Redesigning Hardware-Software Systems

Claudionor Nunes Coelho Jr. Chih-Yuan Jerry Yang Vincent Mooney
Giovanni De Micheli

Center for Integrated Systems
Dept. of Electrical Engineering
Stanford University
Stanford, California 94305
coelho@pegasus.stanford.edu

Abstract

During the life cycle of a digital reactive real-time system implemented as a hardware-software board or chip, some of its components must be redesigned, either because a refocusing of the product market resulted in a specification change, or because bugs in the specification were found at a later stage of the design.

We address the problem of automatically checking if a new version of a specification can utilize a hardware-software implementation of a previous version of the same specification by just changing the software portion of the design.

The redesigning strategy we propose is divided into four phases. In the first phase, we check which parts of the specification were changed. In the second phase, we extract timing constraints from the previous hardware implementation that must be satisfied by the new software implementation. Then, we schedule and select the instructions in the software routine such that the timing constraints are observed. Finally, we check if the final implementation satisfies the specification rate constraints of the design.

We present an example of a keyboard/mouse device, and we show that the hardware-software synthesis system can be made robust with respect to small changes in the specification.

1 Introduction

Several approaches to co-design of reactive real-time digital systems from behavioral specifications have been proposed in the past [7, 6, 15, 3, 2, 5, 16]. In particular, we consider a design flow as shown in Figure 1. The major weakness of such systems is their inability to handle specification changes.

During the life cycle of a digital system, some of its components often need to be redesigned. **Redesign** addresses the problem of updating an implementation as a result of a change in the specification. This change can result from a refocusing of the product market or from the correction of bugs found at a later stage of the design. Presently, such specification change would require the re-synthesis of the specification, which imposes overhead costs to the implementation, and even delays in the time a product will reach the market. Current hardware-software synthesis tools will not guarantee that the new specification will be mapped in a consistent way into a board or chip manufactured for a previous version of the design.

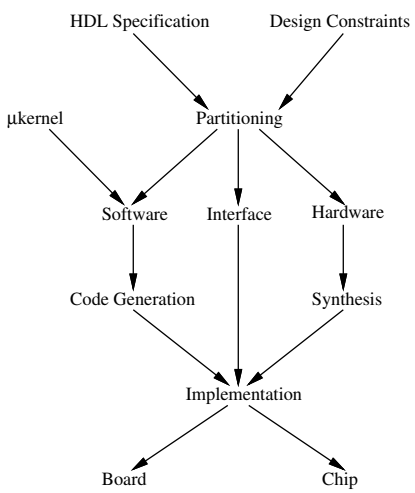

Figure 1: *Generic System for Synthesizing Hardware-Software Reactive Real-Time Systems*

In this paper, we address the problem of automatically checking if a new version of a specification can utilize an implementation of a previous version of the specification by just changing the software portion of the design. Also, we show how to optimally upgrade the implementation to reflect those changes.

This paper is organized as follows. In Section 2 we state the assumptions we make about the specification and synthesis tool. In Section 3, we consider the assumptions on the amount and types of specification changes this formulation can tolerate. In Section 4, we present a solution method for the redesign problem of hardware-software systems, with its four components, which are presented in Sections 5, 6, 7 and 8. Finally, we present the results on the redesigning problem for a keyboard/mouse device on Section 9, and the conclusions of this work on Section 10.

2 Preliminaries

We state in this section the assumptions we make about the specification model, the hardware-software synthesis tool and implementation details that will be considered during the redesign problem of hardware-software systems.

We consider the design of a reactive real-time system that is specified at the behavioral level by some hardware description language, such as VHDL, Verilog HDL or HardwareC. This design is assumed to be further annotated with I/O timing and rate constraints.

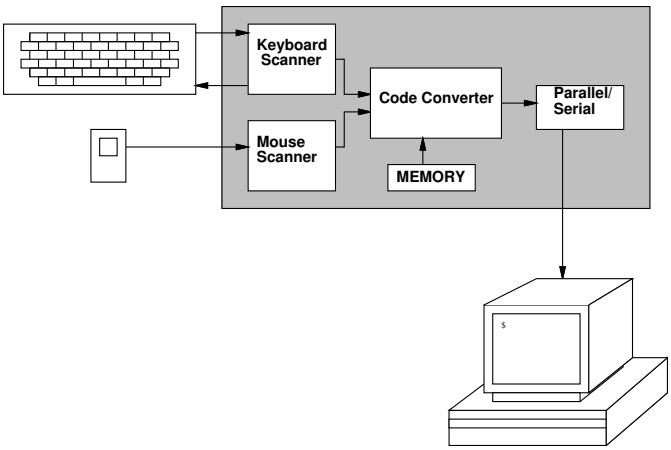

Figure 2: *Keyboard/Mouse Input Device of a Microcomputer*

Example 1 We present in Figure 2 a keyboard/mouse input system, which can be encountered in any microcomputer system. This system is composed by a keyboard input routine, a mouse input routine, a code converter, and a serial transmission line.

In this design, we have design rates under which each part of the specification should function. In particular, the keyboard scanner routine should be able to process any changes in the keyboard every 100 μseconds, the mouse scanner routine should be able to process a movement on the mouse every 100 μseconds, and the serializer should be able to send each bit at a rate of 100 Kbps. □

We assume a co-synthesis tool such as [7, 5] is used to partition the specification into its three components — interface, software and hardware — such that all the interface with the external world is made through registers, which can be implemented either by using ports of a micro-controller or by using registers implemented in hardware.

The interface specifies how the software and hardware partitions communicate. In the original implementation, we assume that data is transferred in either direction by using blocking or non-blocking communication protocols, depending on whether or not the partition also requires the transfer of control.

The parts of the specification to be implemented in software are partitioned into *threads*, which are sets of operations or computations that execute in deterministic time, and can only start execution after another set of computations that execute in non-deterministic time (such as loops and synchronizations with external events) have finished. Since each thread executes in deterministic time once it is started, we can apply conventional code generation for basic blocks [14, 1] in order to generate the code for the operations in the thread. The scheduling of threads over time, however, depends on control and data transfers, and thus a control-data driven scheduler is used in the software that allows both software and hardware to schedule threads over time. For further information on this control-data driven scheduler, we refer the reader to [8, 9]. We also assume that threads will execute in a general purpose microprocessor with a single-level memory hierarchy.

The hardware is synthesized using synthesis tools capable of scheduling operations over time, and allocating and binding resources and registers [13]. We assume here that the degrees of freedom introduced by the software are considered during the hardware synthesis process in order to further optimize the hardware implementation and also to reduce the need for additional synchronizations, as described in general by [10, 11]. For example, hardware-software I/O transfers (also called in this paper **HSI/O**) in a thread that are performed in parallel in the specification can be collapsed and serialized to share the same register port. Strict timing constraints obtained from the HSI/Os can also be used in the hardware to avoid the need for additional synchronizations between the two partitions.

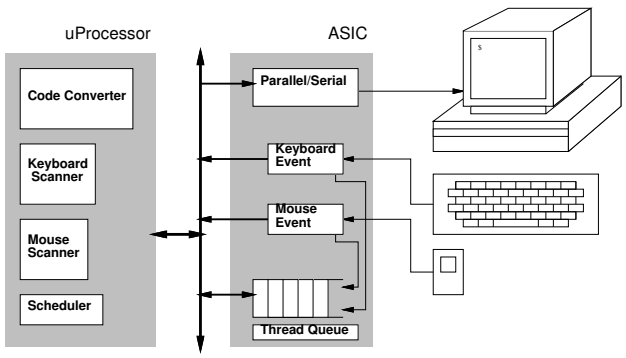

Figure 3: *Hardware-Software Implementation of Keyboard/Mouse Device*

Example 2 For the example of Figure 2, a possible hardware-software partition obtained by any hardware-software synthesis tool is shown in Figure 3.

In this implementation, the latency of the routines CODECONVERT, KEYSCANNER, MOUSESCANNER, SCHEDULER are 305, 155, 84 and 87 cycles on a MIPS R-2000 like microprocessor without pipelining, respectively. Note that this implementation satisfies the rate constraints of the design for a microprocessor running at a speed of 10MHz or more.

The new routines MOUSEEVENT and KEYBOARDEVENT generate events that trigger the software threads MOUSESCANNER and KEYSCANNER, respectively. In addition to that, KEYBOARDEVENT also scans the keyboard sequentially upon an initial command from KEYSCANNER. This circuit was further optimized considering the degrees of freedom introduced by the

schedule of the HSI/O operations in the thread KEYSCANNER. □

3 Assumptions for the Redesign Problem

We want to incorporate the modifications between two versions of a specification in software, thus using the board or chip of the earlier version of the design. We assume that for a version i of the specification, here called S_i, we have already obtained an implementation I_i, for which a board or a chip has been manufactured. Next, a new version $i+1$ of the specification (S_{i+1}) is upgraded from the specification S_i. The new implementation can use the board or chip of the previous version if it observes the HSI/O timing constraints of S_i and the HSI/O rate constraints of S_{i+1}.

We first have to state which types of modifications will be supported by this methodology. We must assume that the changes into the specification were made into the portions mapped into software. Furthermore, we preclude from these changes any modification in the number of I/O signals of the original specification, since this would require changes in the hardware. Note that this does not preclude the external environment to multiplex data into existing I/O signals of the the original design.

We assume that the differences between the two specifications S_i and S_{i+1} can be mostly confined into operations inside basic blocks. Thus, the control-flow structure of S_{i+1} is very similar to the control-flow structure of S_i. In some specific circumstances, our algorithm supports recognizing differences not within the same basic blocks. The conditions and the algorithm for obtaining the differences between S_i and S_{i+1} will be shown later in this paper. Finally, we assume that an annotated version of the specification S_i exists such that we can infer which parts of S_i were implemented in software and which parts of S_i were implemented in hardware.

Example 3 For the example presented in Figure 2, we assume that a new keyboard layout is going to be used. This new keyboard contains the additional key *NUMLOCK* that makes the numeric keyboard generate the same signal codes as the ones generated by the mouse, whenever the key is pressed. □

4 Solution to the Redesign Problem

In order to generate the implementation using the board or chip of a previous version of the design, we must perform the following tasks:

1. Identify the threads that were modified.
2. Extract timing constraints for the modified threads from the previous version
3. Schedule and select instructions for each new thread such that extracted timing constraints are observed.
4. Check that I/O rate constraints of the final implementation are still valid, when these rates are specified.

Each one of these tasks are going to be described in the following sections.

Composition	HL Representation	CF Expression
Sequential	**begin** P; Q **end**	$p \cdot q$
Parallel	**fork** P; Q **join**	$p \| q$
Alternative	**if** (C) P ; **else** Q ;	$c : p + \overline{c} : q$
Loop	**while** (C) P ;	$(c : p)^*$
	wait $(!C)$ P ;	$(c : 0)^* \cdot p$
Infinite	**always** P ;	p^ω

Table 1: *Link between Verilog HDL Constructs and Control-Flow Expressions*

5 Identifying Modified Threads in S_i and S_{i+1}

In this section, we compare the two versions of the specification, S_i and S_{i+1}, and obtain the minimum number of threads that includes the changes between the two specifications. In order to be compared, both specifications are abstracted into their control-flow expressions, which are an abstraction of the control-flow of any hardware description language [10, 11].

5.1 Control-Flow Expressions

Control-flow expressions is an algebra used to model the control-flow of a specification, by abstracting away data-flow information. The data-flow is abstracted in terms of its execution time mapping, which is assumed in this paper to be one clock cycle for each operation; a binding mapping, which binds the operation type to its function; and a synchronization mapping, which specifies how different operations interact.

The data-flow operations can be abstracted by two sets. The first set, called *action* set, associates a label with operations. The second set, called *conditional* set, associates a label with the conditional guards of loops and alternative constructs.

In order to represent the control-flow of a design, control-flow expressions incorporate the usual high-level language constructs, namely the constructs for sequential, alternative, loop, unconditional repetition and parallel composition. Finally, three special symbols, 0, ϵ and δ represent respectively a single cycle action that does not perform any computation and executes in one cycle, a null computation that executes in zero time, and a deadlock, i.e. a computation that cannot finish, respectively. For example, Table 1 shows the representation of the control-flow constructs for Verilog HDL in terms of control-flow expressions. In this figure, we assume that p and q denote the control-flow expressions representing the Verilog HDL code for P and Q, the conditional c abstracts the operation C, and the guards c and $\overline{c}$ encapsulate the con-

ditions for the alternative and loop choices that must be taken during the execution of the Verilog HDL program.

We assume that parentheses have precedence over all control-flow expression operators and they will be used to improve reading clarity in this paper.

Example 4 We can represent the system of Figure 2 by the control-flow expression $\text{DEVICE} = \text{KEYSCANNER} \| \text{MOUSESCANNER} \| \text{CODECONVERTER} \| \text{PARALLELTOSERIAL}$, i.e. the parallel composition of the four parts of the system. The routine PARALLELTOSERIAL can be represented by the control-flow expression $((\overline{\textit{DataReady}} : 0)^* \cdot \text{TXE}_0 \cdot (\text{TXE}_1 \| \text{BIT}_0) \cdot \text{BIT}_1 \cdot \text{BIT}_2 \cdot \text{BIT}_3 \cdot \text{BIT}_4 \cdot \text{BIT}_5 \cdot \text{BIT}_6 \cdot \text{BIT}_7)^\omega$, i.e. a routine that first waits for a data to be ready by sampling the signal $\overline{\textit{DataReady}}$ on every cycle in the control-flow expression $(\overline{\textit{DataReady}} : 0)^*$, and then starts sending the data each bit at a time, and then waits for another data. The action TXE_i corresponds to setting the output signal TXE (transmission enable) to i, and the action BIT_i corresponds to setting the serial data line to the value contained in the i-th bit of some input register. Note that these action names are just conventions used and the only assumption we use in this paper is that two identical operations will have the same action name. □

Control-flow expressions can be pictured as the directed acyclic graph $(V, E, \mathcal{C})$, where V is the set of nodes, E is the set of edges, and $\mathcal{C}$ is a mapping from the edges to the set of guards over the set of conditionals.

The set of nodes V can be further subdivided into $\mathcal{A} \cup \mathcal{O}$, where $\mathcal{A}$ corresponds to the set of actions of the control-flow expression, and $\mathcal{O}$ corresponds to the possible compositions of the specifications, such as sequential $(\cdot)$, parallel $(\|)$, alternative $(+)$, loop $(*)$ and infinite (ω). The sets $\mathcal{A}$ and $\mathcal{O}$ can be also classified as being either deterministic or non-deterministic. Deterministic nodes are assumed to execute in finite time, and non-deterministic nodes have unknown execution time. These latter nodes are also called *anchors* by [12]. Examples of non-deterministic operations are blocking communication nodes, and loops.

Example 5 The directed acyclic graph for the routine PARALLELTOSERIAL shown in Example 4 is shown in Figure 4. □

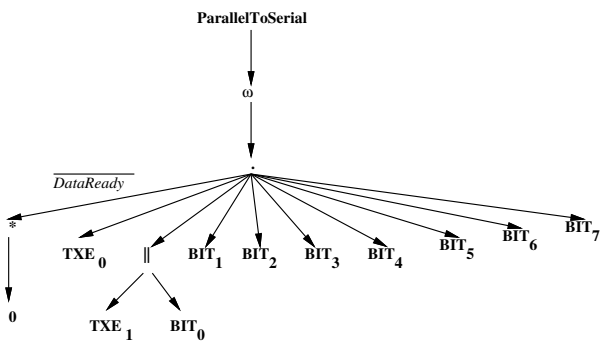

Figure 4: *Directed Acyclic Graph for Control-Flow of Routine* PARALLELTOSERIAL

The advantage of using control-flow expressions for representing the control-flow of the specification is that they concisely capture the behavior of the system in terms of its concurrent control-flow components, their interaction and synchronization. The reader is referred to [11] for a more complete explanation of control-flow expressions. In this paper, we will use control-flow expressions to represent the specifications S_i and S_{i+1} in order to detect the differences between them.

5.2 Matching S_i and S_{i+1}

Verifying whether two specifications are equivalent or not at the behavioral level is a formidable task in general, because the state space involved in the equivalence check is enormous. Thus, instead of considering an exact method for matching S_i and S_{i+1} at an expense of a prohibitive time complexity, we adopt in this paper a *structural checking* of the behavior in terms of its control-flow. This method will identify all threads of the two versions of the specification that do not match, and it will also identify some equivalent threads as mismatches, but at a time complexity that is manageble.

In order to identify which threads of S_i were modified in S_{i+1}, we first have to partition the control-flow of S_i and S_{i+1} into threads. Since the control-flow of the specifications are represented by control-flow expressions, identifying modified threads in the control-flows of S_i and S_{i+1} is equivalent to computing the best match between the subgraphs of S_i and S_{i+1}.

We defined previously that a thread was a set of deterministic operations triggered by one or more non-deterministic operation. Note also that for S_i, we already know which parts of the specification were implemented in software and which parts of the specification were implemented in hardware, by considering each node of the control-flow expression of S_i to be colored by HARDWARE or SOFTWARE, and in the latter case, by a thread number.

The algorithm begins by matching the nodes in S_{i+1} that were colored with HARDWARE with their counterparts in S_i. The algorithm proceeds by marking the non-deterministic operations of S_{i+1} in all other nodes which were not colored by HARDWARE. A non-deterministic operation in a control-flow expression is determined as follows:

- Alternative and parallel composition is considered non-deterministic only if one of its branches are non-deterministic.
- Any suffix of a sequential composition is non-deterministic if the operation preceding it is non-deterministic.
- The bodies of loops and infinite computations are considered to begin in non-deterministic time.
- The exit point of a loop is considered to have a non-deterministic starting time.

Each thread is then formed by considering a set of non-deterministic operations as its starting point, and enclosing as many deterministic control-flow expressions that are composed sequentially as possible.

Example 6 In Figure 5, we represent the threads for the control-flow expression $(o_1 \cdot (c : o_2)^* \cdot o_3)^\omega$, where o_i are actions representing operations, and c is a conditional.

The first thread begins with o_1, i.e. the first operation executed in the infinite composition. The second thread is represented by the loop body. The third thread begins when the loop exits. □

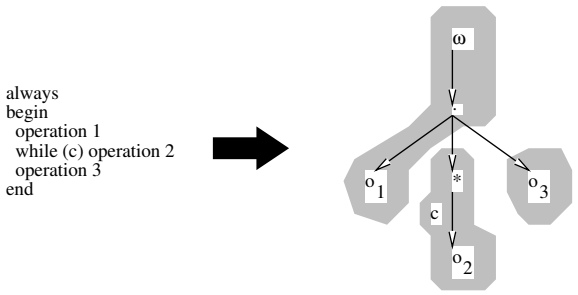

Figure 5: *Thread Extraction from Control-Flow Expression*

In order to match the control-flow expressions, we have to consider two cases. If only deterministic operations can be added or deleted in the original specification, then the difference between threads can be obtained exactly, since there will be a one-to-one correspondence between the activation point of the threads (anchors) of S_i and S_{i+1}. This exact match is possible if the non-deterministic operations appear in the same order in the dags for S_i and S_{i+1}.

If we now consider that non-deterministic operations can be added or deleted in the specification, then we can only find a conservative match between S_i and S_{i+1}. The approximation that we make is that we try to match the HSI/O points and the hardware implementation parts of S_i and S_{i+1}, i.e. the points where timing constraints across the hardware-software partition will have to be analyzed.

The difficulty in the matching S_i and S_{i+1} when non-deterministic operation changes are allowed is the following. Each thread corresponds to a set of operations that execute in deterministic time and thus have a fixed latency. On the other hand, operations that execute in non-deterministic time determine the firing rate in which the threads execute and the synchronization requirements between hardware and software. Since the HSI/O interface was built for an implementation of S_i with all synchronizations made relative to the anchors of S_i, allowing S_{i+1} to have anchor changes may result in synchronizations that were not implemented for S_i, thus requiring a new implementation.

Thus, under the assumptions of small changes to the specification (or changes do not add any synchronization to the interface), we try to match two threads if they have exactly the same HSI/Os and control transfers to hardware, or if two non-deterministic operations of S_i and S_{i+1} enclose HSI/Os that contain timing constraints (of course, if timing constraints are partitioned between two different threads in S_{i+1}, we assume that it is not possible to satisfy the timing constraints, in a similar manner as in relative scheduling [12]). Finally, for the remaining of the non-deterministic operations, we match them in an as soon as possible basis, with respect to the sequential composition.

Example 7 Let us consider now the control-flow expression of Example 6 to be an earlier version of the specification, and the control-flow expression $(o_0 \cdot o_1 \cdot (c : o_2)^* \cdot o_3)^\omega$ to be the upgraded version of the specification. For sake of simplicity, let us assume that the original specification was completely implemented in software.

In this case, an exact match is possible between the two versions of the specification, and only the software for the first thread would be re-implemented from o_1 to $o_0 \cdot o_1$ in this new version. □

6 Obtaining Timing Constraints

Two types of timing constraints should be considered when re-implementing software threads: timing constraints that can be extracted from the interaction of the system with the external environment (usually called I/O timing constraints [4]), and timing constraints that are obtained from the hardware-software interface of the implementation of S_i, which we call HSI/O timing constraints. Note that both timing constraints are made relative to some thread initializations.

Both timing constraints can be considered to be of the form of minimum/maximum timing constraints with respect to a thread initialization. Minimum I/O timing constraints have an upper bound on the thread latency for the previous version. Maximum I/O timing constraints have a lower bound on the thread's initial execution time. Thus, each minimum/maximum timing constraint will appear in pairs, and we will refer to a minimum/maximum timing constraint as an interval timing constraint.

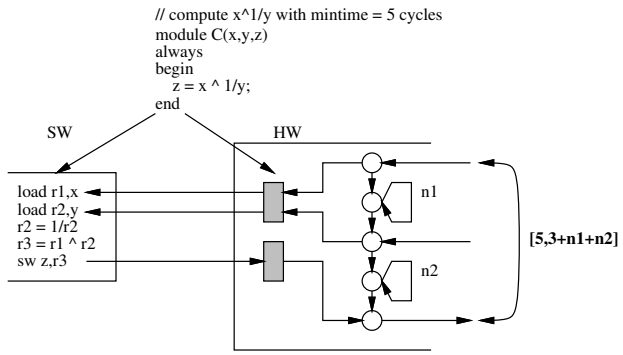

Figure 6: *Interval Timing Constraints*

Example 8 In Figure 6, we present an example on how the minimum timing constraints is transformed into an interval timing constraint after an implementation is obtained. Once a schedule of the hardware and software is obtained, we introduce upper bounds on the time an operation in the new version of the software can be executed, thus creating the interval $[5, 3 + n_1 + n_2]$. □

For the HSI/O timing constraints obtained from the previous implementation I_i, the new version of the specfication S_{i+1} will only interact with the hardware through the register ports, which are shared over time.

In this section, we assume that the interval timing constraints can be obtained for the former type of timing constraints, in a straightforward manner from the specification's I/O timing constraints and the schedules of the implementation I_i. We will thus focus on the problem of extracting the intervals for HSI/Os, which incorporates the external world I/Os under our assumption that all transfers between the design and the external world are made through the hardware.

6.1 Interval Extraction

The interval extraction problem is to find the HSI/O interval $\mathcal{I}(o)$ for all $o \in O$, where O is the set of HSI/Os of a thread.

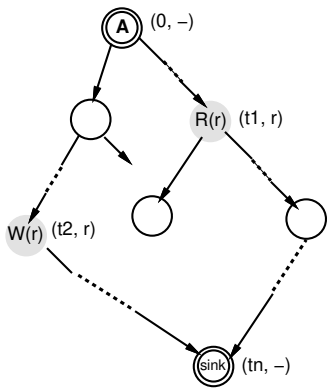

Figure 7: *Example of Interval Extraction*

The procedure to extract the interval sets is relatively straightforward. Before providing the algorithm, we provide the intuition using the graph segment shown in Figure 7. The top vertex A refers to the local anchor of the hardware component [12], and the vertex $sink$ refers to the end of the subgraph execution. Let $R(r)$ and $W(r)$ be HSI/Os of the software that are reads from register port r at time t_1, and writes to register r at time t_2. *Both* are assigned to use register port r.

For hardware input operations, such as operation $R(r)$, the latest time (I_{max}) a value can be written to r by the software is one cycle *before* the execution of $R(r)$, or $t_1 - 1$. The earliest time (I_{min}) is determined by when the register port is available to be written. In this case, since r is not used by any previous operation, $I_{min} = 0$ and the interval set for $R(r)$ is then $\mathcal{I}(R) = [0, t_1]$.

For hardware output operations, such as $W(r)$, the earliest time a software routine can extract data from r is $t_2 + 1$. Since no other operation uses r subsequently, the latest that the software can access r is t_n. The interval for W is $\mathcal{I}(W) = [t_1, t_n]$.

The algorithm is as follows:

```
FindInterval(InOutVertices, Starttime, Binding) {
  foreach v in InOutVertices {
    if (InOutVertices->type == READ) {
    Interval[v]->min = Last previous usage of Binding(v);
      Interval[v]->max = Starttime(v) - 1;
    }
    if (InOutVertices->type == WRITE) {
      Interval[v]->min = Starttime(v) + 1;
      Interval[v]->max = Next usage of Binding(v);
    }
  }
}
```

The parameter $InOutVertices$ is the set of HSI/O nodes implemented in a hardware thread, the schedule of this thread is in $Starttime$, and the resource binding is in $Binding$. Since the algorithm performs a single traversal of all HSI/O nodes, and at each node the computation is unit time, the complexity of the algorithm is linear with respect to the size of the graph.

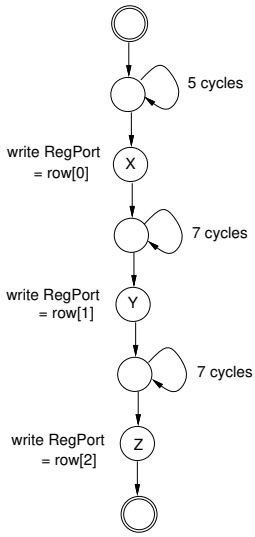

Figure 8: *Interval Extraction for Keyboard Scanner*

Example 9 The hardware schedule for the keyboard scanner is shown in Figure 8. The schedule begins execution upon the activation of the anchor node on top of the graph. The schedule performs a 3 HSI/O operations by writing to the register ports after 5 cycles, 7 cycles, and 7 cycles.

We now extract intervals for X, Y, Z using $FindInterval$. We assume that X, Y, and Z write to the *same* register port. For X, the data to the register port is available at cycle 6 and can be accessed by software until execution of Y at cycle 12. Similarly, interval for Y is $[13, 19]$, and interval for Z is $[21, \lambda]$, where λ is the thread latency. □

7 Scheduling and Selecting Instructions with Timing Constraints

Once the modified threads are obtained and the interval timing constraints for the HSI/O operations are computed, we have to generate the code for these threads. Each thread can be considered to contain a set of data-flow, a set of HSI/O operations and their control/data dependencies. For some of the HSI/O operations, we also have intervals that correspond to the window of time in which the HSI/O operation should occur with respect to the thread's execution.

We initially construct a directed acyclic graph for the thread, where the nodes are data-flow or HSI/O operations. Figure 9 presents a graph in which the shaded node corresponds to an HSI/O operation. For each of the nodes of this graph, we select the instructions by expression tree matching algorithms, such as Burg [14], with a

fixed number of registers. The instruction selection step also provides the execution time for each node of the thread, in terms of cycles.

After obtaining a schedule for each node of the thread, we have to schedule the nodes sequentially such that the HSI/O operations observe the timing intervals obtained from the hardware implementation of the previous specification.

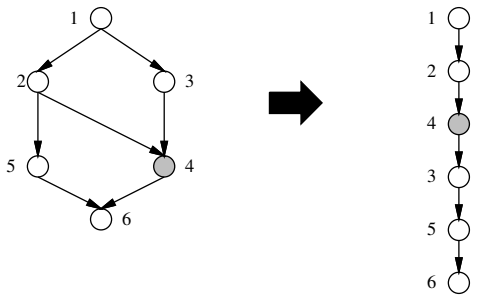

Figure 9: *DAG Representation of a Thread*

The operation execution times and the HSI/O intervals obtained from the previous implementation are then used to serialize the thread operations. We consider that input operations can be scheduled earlier than its use, by using additional storage. We also consider that the scheduling of an output operations can be at most postponed, if such requirement is necessary. Note that we may have to iterate over the code generation for the expression trees in the case we have to interleave HSI/O operations with the code for expression trees, since in this case the number of registers used for the code generation of an expression tree may have to be reduced.

8 Microprocessor Utilization Check

If data rates are specified in the design, after re-implementing the changes that are required in software, we have to check that these data rates are satisfied by the new software implementation. For each thread i of the specification, we have its execution rate ρ_i [7]. Thus, we can check the satisfiability of rate constraints in the new implementation by checking if $\sum \rho_i(\lambda_i + o) \leq 1$, assuming that λ_i is the thread's latency and o is the scheduler overhead.

9 Example

We present in Figure 10 the new implementation for the modified keyboard layout, as presented in Example 3. In this new implementation, the load operations in the software routine of the example must be executed within the time windows shown for the hardware implementation.

A new scheduled routine can be implemented in the target microprocessor with a latency of 188 clock cycles for an unpipelined version of the MIPS-R2000 processor. We can check that this new version of the routine KEYSCANNER still satisfies the execution rates of the specification for a microprocessor running at a speed of 10MHz.

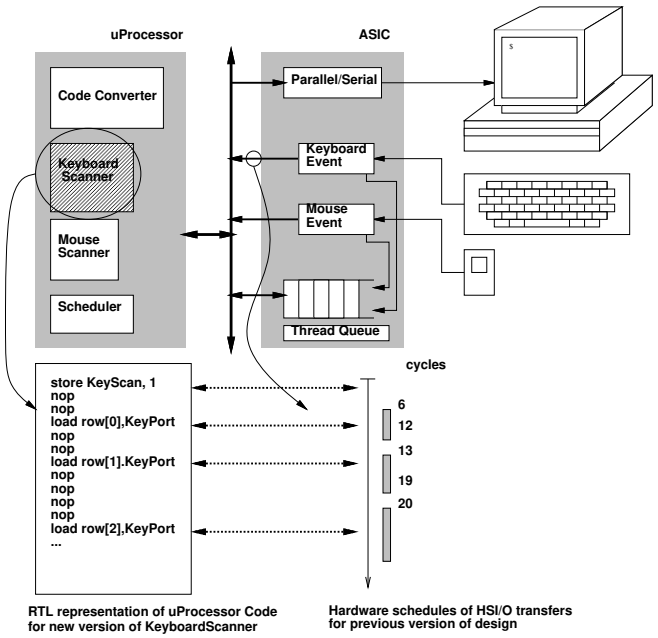

Figure 10: *New Implementation obtained from Previous Version of Design*

10 Conclusions

We presented the problem of implementing a new version of a specification into the board or chip of the previous version. This approach allows the reduction of the costs of re-implementing the specification and thus reduces the time to market of a product.

We assumed that the original specification was partitioned into a software program that runs on a microprocessor or micro-controller, and the hardware is implemented as an ASIC. The program was generated by further partitioning the program into threads, which executed in deterministic time, and which were assumed to be scheduled using a control-data driven scheduler. The hardware was assumed to be synthesized using the degrees of freedom introduced by the software threads. The hardware-software interface was assumed to be composed of blocking and non-blocking communication protocols.

The degrees of freedom allowed by the previous implementation was extracted by examining the hardware schedules of operations involved in the hardware-software interface. The resulting interval sets represented the timing constraints for subsequent software scheduling.

The modification model we considered was built on the assumption that only software changes were required. We used control-flow expressions to determine the changes between the two versions of the specification that would be re-implemented by software.

The problem of synthesizing the new implementation was then divided into four phases. In the first phase, we extracted the changes between the old and the new specification. In the second phase, we extracted the timing constraints from the implementation of a previous version of the specification. In the third phase, the software for the new threads was scheduled observing the hardware-

software transfer behavior of the old implementation. In the last phase, we checked that the new software implementation satisfied the I/O rate constraints of the specification.

We showed an example on the modification of the specification of a keyboard/mouse device that allowed the automatic incorporation of the changes into software.

As future work, we intend to fully automate the redesign process. We also intend to obtain better matching algorithms for the old and the new versions of the specification. Finally, we want to consider some types of hardware changes that can be supported if the hardware is implemented in a look-up table based field-programmable device.

Acknowledgments

The authors would like to acknowledge Dr. K.-C. Chen from Fujitsu Laboratories of America for early discussions on the redesign problem. This research is sponsored by NSF-ARPA under grant MIP 9115432. The first author was supported by the scholarship 200212/90.7 from CNPq/Brazil, and also by a fellowship from Fujitsu Laboratories of America. The second author was supported by a fellowship from Hitachi.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers - Principles, Techniques and Tools*. Addison Wesley, 1988.

[2] E. Barros, W. Rosenstiel, and X. Xiong. Hardware/Software Partitioning with UNITY. In *Notes of Workshop on Hardware/Software Co-design*, October 1993.

[3] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni Vincentelli. Synthesis of mixed software-hardware implementations from CFSM specifications. In *International Workshop on Hardware-Software Co-design*, October 1993.

[4] P. Chou and G. Borriello. Softwware scheduling in the co-synthesis of reactive real-time systems. In *Proceedings of the* 31^{st}*Design Automation Conference*, pages 1–4, June 1994.

[5] P. Chou, E. Walkup, and G. Borriello. Scheduling issues in the co-synthesis of reactive real-time systems. *to appear in IEEE Micro*, 1994.

[6] R. Ernst, J. Henkel, and Th. Benner. Hardware-Software Co-synthesis for Microcontrollers. *IEEE Design & Test of Computers*, December 1993.

[7] R. K. Gupta. *Co-synthesis of Hardware and Software for Digital Embedded Systems*. PhD thesis, Stanford University, 1993.

[8] R. K. Gupta, C. N. Coelho Jr., and G. De Micheli. Synthesis and simulation of digital systems containing interacting hardware and software components. In *Proceedings of the* 29^{th}*Design Automation Conference*, pages 225–230, June 1992.

[9] R. K. Gupta, C. N. Coelho Jr., and G. De Micheli. Program implementation schemes for hardware-software systems. *IEEE Computer*, pages 48–55, January 1994.

[10] C. N. Coelho Jr., D. Ku, and G. De Micheli. An algebra for modeling concurrent digital circuits. In *ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, 1993.

[11] C. N. Coelho Jr. and G. De Micheli. Analysis and synthesis of concurrent digital circuits using control-flow expressions. Technical report, Stanford University, 1994.

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

[13] P.Kission, E. Closse, L. Bergher, and A. Jerraya. Industrial Experimentation in High-Level Synthesis. In *Proceedings of the European Design Automation Conference with EURO-VHDL*, pages 506–511, September 1993.

[14] T. A. Proebsting. *Code Generation Techniques*. PhD thesis, University of Wisconsin - Madison, 1992.

[15] D. E. Thomas, J. K. Adams, and H. Schmit. A model and methodology for hardware-software codesign. *IEEE Design & Test of Computers*, pages 6–15, September 1993.

[16] N. Woo, a. Dunlop, and W. Wolf. Codesign from cospecification. *IEEE Computer*, pages 42–47, January 1994.