency between program threads can be achieved by using an

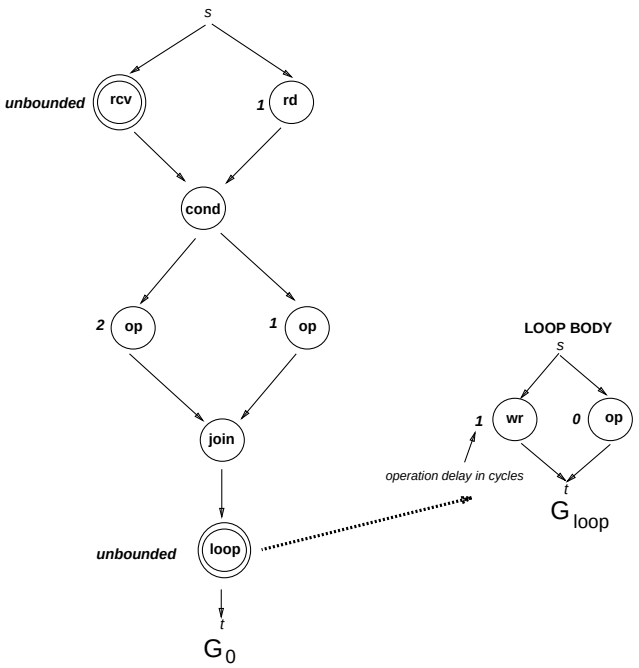


Figure 14: *Example of a graph model containing unknown delay operations*

inter-leaved computation model as explained later in this section.

A program thread may be initiated by a synchronization operation, such as a blocking receive operation (`rcv_synch`). However, within each thread all operations have fixed delay. The (unknown) delay in executing the synchronization operation appears as a delay in scheduling the program thread and it is not considered a part of the thread latency. Therefore, for a given re-programmable device the latency of each thread is known statically. Referring to the example in Figure 14, there are two program threads T_0 and T_{loop}. The thread, T_{loop} consist of serialized operations in the corresponding graph body.

T_0	**T_{loop}**
`rcv_synch`	`loop_synch`
`read`	`write`
`cond_eval`	`op`
`cond_jump`	`detach`
`op add`	
`op mpy`	
`detach`	

Though only a feature of representation, this use of hierarchy to represent control flow is well suited to eventual implementation of the software component as a set of program routines. Since all the operations in a given graph model are always executed, the corresponding routines can be constructed with known and fixed latencies as explained earlier. As with the graph model, the uncertainty due to data-dependent

delay operations is related to invocations of the individual routines. A software implementation consisting of dynamic invocations of fixed latency program threads simplifies the task of software characterization for satisfaction of data rate constraints. Satisfaction of imposed data rate constraints depends on the performance of the software component. Even in presence of unbounded delay operations bounds on software performance can be determined based on its implementation of program threads. In the following sections, we describe a code-level transformation of the data-dependent loop operations that makes it possible to observe imposed input/output rate constraints. In cases, where such transformations are not possible, we use processor interrupts along with bounds on number of interrupts and interrupt latencies to ensure satisfaction of rate constraints.

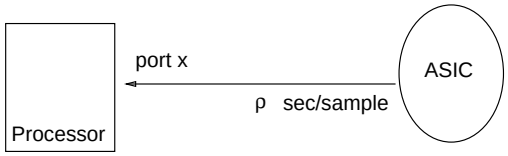
6.1 Rate constraints and software performance

The data rate constraints on the inputs and outputs of the software component are derived based on the corresponding constraints on system inputs and outputs. A data rate constraint on an input (output) specifies a lower bound (in terms of sec/sample) on the rate at which the particular I/O data should be consumed (produced). In case of a deterministic software component, that is, software component with known and bounded execution delays, precise data rates can be computed and checked against corresponding data rate constraints. However, the presence of an unbounded-delay operation between consecutive read (write) operations requires computation of statistical measures (such as distribution of input data value and inter-arrival time) to determine the rate of data production and consumption. A major contribution to the variability of data rates is due to the data-dependent loop operations since the delay due to these operations consists of active execution times rather than 'busy-wait'-type delays encountered by other synchronization operations.

Cyclo-static loop implementation

In some cases, the need for statistical measures can be avoided by transforming the corresponding *dynamic* loop execution model into a *cyclo-static* loop execution model as follows. Consider, for example, a software component that consists of reading a value followed by a data-dependent delay operation shown in Example 6.1.

Example 6.1. Consider a mixed implementation shown by the figure below.



The ASIC component sends to the processor some data on port x at an input rate constraint of ρ sec/sample. The function to be implemented by the processor is modeled by the following *HardwareC* process fragment.

```
process test(x, ...)      |    Thread T1      Thread T2
   in port x [SIZE];      |
{                         |    read           loop_synch
```

```
   ...                    |    detach         <loop_body>
   read x ;               |                   x = x - 1
   while (x >= 0)         |                   detach
   {                      |
      <loop-body>         |
      x = x - 1 ;         |
   }                      |
}
```

x is a boolean array that represents an integer. In its software implementation, this behavior is translated into a set of two program threads shown on the right, where one thread performs the reading operations, and the other thread consists of operations in the body of the loop. For each execution of thread T1 there are *x* execution of thread T2. □

For the *HardwareC* process in Example 6.1, the interval between successive executions of the read operation is determined by the overall execution time of the `while` statement. Due to this variable-delay loop operation, the input data rate at port *x* is variable and is dependent on value of *x* as a function of time. For each invocation of thread T1 there are *x* invocations of thread T2. In other words, thread T1 can be resumed after *x* invocations of thread T2. In absence of any other data-dependency to operations in the loop body, thread T1 can be restarted before completing all invocations of thread T2 by buffering the data transfer from thread T1 to T2. Further, if variable *x* is used only for indexing the loop iterations, the need for inter-thread buffering can be obviated by accumulating value of *x* into a separate loop counter as shown in example below. We call such an implementation of a loop construct in software a *cyclo-static* loop based on the fact that an upper bound on the number of iterations of the loop body is statically determined by the data rate constraints on inputs and outputs that are affected by the data-dependent loop operation.

A cyclo-static loop implementation assumes that there exists a *repeat-count* counter associated with every loop and a loop body is required to be executed as long as its repeat-count is a non-zero number. Additionally, the repeat-count is not used by the corresponding loop body for any purposes other than can keeping a count of number of iterations remaining. Under such conditions, the above component can be transformed into two program threads where one thread reads port *x* and increments the *repeat-count* for the loop body contained in the other thread.

Example 6.2. Transformation of data-dependent loop in Example 2 into a cyclo-static loop

```
process test(x, ...)                  |     Thread T1  Thread T2
   in port x [SIZE]                   |
{                                     |     read       loop_synch
   integer repeat-count = 0 ;         |     add op     <loop-body>
                                      |     detach     repeat-count--
   read x ;                           |                detach
   repeat-count = repeat-count + x ;  |
   while (repeat-count >= 0)          |
   {                                  |
      <loop-body>                     |
      repeat-count = repeat-count-1   |
   }
}
```

(1). For each execution of thread T1 there are $\max(x, m)$ execution of thread T2 where constant m is determined by input data rate constraint, ρ, on the read operation in T1 given by the relation:

$\frac{1}{\rho} = (\lambda_{T1} + m \cdot \lambda_{T2}) \cdot t_{cy}$ where thread latencies λ_{T1} and λ_{T2} include synchronization overheads. t_{cy} denotes cycle time of the processor.

(2). Initialization of variables is performed during system RESET state. □

In this case, we can provide a bound on the rate at which port is read by ensuring that the read thread, Thread T1, is scheduled, say after utmost *m* iterations of the loop body. Due to accumulation of repeat-count additional care must be taken to avoid any potential overflow of this counter. [Generally, overflow can be avoided if *m* is greater than or equal to the average value of *x*. In the extreme, it can be guaranteed not to overflow if *m* is at least maximum of *x* which is equivalent to assigning worst-case delay to the loop operation].

Decoupling data rate from software non-determinism

Due to unbounded delay operations in the software component that is translated into a data-dependent number of invocations of some threads of execution, use of cyclo-static loops may not always be possible or it may lead to implementations that under-utilize the system bus bandwidths, for example, by reserving worst case data-transfer rates for some I/O operations. With concurrent threads, to a certain extent, we can insulate the input/output data rates from variable delays due to other threads by buffering the data transfers between threads. Thus, the **inter-thread buffers** hold the data in order to facilitate multiple executions among program threads. Threads containing specific input/output operations are scheduled at fixed intervals via processor interrupt routines as shown in the Example 6.3 below. In this scheme, finite-sized buffers are allocated for each data-transfer between program threads. In order to ensure the input/output data rates for each thread, we associate a timer with every I/O operation that interrupts the processor once the timer expires. The associated interrupt service routine performs the respective I/O operation and restarts the timer. In case a data is not ready the processor can send the previous output and (optionally) raise an error flag.

Example 6.3.

```
Thread T2          Timer Process           T1 (interrupt service routine)

loop_synch         timer-- per clock tick  read x
<loop_body>        if (timer == 0)         load timer = CONSTANT
x = x - 1                  interrupt       enqueue (x) on dFIFO
detach
```

Thread T1 is now implemented into an interrupt service routine that is invoked at each expiration of the timer process. Timer process represents a processor timer (or an external hardware timer) that is used to generate interrupts at regular intervals. The interruption interval `CONSTANT` is determined by the rate constraint and latencies of interrupt service routines. dFIFO in the interrupt service routine refers to the buffer between threads T1 and T2. □

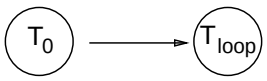
This scheme is particularly helpful in case of widely non-uniform rates of production and consumption. In this case, data transfer from processor to ASICs is handled by the interrupt routines thereby leading to a relatively smaller program size for the cost of increased latencies of the interrupt service routines. Section 7.2 presents implementation costs and performance of this scheme.

Next we consider the problem of software synchronization and scheduling mechanisms to make a hardware-software system design feasible.

6.2 Representation of Inter-thread dependencies

Inter-thread dependencies are represented by a *program flow graph*, $\mathcal{P} = (\mathcal{V}, \mathcal{E})$. The vertices of $\mathcal{P}$ are individual program threads. A directed edge between two vertices indicates a dependency between the two corresponding threads. Example below shows the program flow graph corresponding to the *HardwareC* process described in Example 3.2.

Example 6.4. Program flow graph corresponding to `process example`



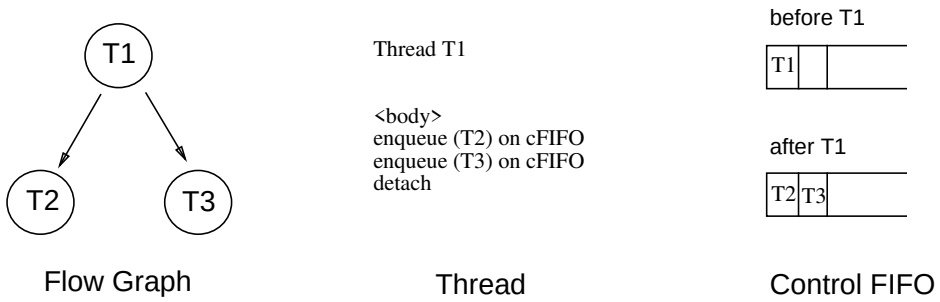
□

6.3 Control Flow in the Software Component

Our model of software component relies on the sequential execution of each thread of execution. Concurrency between threads is achieved through interleaved execution of the threads. Since multiple program threads may be created out of a graph model each starting with an unbounded-delay operation, therefore, software synchronization is needed to ensure correct ordering of operations within the program threads and between different threads.

Since the total number program threads and their dependencies are known statically, the programs threads are constructed to observe these dependencies. The threads are identified by unique tags. A run-time FIFO, called **control FIFO**, maintains the id of the tags that are ready to run based on control flow (while they may still be waiting for data). Before detaching, each thread performs one or more *enqueue* operations to the FIFO for its successor threads as shown in Example 6.5 below.

Example 6.5. Inter-thread control dependencies

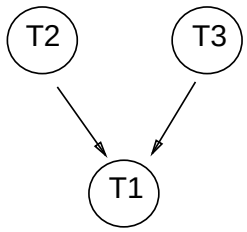


`<body>` refers to the (linearized) set of operations from the corresponding graph models. Control dependency from thread T1 to T2 is built into the code of T1 by the enqueue operation on the control FIFO. □

A thread dependency on more than one predecessor thread (that is a multiple indegree (fanin) node in the flow graph) is observed by ensuring multiple enqueue operations for the thread by means of a counter. For example, a thread node with a indegree of 2 would contain a synchronization preamble code as indicated by the while statement shown in Example 6.6 below.

Example 6.6. Thread with multiple input control dependencies

```
Thread T1

while (count != 1)
{
        count = count + 1;
        detach
}
<body>
count = 0
enqueue <successor threads> on cFIFO
detach
```



□

Control transfer for multiple fanin nodes entails program overheads that add to the latency of the corresponding threads. For this reason, an attempt should be made to reduce multiple dependencies for a program thread through a careful dependency analysis. In case of multiple outdegree nodes in the flow graph, a necessary serialization among enabling of successor threads occurs. However, this serialization is of little significance since there exists only a single re-programmable component.

6.4 Concurrency in Software Through Interleaving

The problem of concurrent multi-thread implementation is well known[28]. In general, multiple program threads may be implemented as subroutines operating under a global task scheduler. However, subroutine calling adds overheads which can be reduced by putting all the program fragments at the same level of execution. Such an alternative is provided by implementing different threads as coroutines [29]. In this case, routines maintain a co-operative rather than a hierarchical relationship by keeping all individual data as local storage. The coroutines maintain a local state and *willingly* relinquish control of the processor at exception conditions which may be caused by unavailability of data or an interrupt. In case of such exceptions the coroutine switch picks up the processes according to a predefined priority list. The code for such a scheduler for coroutines takes approximately 100 bytes in an instruction set that supports both register and memory operands.

Software implementation with explicitly modeled points of non-determinism

Since the processor is completely dedicated to the implementation of the system model and all software tasks are known statically, we can use a simpler and more relevant scheme to implement the software component. In this approach, we merge different routines and describe all operations in a single routine using a method of description by cases [30]. This scheme is simpler than the coroutine scheme presented

Implementation	*Processor type*	*Overhead cycles*
Subroutine	R/M	728
Coroutine	R/M	364
Restricted Coroutine	R/M	103
Description by cases	R/M	85
Restricted Coroutine	L/S	19
Description by cases	L/S	35

Table 3: *Comparison of program thread implementation schemes*

above. Here we construct a single program which has a unique state assignment for each point of non-determinism. A global state register is used to store the state of execution of a thread. Transitions between states are determined by the requirement on interrupt latency for blocking transfers and scheduling of different points of non-determinism based on data received.

This method is restrictive since it precludes use of nested routines and requires description as a single switch statement, which in cases of particularly large software descriptions, may be too cumbersome. Overhead due to state save and restore amounts to 85 clock cycles for every point of non-determinism when implemented on a 8086 processor. Consequently, this scheme entails smaller overheads when compared to the general coroutine scheme described earlier.

Table 3 summarizes program overhead for different implementation schemes. The processors are categorized based on availability of memory operands in the instruction set. A *register-memory* (R/M) processor supports both register and memory operands for its instructions, typical of 'complex-instruction set' processors like Motorola 68K or Intel x86 series. A *load-store* (L/S) processor supports use of memory operands only in two specific 'load' and 'store' instructions, typical of 'reduced-instruction set' processors like Mips R(2/3)K and Sun SPARC series. *Overhead cycles* refers to the overhead (in cycles) incurred due each transfer operation from one program thread to another. A *Subroutine* implementation refers to translation of program threads to program subroutines that operate under a global task scheduler (or the *main* program). A *Coroutine* implementation reduces the overhead by placing routines in a co-operative, rather than hierarchical, relationship to each other. A *Restricted coroutine* implementation reduces the overhead further by suitably partitioning the onboard register storage between program threads such that program counter is the only register that is saved/restored during a thread transfer. In case of R/M processors the case description scheme reduces the overhead by reducing amount of ALU operations in favor of a slight increase in memory input-output operations.

6.5 Issues in Code Generation from Program Routines

As mentioned earlier, we generate C-code from partitioned graph models. Use of high-level programming language for software generation provides the ability to generate corresponding object code for most commonly used processors. While this *retargetability* can be realized for the most part of the software

component, there are certain program implementation issues that must be addressed while compiling and loading the generated C-programs. In this section, we address the major practical implementation issues.

6.5.1 Memory allocation

The C-compiler uses two kinds of memory structures: *stack* for storing local variables in order to facilitate subroutine calls; *heap* for dynamic allocation of memory space to run-time generated data structures. When using target systems with limited available memory (especially in case of microcontrollers where the on-chip memory is severely constrained), unconstrained use of stack and heap space may lead runtime exceptions that may make the software component non-functional. Fortunately, use of both stack and heap can be avoided by performing static memory allocation in the generated program. Static memory allocation makes the generated program **non-recursive** and **non-reentrant**. The non-recursive nature of the software component is not an issue since the input graph models are themselves non-recursive thus ruling out possibility of recursion in generated programs. A non-reentrant program can not be entered by more than one task. This is usually a problem in case of general-purpose computing systems where a program execution must co-exist with other programs and the operating system software. In our application, the only restriction placed by non-reentrant code is that the main program and the interrupt service routines must not share any procedure calls.

6.5.2 Data types

The standard C programming languages supports the following data types: char, short int, int, long int, float and double. Format compatibility for the encoded/interpreted data types (types other than bit-vectors) becomes an issue when interfacing a general-purpose processor to external hardware such as A/D converters. Further, most standard C-compilers support declaration prefixes *const* and *volatile*. A *const*-declared data set can be mapped to on-chip read-only memory (ROM). For variables declared as shared-storage between program threads and as memory-mapped I/O variables, use of *volatile* declaration preserves these from any compiler-driven optimizations.

6.5.3 The C Standard Library

The standard C-library contains procedures that are called by most C-programs. While most of these procedures are coded as C-programs thus making it portable across systems. However, some of these are written as assembly programs. Commonly used assembly routines are *getchar()* and *putchar()* that are used for most I/O operations. These routines must be written for the target processor. Example 6.7 shows these routines for the MC68HC11 processor.

Example 6.7. Assembly input/output routines for MC68HC11 processor

```
#define RDRF 0x20                    /* Receive data register full */
#define TRDE 0x80                    /* Transmit data ready empty */
#define SCSR * (char *) 0x102e       /* SCI status register */
#define SCDR * (char *) 0x102f       /* SCI data register */

int putchar (c)
```

```
    int c;
{
    while ( !(SCSR & TRDE) );         /* Wait until ready to receive */
    SCDR = c;
    return(c);
}
int getchar ()
{
    while ( !(SCSR & RDRF) );         /* Wait for data */
    c = SCDR;
    return(c);
}
```

□

6.5.4 Linking and loading compiled C-programs

When using routines from the standard C-library, only the routines used by the program are loaded into the object image. The object image consists of memory-relocatable modules. A hardware-software interface often contains fixed memory locations for interface semaphores, hardware devices addresses et cetra. When using relocatable object code, fixed addresses can be generated and used by the program by creating special relocatable modules that are loaded at fixed addresses during executions. Use of smaller relocatable modules for fixed-address generation avoids the problem of having to create fixed-address object modules for the entire software component. Example 6.8 shows how such modules can be used to address a fixed location interrupt vector table.

Example 6.8. Using relocatable modules to generate fixed-address locations

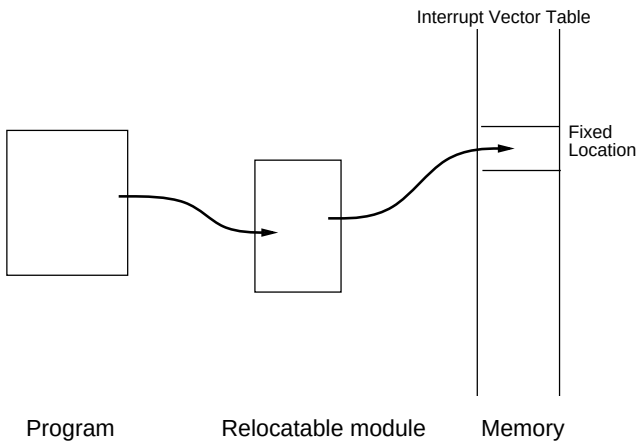


Figure 15: *Generating fixed addresses from C-programs*

The interrupt-vector table is located at a fixed address 0xffd6. The following relocatable module `vector-table` contains pointers to various service routines. `vector-table` is compiled separately and loaded at address 0xffd6.

```
extern void reset();
extern void sci();
```

```
extern void spi();
...
void (* const vector-table[])() = {
   sci(),                          /* SCI service routine */
   spi(),
   ...
   reset,
   };
```

□

6.5.5 Interface to assembly routines

For a variety of reasons, often assembly routines are needed to simplify the task of hardware-software interface tasks. Most common example of assembly programs are programs for runtime startup routines to setup the environment for execution of C-coded programs. A startup routine typically performs the following functions:

1. Load stack pointer (if using stack)
2. Manipulation of hardware registers. Sometimes, a hardware register must be initialized within a certain time interval of power-up that can only be performed by an assembly routines. For example, the block protection register (BPROT) in MC68HC11 must be written within 51 cycles after power-up inorder to enable writes to the on-chip EEPROM.
3. Initialize global variables either by initializing the *automatic initialization block* in static RAM memory generated by the C-compiler for auto-initialized variables, or by using initialized values from a ROM.

When interfacing a C-compiled program to assembly programs, the following issues must be considered:

- global symbols are renamed by the compiler with a prefix that must be used by the assembly routines.
- when passing parameters or returning values from routines, some values may be passed via registers while others may need use of an external stack.
- registers that are used by compiler must be saved and restored when manipulated by the assembly routines.

Use of in-line assembly routines in C-programs simplifies the task of interfacing object code to the underlying processor hardware. A common example of in-line assembly is in enabling/disabling interrupts as shown by the Example below.

Example 6.9. Use of in-line assembly

```
main()
{
        ...
        _asm("di\n");
```

```
        <critical code>

        _asm("ei\n");
}
```

□

As a matter of programming convenience, the in-line assembly instructions need not use explicitly assigned processor registers. Most C-compilers allow use of C-expressions as operands to assembly instructions. This allows us to use critical functions as assembly macros in C source programs as shown by the example below.

Example 6.10. C functions as assembly macros

```
#define sin(x) \
({double _value, _arg = (x) ; \
asm ("fsinx %1, %0" : "=f" (_value) : "f" (_arg)); \
_value; })
```

The assembly instruction `fsinx` uses C expression `x` as an operand. Type declaration `'f'` indicates that a floating point register must be used for this operand. A `'=f'` declaration indicates that output is a floating point register. The output operand `_value` must be a write-only *l-value*. □

7 System Synchronization

Due to pseudo-concurrency in the software component a data transfer from hardware to software must be explicitly synchronized. Using a *polling* strategy, the software component can be designed to perform *premeditated transfers* from the hardware components based on its data requirements. This requires static scheduling of the hardware component. In cases where the software functionality is communication limited, that is, the processor is busy-waiting for an input-output operation most of the time, such a scheme would be sufficient. Further, in absence of any non-determinism, the software component in this scheme can be simplified to a single program thread and a single data channel since all data transfers are serialized. However, this would not support any branching, no reordering of data arrivals since dynamic scheduling of operations in hardware would not be supported.

In order to accommodate different rates of execution of the hardware and software components, and due to unbounded delay operations, we look for a *dynamic* scheduling of different threads of execution. Such a scheduling is done based on availability of data. This scheduling is by means of a **control FIFO** introduced in Section 6.3 which attempts to enforce the policy that data items are consumed in the order in which they are produced. The hardware-software interface consists of data queues on each channel and a FIFO that holds the identifiers for the enabled program threads in the order in which their input data arrives. The control FIFO depth is sized with the number of threads of execution, since a program thread is stalled pending availability of the requested data.

Example 7.1. Hardware-Software Interface

Figure 16 shows schematic connection of the FIFO control signals for **a single data queue**. In this example, the data queue is **memory mapped** at address 0xee000 while the data queue request signal

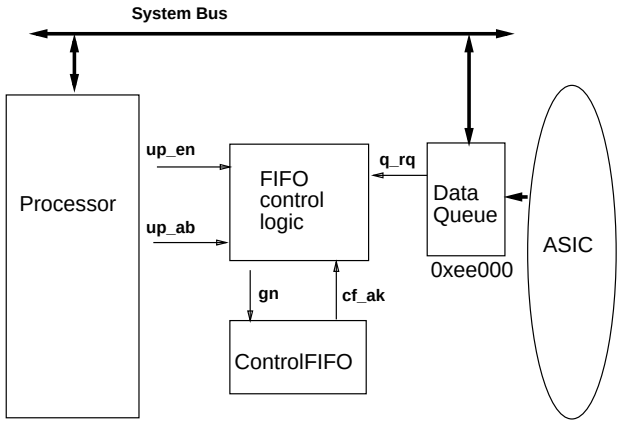


Figure 16: *Control FIFO schematic*

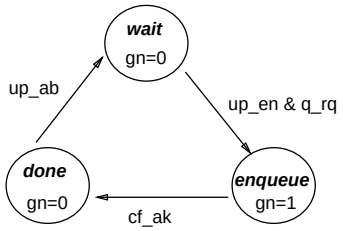


Figure 17: *FIFO control state transition diagram*

is identified by bit 0 of address 0xee004 and enable from the microprocessor (up_en) is generated from bit 0 of address 0xee008.

The control logic needed for generation of the enqueue is described by a simple state transition diagram shown in Figure 17. The control FIFO is ready to enqueue (indicated by `gn = 1`) a process id if the corresponding data request (`q_rq`) is high and the process has enabled the thread for execution (`up_en`). Signal `up_ab` indicates completion of a control FIFO read operation by the processor.

In case of multiple fanin queues, the `enqueue_rq` is generated by OR-ing the requests of all inputs to the queues. In case of multiple-fanout queues, the signal `dequeue_rq` is generated also by OR-ing all dequeue requests from the queue. □

The control FIFO and associated control logic can be implemented either in hardware as a part of the ASIC component or in software. In case the control FIFO is implemented in software the FIFO control logic is no longer needed since the control flow is already in software. In this case, the `q_rq` lines from data queues are connected to processor unvectored interrupt lines, where the respective interrupt service routines are used to enqueue the thread identifier tags into the control FIFO. During the enqueue operations the interrupts are disabled in order to preserve integrity of the software control flow. An specification for the control FIFO based on two threads of execution is given in the Example 7.2 below.

Example 7.2. Specification of the control FIFO based on two threads of execution

```
queue [2] controlFIFO [1];
```

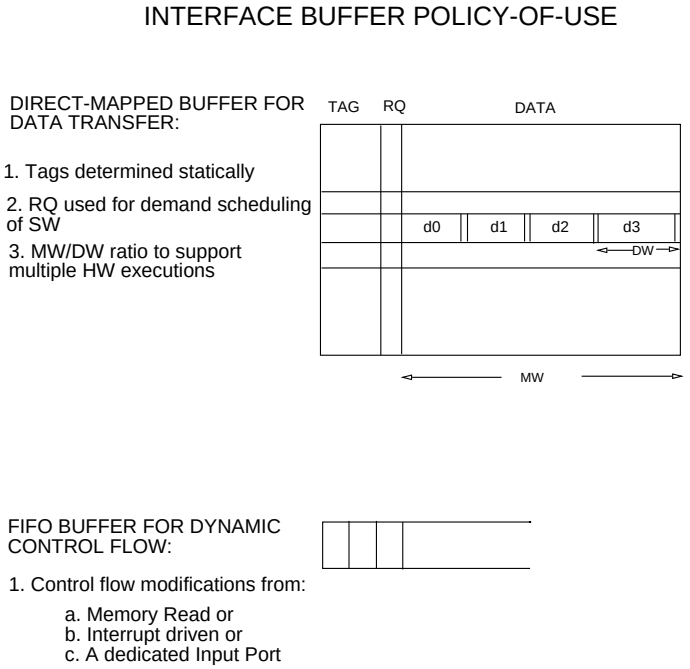


Figure 18: *Hardware and Software Interface Architecture*

```
queue [16] line_queue [1], circle_queue [1];

when ((line_queue.dequeue_rq+ & !line_queue.empty) & !controlFIFO.full) do
controlFIFO enqueue #1;
when ((circle_queue.dequeue_rq+ & !circle_dequeue.empty) & !controlFIFO.full)
do controlFIFO enqueue #2;
when (controlFIFO.dequeue_rq+ & !controlFIFO.empty) do controlFIFO dequeue
dlx.0xff000[1:0];

dlx.0xff000[2:2] = !controlFIFO.empty;
```

In this example, two data queues with 16 bits of width and 1 bit of depth, `line_queue` and `circle_queue`, and one queue with 2 bits of width and 1 bit of depth `controlFIFO` are declared. The guarded commands specify the conditions on which the number 1 or the number 2 are enqueued – here, a '+' after a signal name means a positive edge and a '-' after the signal means a negative edge. The first condition states that when a request for a dequeue on the queue `line_queue` comes and the queue is not empty and the queue `controlFIFO` is not full, then enqueue the value 1 in the `controlFIFO`. The last command just specifies a direct connection between signal `not controlFIFO.empty` and bit 2 of signal `dlx.0xff000`. □

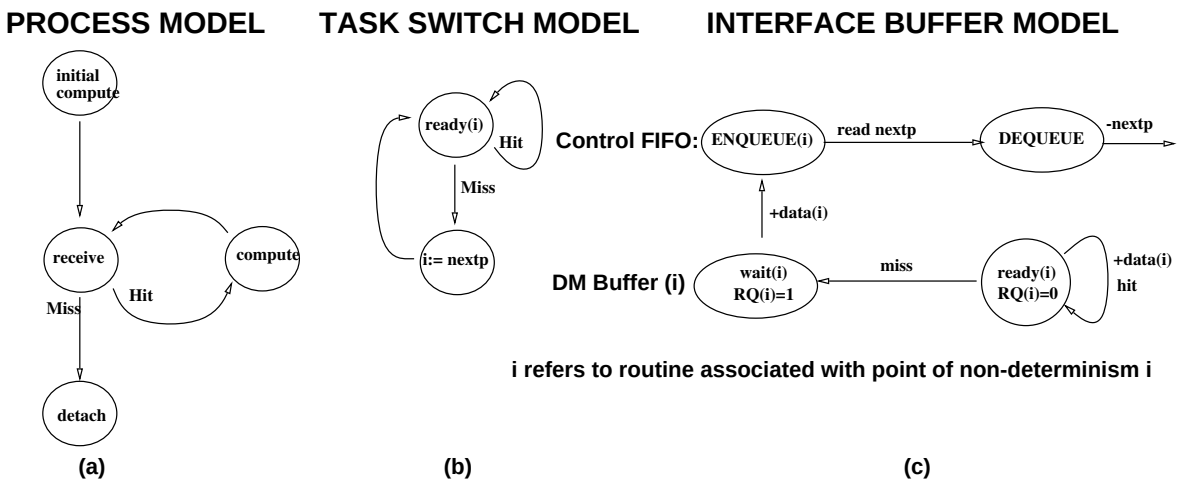


Figure 19: *Hardware and Software Interface Model*

7.1 Hardware-Software Interface Architecture

As mentioned earlier, the hardware-software interface depends on the corresponding data transfer requirements imposed on the system model. In the case of known data-rates where (non-blocking) synchronous data transfers are possible, the interface contains an interface buffer memory for data transfer. Different policy-of-use for the interface buffer is adopted when transferring data or control information across the partition. Therefore, the interface buffer consists of two parts: a data-transfer buffer and a control-transfer buffer (Figure 18). The data-transfer buffer uses an associative memory with *statically determined* tags while the control-transfer buffer uses a FIFO policy-of-use in order to dynamically schedule multiple threads of execution in the software. Associated with each data-transfer we assign a unique tag which consists of two parts, software thread id and the specific data-transfer id. Since all the threads and all input/output operations are known, the tags are determined statically. In addition, the data-buffer contains a request flag (RQ bit) associated with each tag to facilitate demand scheduling of various threads in software. Figure 19 explains the *modus operandi* of data transfer across a hardware-software partition. In the software, a thread of execution is in the compute state as long as it has all the available data [Figure 19(a)]. In case of a miss on a data, the corresponding RQ bit is raised and the thread is detached [Figure 19(c)]. The processor then selects a new thread of execution from the control FIFO [Figure 19(b)]. In case of data arrival to the interface buffer, if the corresponding RQ bit is on, its tag is put into the control FIFO [Figure 19(c)].

Note that the interface architecture described here shows only a *mechanistic* view of the hardware-software synchronization concepts presented before. Its implementation may be made simpler and yet achieve the same effect. For example, the functionality of the associative memory buffer can be translated into a software thread while using a simpler memory structure.

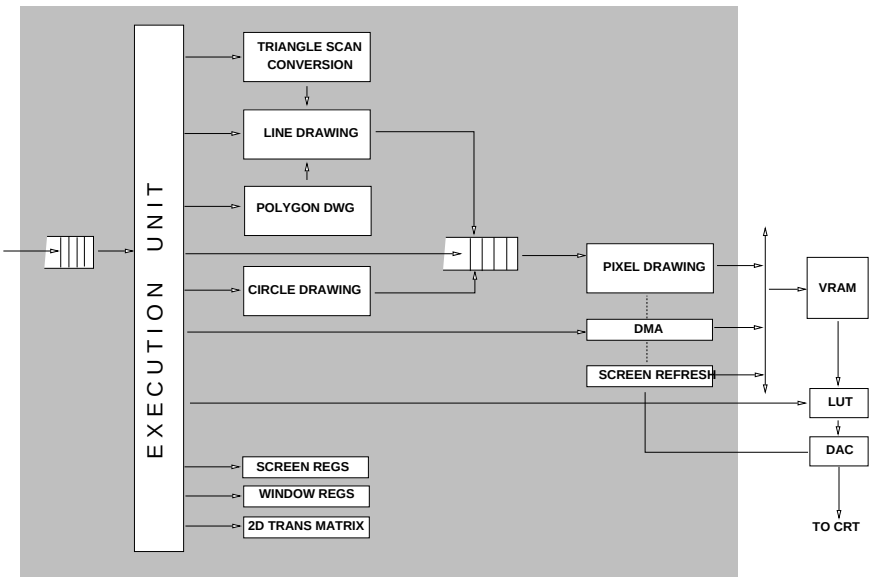


Figure 20: *Graphics Coprocessor Block Diagram*

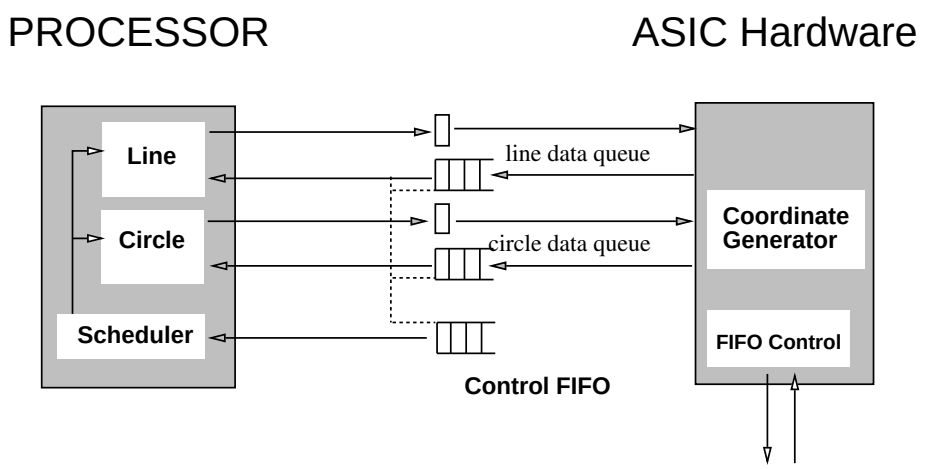


Figure 21: *Graphics Coprocessor Implementation*

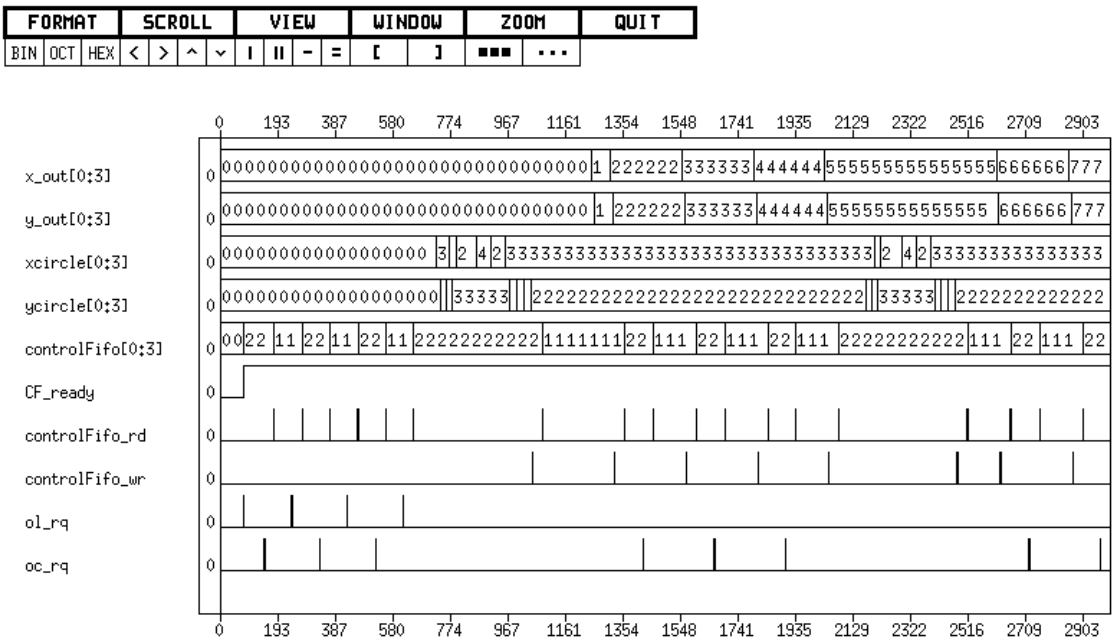


Figure 22: *Graphics Coprocessor Simulation*

7.2 Example

In order to illustrate the effect of software and hardware-software interface implementation, we present design of a graphics controller that outputs pixel coordinates for lines and circles given the end coordinates (and radius in case of circle). The final implementation of the system design consists of line and circle drawing routines in the software component while the ASIC hardware performs initial coordinate generation, coordinate transfer to the video ram. The software component consists of two threads of execution corresponding to the line and circle drawing routines. Both program threads generate coordinates that are used by the dedicated hardware. The data-driven dynamic scheduling of program threads is achieved by a 3-deep control FIFO. The circle and line drawing program threads are identified by id numbers 1 and 2 respectively. The program threads are implemented using the coroutine scheme described in Section 3.1.2.

Figure 23 shows the main program in case of a hardware control FIFO implementation. Like the line and circle drawing routines, this program is compiled using existing C-compiler. Table 4 compares the performance of different program implementations using control FIFO either in hardware or in software component. The hardware implementation of a control FIFO with fanin 3 when synthesized by program *Hebe* and mapped to LSI 10K library of gates using program *Ceres* costs 228 gates. An equivalent software implementation adds 388 bytes to the overall program size of the software component. Note that the cost of hardware control FIFO increases as the number of data queues increases. On the other hand, software implementation of control FIFO using interrupt routines to perform the control FIFO

```
#include "transfer_to.h"

int lastPC[MAXCOROUTINES] = {scheduler, circle, line,main};
int current=MAIN;

int  *controlFIFO_out = (int *) 0xaa0000;
int  *controlFIFO = (int *) 0xab0000;
int *controlFIFO_outak = (int *) 0xac0000;

#include "line.c"
#include "circle.c"

void   main(){
         resume (SCHEDULER);
};

int nextCoroutine;

void scheduler() {
        resume (LINE);
        resume (CIRCLE);
        while (!RESET) {
             do {
                  nextCoroutine = *controlFIFO;
              } while ((nextCoroutine & 0x4) != 0x4);
             resume (nextCoroutine & 0x3);
          }
}
```

Figure 23: *Graphics Controller Software Component*

Scheme	*Program size* (bytes)	*Synchronization overhead delay* (% cycles)	*Input data rate*$^{-1}$ (cycles/coordinate)	*Output data rate*$^{-1}$ (cycles/coordinate)			
				line		circle	
				ave.	peak	ave.	peak
Hardware CFIFO	5972	0	81	535.2	502	76.4	30
Software CFIFO	6588	50	95	749.5	407	106.8	31
Opt. Software CFIFO	6360	29.4	95	651	330	94	31

Table 4: *A comparison of control FIFO implementation schemes*

enqueue operations offers lower implementation cost for a 50% increase in the thread latencies. In case of software implementation of control FIFO, the enqueue and dequeue operations are described in C which are then compiled for DLX assembly. The overhead due to enqueue and dequeue operations is reduced further by manually optimizing the assembly code for enqueue and dequeue operations as indicated by the entry 'Opt. Software CFIFO'. This one time optimization of enqueue and dequeue routines, which does not affect the C-code description of the program threads, leads to a reduction in the program size and program thread overhead to 29.4% thereby improving the rate at which the data is output. Note that data input and output rates have been expressed in terms of number of cycles it takes to input or output a coordinate. Due to a purely data-dependent behavior of program threads, the actual data input and output rates would vary with respect to value of the input data. In this example simulation, the input rate has been expressed for a simultaneous drawing of a line and 5 pixel radius with width of 1 pixel each and the results are accurate to one pixel. An input rate of 81 cycles/coordinate corresponds to approximately 0.25 million samples/sec for a processor running at 20 MHz. Similarly, a peak circle output rate of 30 cycles/coordinate corresponds to a rate of 0.67 million samples/sec.

Though instructive, the line and circle drawing algorithms are simple enough that their software implementation do not fully exploit the potential of a mixed implementation. However, a more computationally intensive operation like spline generation or operations involving floating point arithmetic would greatly benefit by their program implementations.

8 Example of System-level Synthesis: Network Coprocessor

The coprocessor manages the processes of transmitting and receiving data frames over a network under CSMA/CD protocol. CSMA/CD refers to Carrier Sense Multiple Access with Collision Detection protocol used to facilitate communication among many stations over a shared medium (or channel). It is defined by IEEE 802.3 standard. Briefly, CS means that any station wishing to transmit 'listens' first and defers its transmission until the channel is clear. MA implies simultaneous accesses by multiple stations is allowed without the use of any central arbitration. CD refers to collision detection protocol used to detect simultaneous transmission by two or more stations.

The purpose of this coprocessor is to off-load the host CPU from managing communication activities. The coprocessor contains two independent 16 byte wide receive and transmit FIFO buffers. The coprocessor provides a small repertoire of eight instructions that let the host CPU program the machine for specific operations (transmit some data from memory, for example). The coprocessor provides following functions.

- Data Framing and De-Framing
- Network/Link Operation
- Address sensing
- Error Detection
- Data Encoding
- Memory Access

START	enables reception
STOP	disable reception
XMIT	transmit frame
CTADDR	set controller address
SIF	set inter-frame spacing
JAM	set jamming parameter
PREAMBLE	set preamble length in bytes
SFRDELIM	set frame delimeter

Table 5: *Network Coprocessor Instruction Set*

8.1 Host CPU-Coprocessor Interface

Both the CPU and the coprocessor share a bus which can be controlled either by CPU or by the coprocessor. The exclusivity of bus-master is ensured by handshake signals used between the two. The shared bus consists of all Address and Data lines.

In additions to CPU and coprocessor, the bus is also connected to system memory. The coprocessor contains a PC which contains the address from where its next instruction fetch occurs.

8.2 Coprocessor Operation

A typical coprocessor operation can be described as follows:

1. host cpu invokes the coprocessor by write and a memory mapped address
2. the coprocessor responds by making a request for bus control
3. once acknowledged the coprocessor initiates memory read operation to receive command operations
4. once initialized the coprocessor relinquishes control of the bus to host cpu

In the event of a **collision**, the controller manages the 'jam' period, random wait and retry process by re-initializing the DMA pointers without CPU intervention. In case of any **errors** in the received data, the controller re-initializes the DMA pointers and reclaims any data buffers containing the bad frame. All the transmitted and received data is manchester encoded/decoded.

8.3 Coprocessor Architecture

The coprocessor architecture is modeled after the target system architecture shown in Section 2.1. A modification is addition of a local memory and local bus in order to reduce the system bus bandwidth. The coprocessor can be thought of logically consisting of following functional units: execute, transmit and the receive unit. The ethernet controller block diagram is shown in Figure 24.

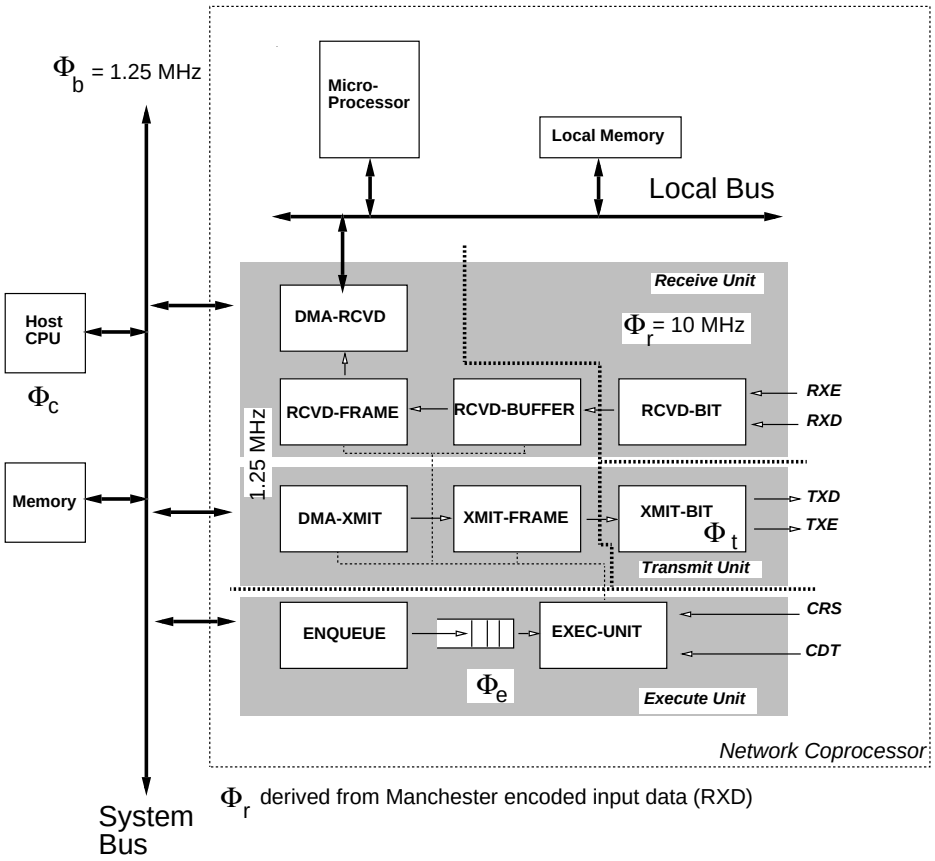


Figure 24: *Network Coprocessor Block Diagram*

The **Execute unit** provides for fetching and decoding of coprocessor instructions. It provides a repertoire of eight instructions listed in Table 5 The **Receive unit** receives frames and stores them into memory. The host cpu sets aside an adequate amount of buffer space and then enables the controller. Once enabled, frames arrived asynchronously. The controller must always be ready to receive the data and store them into a free memory area. The controller checks each received frame for an address match. If a match occurs, it stores the destination and source address and length field in the next available free space. Once an entire frame is received without errors, the controller does the following:

- updates the actual count of the frames received
- fetches address of the next free receive buffer
- interrupts the cpu

Given a pointer to the memory, the **Transmit unit** generates the preamble start frame delimiter, fetches the destination address and length field from the transmit command, inserts its unique address as the source address, fetches data field from buffers pointed by the transmit command, computes and appends CRC at the end of the frame.

The important rate and timing constraints on the coprocessor design are: the maximum input/output bit rate is 10 Mb/sec; maximum propagation delay is 46.4 μs; maximum jam time is 4.8 μs and the minimum inter-frame spacing is 67.2 μs.

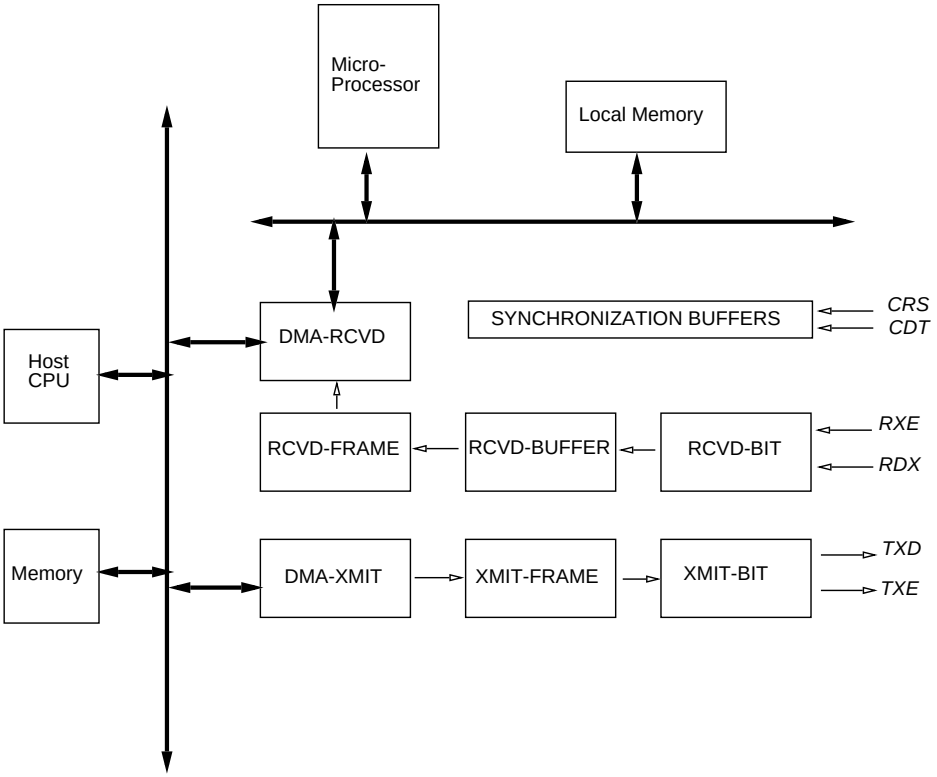


Figure 25: *Network Coprocessor Implementation*

8.4 Network Coprocessor Implementation Results

Due to this partitioning of system behavior into hardware and software components we demonstrate feasibility of achieving a 20 MHz coprocessor using a slower general microprocessor component running at 10 MHz. This speedup in coprocessor performance is achieved by identifying time critical operations and implementing them in dedicated hardware.

The ethernet coprocessor is modularly described as a set of 13 concurrently executing processes which interact with each other by means of 24 send and 40 receive operations. The total *HardwareC* description consists of 1036 lines of code. A mixed implementation following the approach outlined in Section 4.4 was attempted by decoupling the points of non-determinism in the system model. Table 6 shows the results of synthesis of application-specific hardware component of the system implementations that was synthesized in the Olympus Synthesis System and mapped using LSI logic 10K library of gates. Table 7 shows synthesis results using ACTEL library of gates. The software component is implemented in a single program containing case switches corresponding to 17 synchronization points, i.e., internal points

Unit	*Process*	*Area*	*Delay*
Transmit Unit	xmit_bit	271	14.31 ns
	xmit_frame	3183	37.15 ns
	DMA_xmit	2560	45.06 ns
Receive Unit	DMA_rcvd	400	27.51 ns
	rcvd_bit	282	12.30 ns
	rcvd_buffer	127	22.09 ns
	rcvd_frame	1571	38.12 ns
Coprocessor		8394	45.06 ns

Table 6: *Network Coprocessor Synthesis Results using LSI LCA10K Gates*

Unit	*Process*	*Area*	*Delay*
Transmit Unit	xmit_bit	268	128.10 ns
	xmit_frame	2548	246.0 ns
	DMA_xmit	2028	472.85 ns
Receive Unit	DMA_rcvd	563	236.65 ns
	rcvd_bit	211	115.50 ns
	rcvd_buffer	121	199.28 ns
	rcvd_frame	1226	298.40 ns
Coprocessor		7022	472.85 ns

Table 7: *Network Coprocessor Synthesis Results using Actel Gates*

of non-determinism as described in Section 6. With reference to Figure 24, the software component consists of the execution unit and portions of the `DMA_rcvd` and `DMA_xmit` blocks. The reception and transmission of data on the ethernet line is handled by the application-specific hardware running at 20 MHz. The total interface buffer cost is 314 bits of memory elements. Table 8 lists statistics on the code generated by existing software compilers for the ethernet software component implementation.

By contrast, a purely hardware implementation of the Network Coprocessor requires 10882 gates (using LSI 10K library). With a maximum limit of 10000 gates on a single chip, a pure hardware implementation would require *two* application-specific chips. Thus by partitioning into hardware and software components we are able to achieve a 20 MHz coprocessor operation while decreasing the overall hardware cost to only one application-specific chip (or 23% in terms of gate count). The reprogrammability of software components makes it possible to increase the coprocessor functionality, for example addition of self-test and diagnostic features, with little or no increase in dedicated hardware required.

Target Processor	*Pgm & Data Size*	*Max Delay*
R3000, 10 MHz	8572 bytes	56 cycles, 5.6 μs
8086, 10 MHz	1295 bytes	115 cycles, 11.5 μs

Table 8: *Network Coprocessor Software Component*

Figure 26 shows a results of simulation of the final network coprocessor design. This simulation portion shows when the coprocessor is being programmed for transmission. Instructions are supplied to the coprocessor by external programmer (the host CPU) which are enqueue in a 1-deep queue, `ctqueue`. The last pulse on signal `ctqueue_ak` indicates transmission enable instruction after which data is transmitted serially through signal TXD.

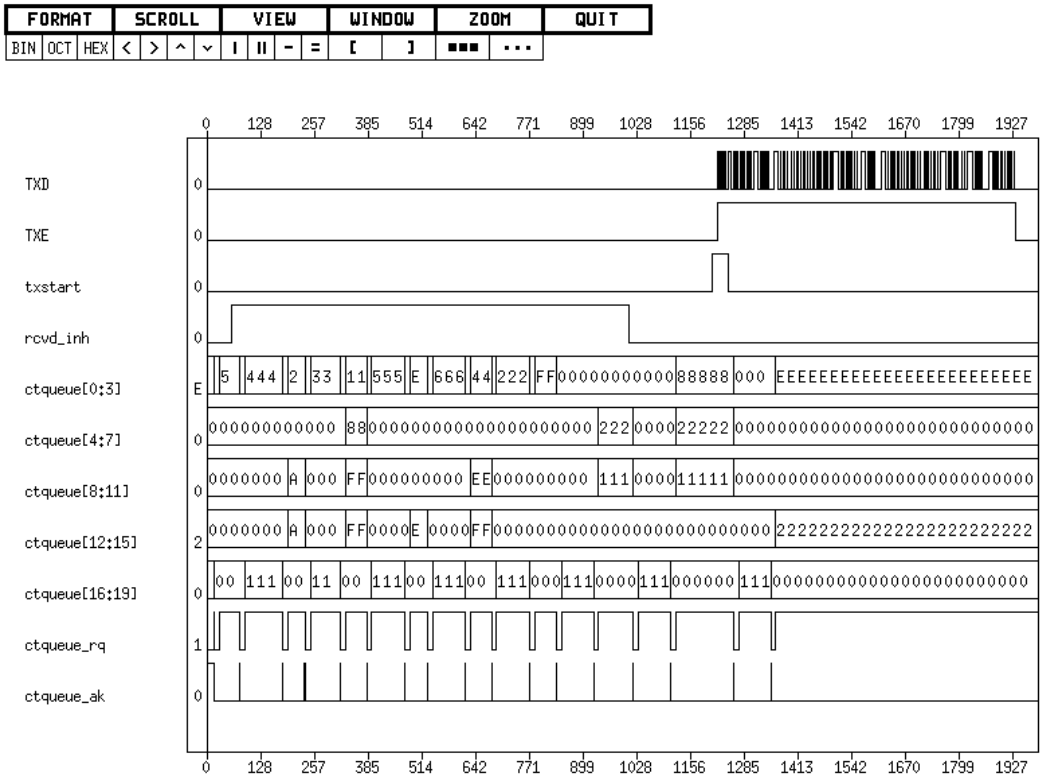


Figure 26: *Network Coprocessor Simulation*

9 Summary

Synthesis of systems containing both general-purpose re-programmable as well as application-specific components can be formulated as a hardware-software co-design problem due to two predominantly different computation models used by the system components. This report attempts to identify important topics and sub-problems in synthesis of hardware-software systems which are then addressed individually. Among the important topics are – system functionality and constraint modeling, system partitioning, hardware-software synchronization and synthesis of hardware and software components. We model system behavior using flow graph that encapsulate system data and control flow. Constraints are specified on these graphs as additional edges or as constraints on graph properties. Constraint-driven partitioning into hardware and software components is performed using a system model that supports non-deterministic delay operations and timing constraints. This partitioning is driven by the satisfaction of timing constraints. A feasible solution to the timing constraints is obtained by identification and separation of internal and external points of non-determinism in the system model.

Hardware implementation of partitioned system models uses synthesis approach formulated in the *Olympus Synthesis System*. Software component design for such systems poses interesting problems due to inherently serial nature of program execution that must interact with concurrently operating hardware components. The software component is generated in two steps. First we create a set of linearized sets of operations, called *threads*. Next, based on concurrency implementation technique, program threads are translated into program routines. We have compared program implementation schemes for achieving low-overhead pseudo-concurrency in the program threads. A coroutine implementation reduces overheads due to hierarchical calling mechanisms by treating all routines symmetrically, therefore, the context information needed to be saved/restored is reduced. However, the necessity to embed control flow into the individual coroutines reduces this gain somewhat, since in some ways the hierarchical context save/restore also contains this control flow information. At the same time, the ability to do intelligent dependency analysis can reduce this overhead in case of coroutines. Case descriptions may result in a smaller overall program implementation in certain cases. The tradeoff between cases and coroutine implementations is dependent on processor ISA.

Synchronization between hardware and software is achieved through the use of a control FIFO buffer. We have demonstrated feasibility of control FIFO-based hardware-software synchronization schemes where the FIFO control can be implemented either as a dedicated hardware or as a program. The software implementation of control FIFO reduces the size of hardware component of system design, but it increases program size and adds to the latencies of program threads. This makes the input data rate about 15% slower in case of the graphics controller example. Depending on the objective of system synthesis either of the hardware and software alternatives can be selected and simulated using program *Poseidon*. Generally, an implementation that aims to rapidly prototype the system design would favor software component of the system design for a small loss of performance. Hardware control FIFO-based schemes require sophisticated hardware in order to implement multi-fanin queues. The interrupt-based schemes reduce the external overhead for the price of additional storage/counters, achieve greater bandwidth utilizations, reduce the effect of thread sizes on supported data rates, thus making an otherwise infeasible partition feasible.

Using the partitioning and implementation approach outlined here we are able to implement the design of an Ethernet based network coprocessor into feasible hardware and software components. The mixed implementation requires 23% less dedicated hardware than a purely application-specific implementation. More importantly, reprogrammability of the software component makes it possible to extend the coprocessor functionality without the need of additional application-specific hardware modules. We are able to simulate mixed system designs using by running concurrent hardware, software simulations with interface protocol described as a set of guarded commands.

The topic of system synthesis using hardware and software is explorative in nature, because of its novelty. Even with the simplifying assumptions relating to the target architecture, the problems of accurately characterizing software component and its synthesis are challenging problems. This work takes a first step in formulating the problem of system synthesis containing hardware and software components. There are several limitations of the approach presented here. First, even though system specifications in *HardwareC* contain explicit concurrency, the algorithmic control flow is not modified during the behavioral synthesis process. This control flow eventually translates into a hierarchical sequencing graph model which influences system partitioning and the generation of program threads. Since no across the hierarchy optimizations are performed the system implementations are affected by the style of specification in *HardwareC*. Current research efforts are attempting to formulate behavioral transformations that alter control flow while preserving overall functional and timing characteristics of a system model. With respect to partitioning, the limitation of the technique presented here would be related to the lack of a feasible partition on some system designs. In addition, the assumptions on hardware, software implementation model and interface scheme influence the partition. As a result, the partitioning results may not be as general to all system designs but specific to the assumptions made for example to the type of re-programmable processor being considered. Further, the hardware and processor cost models used are simplified in order to speed up evaluations of different partitioning alternatives. In particular, our formulation does not make use of processor specific capabilities in the performance estimation of the software component. Currently we do not consider memory hierarchy in our model of system design. Most modern processors come with a certain amount of on-chip cache memory that can be used to speed up the response time of the software component. However, this is not an inherent limitation of our approach, and future extensions include modeling of the effect of cache misses on software performance.

10 Acknowledgments

Authors would like to thank Claudionor Coelho, Jr. and Martin Freeman for helpful discussions. This research was sponsored by NSF-ARPA, under grant No. MIP 8719546 and, by DEC jointly with NSF, under a PYI Award program, and by a fellowship provided by Philips/Signetics. We also acknowledge support from ARPA, under contract No. J-FBI-89-101.

References

[1] G. D. Micheli, D. C. Ku, F. Mailhot, and T. Truong, “The Olympus Synthesis System for Digital

Design," *IEEE Design and Test Magazine*, pp. 37–53, Oct. 1990.

[2] J. Rabaey, H. D. Man, and *et. al.*, *Silicon Compilation, D. Gajski, editor*, ch. Cathedral II: A Synthesis System for Multiprocessor DSP Systems, pp. 311–360. Addison Wesley, 1988.

[3] D. Thomas, E. Lagnese, R. Walker, J. Nestor, J. Rajan, and R. Blackburn, *Algorithmic and Register-Transfer Level: The System Architect's Workbench*. Kluwer Academic Publishers, 1990.

[4] R. Camposano and W. Rosenstiel, "Synthesizing Circuits from Behavioral Descriptions," *IEEE Transactions on CAD/ICAS*, vol. 8, no. 2, pp. 171–180, Feb. 1989.

[5] M. Ligthart, A. Bechtolsheim, G. D. Micheli, and A. E. Gamal, "Design of a Digital Audio Input Output Chip," in *Proceedings of the Custom Integrated Circuits Conference*, pp. 15.1.1–15.1.6, May 1989.

[6] R. K. Gupta, C. C. Jr., and G. D. Micheli, "Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components," in *Proceedings of the* 29^{th}*Design Automation Conference*, June 1992.

[7] C. N. Coelho, D. Filo, and G. D. Micheli, "Channel Sharing," *under preparation*, Stanford University, 1992.

[8] M. C. McFarland, "The Value Trace: A Data Base for Automated Digital Design," Technical Report DRC-01-4-80, Design Research Center, Carnegie-Mellon University, Nov. 1978.

[9] R. Camposano, A. Kunzmann, and W. Rosenstiel, "Automatic Data Path Synthesis from DSL Specifications," in *Proceedings of the International Conference on Computer Design*, pp. 630–635, 1984.

[10] A. Parker, J. Pizarro, and M. Mlinar, "A Program for Data Path Synthesis," in *Proceedings of the* 23^{rd}*Design Automation Conference*, pp. 461–466, June 1986.

[11] R. K. Brayton, R. Camposano, G. D. Micheli, R. Otten, and J. van Eijndhoven, *Silicon Compilers*, ch. The Yorktown Silicon Compiler System, pp. 204–310. Addison Wesley, 1987.

[12] V. Sarkar, *Partitioning and scheduling parallel programs for multiprocessors*. MIT Press, Cambridge, Mass., 1989.

[13] D. Ku and G. D. Micheli, "HardwareC - A Language for Hardware Design (version 2.0)," CSL Technical Report CSL-TR-90-419, Stanford University, Apr. 1990.

[14] D. Ku and G. D. Micheli, *High-level Synthesis of ASICs under Timing and and Synchronization Constraints*. Kluwer Academic Publishers, 1992.

[15] B. Chen and R. Yeh, "Formal Specification and Verification of Distributed Systems," *IEEE Transactions on Software Engineering*, vol. SE-9, no. 6, pp. 710–722, Nov. 1983.

[16] V. Haase, "Real Time Behavior of Programs," *IEEE Transactions on Software Engineering*, vol. SE-7, no. 5, pp. 494–501, Sept. 1991.

[17] P. Caspi and N. Halbwachs, "A Functional Model for Describing and Reasoning Time Behavior of Computer Systems," *Acta Informatica*, vol. 22, no. 6, pp. 595–628, Mar. 1986.

[18] K. Apt, N. Francez, and W. D. Roever, "A Proof System for Communicating Sequential Processes," *ACM Trans. on Programming Languages and Systems*, vol. 27, no. 2, pp. 359–385, July 1980.

[19] S. Owicki and D. Gries, "Verifying Properties of Parallel Programs," *Communications of the ACM*, vol. 19, no. 5, pp. 279–285, May 1976.

[20] A. L. Davis and R. M. Keller, "Data Flow Program Graphs," *IEEE Computer*, vol. 15, no. 2, Feb. 1982.

[21] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," in *Proc. 2nd Annual Symposium on Computer Architecture*, pp. 126–132, 1974.

[22] D. Bustard, J. Elder, and J. Welsh, *Concurrent Program Structures*, p. 3. Prentice Hall, 1988.

[23] R. K. Gupta and G. D. Micheli, "Vulcan - A System for High-Level Partitioning of Synchronous Digital Systems," CSL Technical Report CSL-TR-471, Stanford University, Apr. 1991.

[24] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*, ch. Code Generation, pp. 557–565. Addison Wesley, 1986.

[25] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, ch. VI: Graph Algorithms. The MIT Press, 1990.

[26] D. Ku and G. D. Micheli, "Relative Scheduling Under Timing Constraints: Algorithms for High-Level Synthesis of Digital Circuits," *IEEE Transactions on CAD/ICAS*, vol. 11, no. 6, pp. 696–718, June 1992.

[27] T. Amon and G. Borriello, "Sizing Synchronization Queues: A Case Study in Higher Level Synthesis," in *Proceedings of the* 28^{th}*Design Automation Conference*, June 1991.

[28] G. R. Andrews and F. Schneider, "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys*, vol. 15, no. 1, pp. 3–44, Mar. 1983.

[29] M. E. Conway, "Design of a Separate Transition-Diagram Compiler," *Comm. of the ACM*, vol. 6, pp. 396–408, 1963.

[30] P. J. H. King, "Decision Tables," *The Computer Journal*, vol. 10, no. 2, Aug. 1967.

11 Appendix A: Processor Characterization in Vulcan-II

The syntax of the CPU characteristics file is:

```
        .cpumodel <processor_name> ;

                .machine_type [<stack>, <accumulator>, <gpr>] ;
.operand_type [<rr>, <rm>, <mm>] ;

                .cycle_time <num> ns ;
                .load <num> cycles ;

                .address [<str>]* ;
                .data [<str>]* ;
                .interrupt [<str>]* ;
                .reset <str> ;

                .bus_model ;
                        .type [<muxed> , <de_muxed>] ;

                        .de_muxed ;
                                .mem_read <str> ;
                                .mem_write <str> ;
                                .io_read <str> ;
                                .io_write <str> ;
                        .end_de_muxed ;

                        .muxed ;
                                .read <str> ;
                                .write <str> ;
                                .io <str> ;
                                .mem <str> ;
                        .end_de_muxed ;

                        .bus_hold <str> ;
                        .bus_ack <str> ;

                .end_bus_model ;

                .timing_model ;

                        # timing model
                        # instr delay = read/write access + execution delay + operand EA delays
                        .read_access <num> cycles ;
                        .write_access <num> cycles ;

                        .load <num> cycles ;
                        .store <num> cycles ;

                        .move <num> cycles ;
                        .xchange <num> cycles ;

                        .alu <num> cycles ;
                        .mpy <num> cycles ;
                        .div <num> cycles ;
                        .comp <num> cycles ;
                        .call <num> cycles ;
                        .jump <num> cycles ;
```

```
                .branch <num> cycles ;
                .bc_true <num> cycles ;
                .bc_false <num> cycles ;
                .return <num> cycles ;

                # interrupts are all fixed target locations
                .seti <num> cycles ;
                .cli <num> cycles ;
                .int_response <num> cycles ;

                .halt <num> cycles ;

                # EA calculation delays
                .address_modes ;

                        .immediate <num> cycles ;
                        .register <num> cycles ;
                        .direct <num> cycles ;
                        .reg_indirect <num> cycles ;
                        .mem_indirect <num> cycles ;
                        .indexed <num> cycles ;
                        .other <num> cycles ;

                .end_address_modes ;

                .fpadd <num> cycles ;
                .fpsub <num> cycles ;
                .fpdiv <num> cycles ;
                .fpmul <num> cycles ;

        .end_timing_model ;
.endcpumodel ;
```