# System-level Synthesis using Re-programmable Components

Rajesh K. Gupta Giovanni De Micheli

Center for Integrated Systems
Stanford University, Stanford, CA 94305.

## Abstract

*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. The synthesis of dedicated hardware is then achieved by means of hardware synthesis tools [1], while the software component is generated using software compiling techniques. We consider the problem of identifying potential hardware and software components of a system described in a high-level modeling language and we present a partitioning procedure. We then describe the results of partitioning a network coprocessor.*

## 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 system design is the cost-effectiveness of an *application-specific* hardware implementation versus a corresponding software solution using *re-programmable* components, such as off-the-shelf microprocessors. It is often the case that system design requires a *mixed* implementation that blends application-specific (ASIC) chips with processors, memory and other special purpose modules. Important examples are embedded controllers and telecommunication systems.

Indeed, most digital functions can be implemented by software programs. The major reason 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, a subtask and/or on the input/output data rates. For example, consider design of a data encryption/protocol controller chip. As shown in Figure 1, the DES transmitter takes data from memory using a DMA controller, assembles the frame for transmission, encrypts the data after it receives the key and transmits the encrypted data. The encryption algorithm is a long, iterative process of rotations and bit permutations. Generally it is more cost-effective to implement bit-oriented operations in dedicated hardware whereas it may take too long to execute as a sequence of instructions on most processors. On the other hand, operations related to memory access and frame assembly are typically unconstrained such that they can be relegated to a program running on an already available general-purpose microprocessor, thus substantially reducing the amount of dedicated hardware required for system implementation. Therefore, while a complete application-specific hardware implementation of the controller may be too expensive in terms of hardware size and design time, an implementation which utilizes a re-programmable component may satisfy the performance requirements and at the same time provide the ease and flexibility of reprogramming in software.

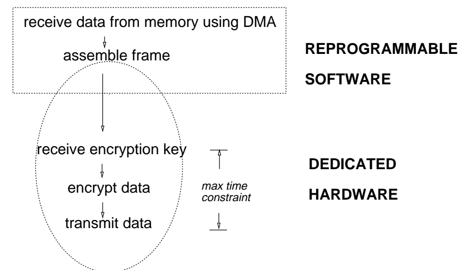


Figure 1: *Example of A Mixed System Implementation*

The problem of hardware-software co-design is fairly complex, and todate there are no available CAD tools to support it. This paper addresses the co-design issue by formulating it as a partitioning problem into application-specific and re-programmable components. We can 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.
2. Determination of the boundary of tasks between ASIC and re-programmable components.
3. Specification and synthesis of the hardware-software interface and related synchronization mechanisms.
4. Implementation of software routines to provide real-time response to concurrently executing hardware modules.

Due to limited space availability in this paper, we touch briefly on the modeling problem and implementation issues in order to describe our approach to obtaining feasible partitions. We consider an approach to hardware-software partitioning based on distribution of unknown delay operations in the system model. This, by no means, is the only way to obtain such a partition, but it allows us to delve into the problem and experiment with existing system designs. For more details on this subject and the related issues mentioned above the reader is referred to [5]. Some potential applications of the techniques presented in this paper are:

**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.

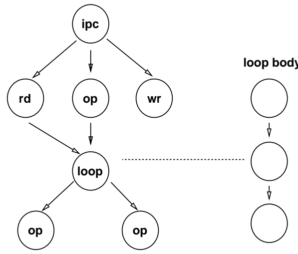


Figure 2: *Example of A System Graph Model*

**Rapid prototyping of complex system designs:** A complete hardware prototype of a complex system is often large. A feasible partition that shifts the non performance-critical tasks to software programs can be used to quickly evaluate the design.

**Speedup of hardware emulation software:** During their development phase, many system designs are often modeled and emulated in software for testing 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 in order to realize maximum speed-up.

# 2 System Model

We model system behavior using a graph representation based on data-flow graphs. The system graph model, $G(V,\ E)$, consists of a set of vertices as operations, $V$, and a set of edges, $E$, which represent either data or sequencing dependency between vertices. Vertices correspond to operations some of which may have data-dependent delays. 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*[6]. We refer to a vertex with data-dependent delay as a *point of non-determinism* in the system graph model. We discuss the property of non-determinism and its effects in Section 3.2. Overall, the system graph model consists of concurrent data-flow sections which are ordered by control flow. 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 and conditionals are specified as separate subgraphs through the use of hierarchy. Figure 2 shows an example of a graph model.

Timing constraints are of two types: (a) minimum/maximum timing separation between pairs of operations and (b) system input-outputs 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 time constraints on multiple executions of the same input or output operation. The system model has no notion of how the actual data transfer takes place in the final system implementation, for example, it may be a synchronous or an asynchronous transfer which may or may not be blocking. The final determination of data-transfer protocol takes place when considering operation partitioning between hardware and software and the associated implementation overheads.

The system graph model can be compiled from most hardware description languages. We use the *HardwareC* [7] language that has a C-like syntax and supports timing and resource constraints.

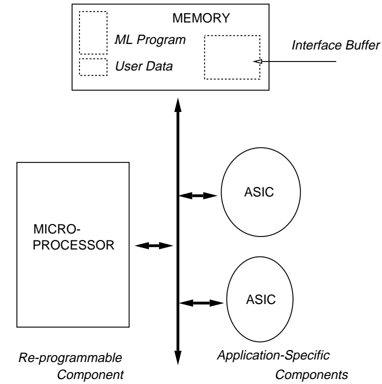


Figure 3: *Target System Architecture*

The input/output rate constraints are specified as additional attributes to the corresponding input/output statements.

# 3 Partitioning of a System Model

Any partition of the system model is also a system graph model and hence it may contain concurrent sets of operations. While such concurrency among operations is natural to any hardware implementation, the corresponding program implementations can execute operations only serially. Thus a hardware-software system may require multiple re-programmable components in order to preserve concurrency inherent (or externally imposed) on the system model. System designs with multiple processors are complicated by the need for a coherent memory management among various processors. In this paper, we make a simplifying assumption that the target system design consists of only one re-programmable component and develop our system synthesis approach based on the target architecture shown in Figure 3. If needed, the hardware component of system design may be partitioned further based on the area, pin-out and latency constraints using the *homogenous* partitioning approaches described in [8]. The resulting hardware modules are connected to the system address and data busses. The memory used for program and data-storage may be on-board the re-programmable component. However, the interface buffer memory needs to be accessible to the hardware modules directly. Because of the complexities associated with modeling hierarchical memory design, we consider the case where all memory accesses are to a single level memory, i.e., outside the re-programmable component.

## 3.1 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. Constraints can be on sizes of hardware and software components, timing constraints between operations and constraints related data rates of system inputs and outputs. Satisfaction of timing constraints is complicated by the presence of non-determinism in the system model. In presence of uniform rates of data transfer, we use in our approach the constraint analysis and synthesis technique [9], that guarantees that either the synthesized circuit satisfies the min/max timing constraints for any value of the delay of the non-deterministic operations, or no solution exists. Since these

techniques have already been described elsewhere [10], we refer the reader to the original papers.

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 [11]. In addition, since the target architecture as shown in Figure 3 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.

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.

## 3.2 Partitioning

We now 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 non-multirate 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.

#### Effect of non-determinism on system partitioning

Non-determinism in our system models is caused either by <u>external</u> synchronization operations or by <u>internal</u> data-dependent delay operations, like conditionals and data-dependent loops. 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.
- For the sake of simplicity and explanation within the scope of this paper, we assume that the system model contains no nested unbounded delay operations, such as a nested loop. However, the software synthesis technique described here can be expanded to include nested unbounded operations[5].

The software component is thought to consist of a set of concurrently executing routines, called <u>*threads*</u>. 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 in order 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 upon 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,$\lambda$, of each thread is known statically. For a given data-input operation in a thread, $i$, with latency, $\lambda_i$, the data consumption rate, $\rho_i$ is bounded as: $\frac{1}{\lambda_{max}} \leq \rho_i \leq \frac{1}{\lambda_i}$ where $\lambda_{max}$ 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 of execution on a single-thread re-programmable processor. The lower bound on $\rho_i$ 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

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

$V = V^d \cup V^n$ /* identify points of non-determinism */
$V^n = V^{n_e} \cup V^{n_i}$ /* 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** $v^d_i \in V^d \cap V_H$ /* 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** $v^d_j \in$ succ($v^d_i$) $\cap V_H$ /* identify successor for move */
      move($v^d_j$)
**return**

Figure 4: *Partitioning Algorithm*

re-programmable component are determined by the software synchronization scheme used. We consider the issue of software implementation in Section 4.2.

The pseudo-code in Figure 4 outlines the partitioning algorithm. A system graph model, $G = (V, E)$, is created by compiling the *HardwareC* description. 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. Alternative approaches to system partitioning when partitioning of non-deterministic operations fails are discussed in [5]. 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, conditional case bodies etc.) 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 correspond-

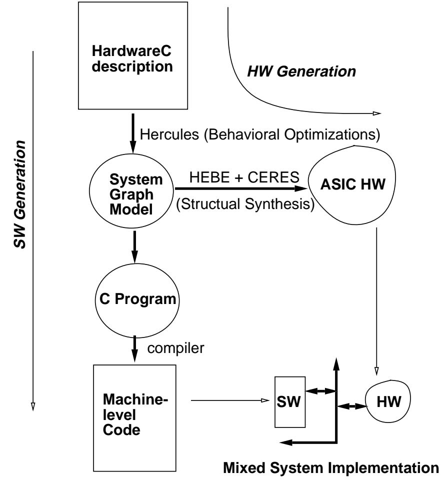


Figure 5: *Generation of Hardware and Software Components*

ing implementation. The algorithm presented in Figure 4 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.

A partition of the system model is indicated by tagging its vertices by either hardware or software. Individual hardware and software components and interface circuitry are created from a partitioned model as described in the following section.

# 4 Implementation of System Components

Figure 5 shows the methodology for generation of individual hardware and software components starting from a functional description in *HardwareC*. The system synthesis is performed by a program called Vulcan-II which invokes appropriate hardware and software synthesis tools to generate the final system design. Hardware synthesis is done using program *Hebe* [1] and software component is generated using available C-compiler for the target processor. Synthesis of interface logic may also be obtained using techniques indicated in [12] [13].

## 4.1 Interface

As mentioned earlier, the hardware-software interface depends upon 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 6). 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

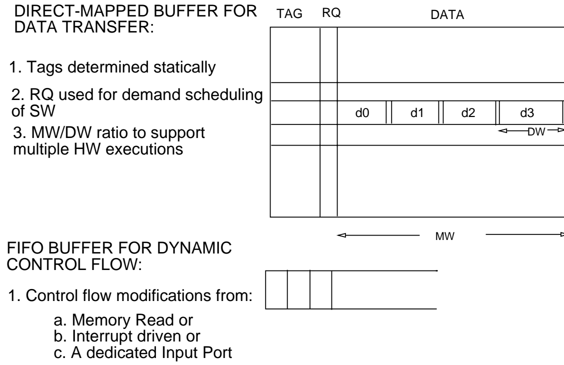


Figure 6: *Hardware and Software Interface Architecture*

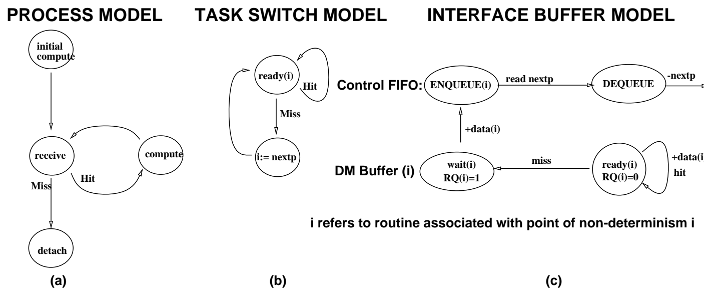


Figure 7: *Hardware and Software Interface Architecture*

data-buffer contains a request flag (RQ bit) associated with each tag to facilitate demand scheduling of various threads in software. Figure 7 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 7(a)]. In case of a miss on a data, the corresponding RQ bit is raised and the thread is detached [Figure 7(c)]. The processor then selects a new thread of execution from the control FIFO [Figure 7(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 7(c)].

### 4.2 Hardware and Software Components

Application-specific hardware synthesis under resource and timing constraints has been addressed in detail elsewhere [10]. Therefore, in this section we focus on the problem of synthesis of the software component of system design. As mentioned earlier, we essentially have a set of program fragments each initiated by a point of non-determinism. The problem of concurrent multi-thread implementation is well known[14]. 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 [15]. 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. The code for a coroutine based scheduler comes to 34 instructions taking about 100

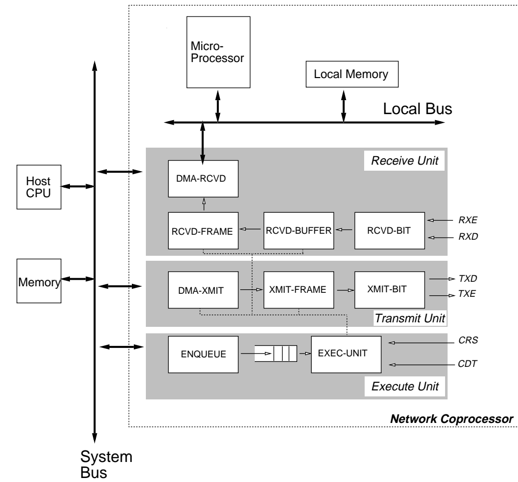


Figure 8: *Network Coprocessor Block Diagram*

bytes in an assembly that provides register-memory operands (like 8086). The coroutine switch takes 364 cycles when implemented for 8086 processor. By contrast, implementation of a global task scheduler using subroutines takes 728 clock cycles for the 8086 processor [5].

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 [16]. This scheme is simpler than the coroutine scheme presented 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.

## 5 Example: Network Coprocessor

As an example of hardware-software system implementation, we describe implementation of a network coprocessor for communication via an ethernet link. The coprocessor manages the processes of transmitting and receiving data frames over a network under CSMA/CD protocol. The purpose of this coprocessor is to off-load the host CPU from managing communication activities. The coprocessor provides following functions: Data Framing and De-Framing, Network/Link Operation, Address sensing, Error Detection, Data Encoding, and Memory Access. In addition, the coprocessor provides a repertoire of eight instructions that let the host CPU program the machine for specific operations (transmit some data from memory, for example). For details on coprocessor operation, the reader is referred to [5]. The network coprocessor block diagram shown in Figure 8 is modeled on the target architecture described in earlier. The important rate and timing constraints on the coprocessor design are: the maximum

| *Unit* | *Process* | *Area* | *Delay* |
|---|---|---|---|
| Transmission Unit | xmit_bit | 271 | 14.31 ns |
| | xmit_frame | 3183 | 37.15 ns |
| | DMA_xmit | 2560 | 45.06 ns |
| Reception 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 1: *Network Coprocessor Application-Specific Hardware Component using LSI LCA10K Gates*

| *Target Processor* | *Pgm & Data Size* | *Max Delay* |
|---|---|---|
| R3000, 10 MHz | 8572 bytes | 56 cycles, 5.6 $\mu$s |
| 8086, 10 MHz | 1295 bytes | 115 cycles, 11.5 $\mu$s |

Table 2: *Network Coprocessor Software Component*

input/output bit rate is 10 Mb/sec; maximum propagation delay is 46.4 $\mu$s; maximum jam time is 4.8 $\mu$s and the minimum inter-frame spacing is 67.2 $\mu$s.

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 3 was attempted by decoupling the points of non-determinism in the system model. Table 1 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. The software component is implemented in a single program containing case switches corresponding to 17 synchronization points, i.e., internal points of non-determinism as described in Section 4.2. With reference to Figure 8 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 2 lists statistics on the code generated by a commercial compiler 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, through the use of system 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.

# 6 Conclusion

We have presented a scheme for performing constraint-driven system-level partitioning into hardware and software components using a system model that supports non-deterministic delay operations and timing constraints. This partitioning algorithm is driven by the satisfaction of data-rate constraints. A feasible solution to the data-rate constraints is obtained by identification and separation of internal and external points of non-determinism in the system model.

Using the partitioning approach presented we were able to partition 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.

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.

The topic of system synthesis using hardware-software partitioning is explorative in nature, because of its novelty. 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.

# 7 Acknowledgements

The authors would like to thank Claudionor Coelho, Jr. 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.*, Cathedral II: A Synthesis System for Multiprocessor DSP Systems, in *Silicon Compilation, D. Gajski, editor*, 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] R. K. Gupta and G. D. Micheli, "System Synthesis via Hardware-Software Co-design," CSL Technical Report CSL-TR, Stanford University, 1992.

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

[7] 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.

[8] R. Gupta and G. D. Micheli, "Partitioning of Functional Models of Synchronous Digital Systems," in *Proceedings of the International Conference on Computer-Aided Design*, (Santa Clara), pp. 216–219, Nov. 1990.

[9] D. Ku and G. D. Micheli, "Relative Scheduling under Timing Constraints," in *Proceedings of the* $27^{th}$ *Design Automation Conference*, (Orlando), June 1990.

[10] D. Ku and G. D. Micheli, "Optimal synthesis of control logic from behavioral specifications," *VLSI Integration Journal*, vol. 3, no. 10, pp. 271–298, Feb. 1990.

[11] 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.

[12] T. H. Meng, *Synchronization Design for Digital Systems*, ch. Synthesis of Self-Timed Circuits, pp. 23–63. Kluwer Academic Publishers, 1991.

[13] G. Borriello and R. Katz, "Synthesis and Optimization of Interface Transducer Logic," in *Proceedings of the IEEE Transactions on CAD/ICAS*, Nov. 1987.

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

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

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