Multiple-Process Behavioral Synthesis for Mixed Hardware-Software Systems [*]

Jay K. Adams and Donald E. Thomas

Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Systems composed of microprocessors interacting with ASICs are necessarily multiple-process systems, since the controller in the microprocessor is separate from any controllers on the ASIC. For this reason, the design of such systems offers an opportunity to exploit not only hardware-software trade-offs, but concurrency trade-offs as well. The paper describes an automated iterative-improvement technique for performing concurrency optimization and hardware-software trade-offs simultaneously. Experimental results illustrate that addressing these two issues simultaneously enables us to identify a number of interesting cost/performance points that would not have been found otherwise.

1 Introduction

Systems composed of microprocessors interacting with ASICs are necessarily multiple-process systems, since the controller in the microprocessor is separate from any controllers on the ASIC. For this reason, the design of such systems offers an opportunity to exploit not only hardware-software trade-offs, but concurrency trade-offs as well. Recently, some researchers have begun to address automatic synthesis of multiple-process systems, an area that we call *multiple-process behavioral synthesis*. Some have addressed the issue of synthesizing multiple-process systems for which the behavior of each process is specified explicitly [1] [2], while others have suggested transforming the behaviors of the processes (e.g., moving functionality from one process to another) to explore a larger subset of the design space [3] [4] [5]. The focus of this paper is the latter case. The goal is to explore the benefits of addressing concurrency optimization and hardware-software trade-offs simultaneously.

In general, multiple-process behavioral synthesis presents three optimization problems: *process partitioning*,

global resource allocation, and *global scheduling*. We speak of *global* resource allocation and scheduling to contrast these concerns from the *local* resource allocation and scheduling that occurs in traditional high-level synthesis. For mixed hardware-software systems, there is also the hardware-software partitioning problem. If, however, we assume that the processes in a multiple-process system might have differing implementation technologies, the hardware-software partitioning problem is subsumed by the process partitioning and global resource allocation problems.

Process partitioning is the act of dividing the original behavioral specification into a number of concurrent processes. The result is a behavioral description composed of either more or fewer processes than the original behavioral description. Since the processes may have differing implementation technologies (e.g., software or custom hardware), process partitioning must consider the capabilities of the technology. The goal is to maximize the performance of the system by maximizing the availability of concurrency and choosing an appropriate implementation technology for each function.

Global resource allocation determines how much of the available custom hardware resources should be dedicated to each process. Hardware synthesis tools are capable of producing designs that represent a number of different cost/performance tradeoffs. Global resource allocation selects the cost/performance point that is appropriate for each section of each process. Resource allocation must also take into account the implementation technology of each process, since, clearly, it has an impact on the amount of hardware resources used.

Global scheduling is concerned with the coarse ordering of the computation and the process interactions within each process. Global scheduling has an impact on the amount of concurrency that is achieved. Global scheduling is also clearly affected by global resource allocation and process partitioning.

The problems of multiple-process behavioral synthesis

appear to be interrelated, making it difficult to solve them sequentially. This paper presents a technique for solving the multiple-process behavioral synthesis problem as it relates to synthesis of a class of mixed hardware-software systems. The key to our approach is addressing the interrelated problems simultaneously. One benefit is that the hardware-software trade-offs are considered at the same time as concurrency and global scheduling and allocation issues.

The mixed hardware-software systems we address are characterized by the presence of a number of concurrent processes implemented in custom hardware and a single statically scheduled process implemented in software. We restrict ourselves to a single statically scheduled software process so that it can be implemented with a single off-the-shelf CPU without requiring the presence of a runtime system capable of performing dynamic scheduling.

The goals of this work are similar to those of [6], [7], and [8]. However, previous work in this area has not considered how concurrency trade-offs impact the hardware-software trade-off. Some work has addressed concurrency transformation for behavioral synthesis [4] [5], but none yet has sought to optimize concurrency for mixed hardware-software systems.

Our optimization technique, which is based on iterative improvement, is described in the following section. The experimental results shown in Section 3 illustrate that considering concurrency and hardware-software trade-offs simultaneously enables us to identify a number of interesting cost/performance points that would not have been found otherwise.

2 Approach

Figure 1 illustrates the steps used to optimize the process partitioning (and thus the hardware/software partition) and the global schedule.

The first step is to cluster the operations in the behavioral descriptions into groups of operations called *tasks*. These tasks represent the atomic units of functionality for the rest of the synthesis process. This clustering is done for two reasons: to simplify the remaining synthesis phases by reducing the number of objects in the design, and to improve the accuracy of estimators for hardware resource usage and hardware and software runtime. Further discussion of the importance of task clustering in co-synthesis can be found in [9] and [10]. Currently, we form tasks by simply grouping the operations within a basic block.

Once the clustered system behavioral description is obtained, the next step is to estimate the characteristics of hardware and software implementations of each task. The characteristics estimated are the hardware and software runtimes of the task and the amount of hardware required
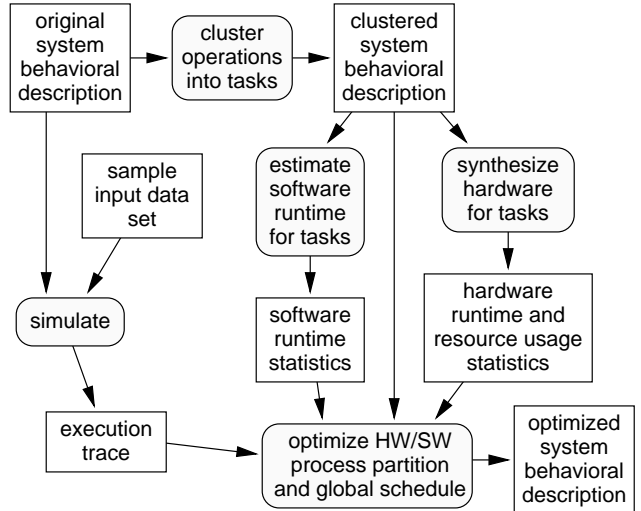

Figure 1: Steps involved in hardware/software process partitioning.

to implement the task. Estimating the software runtimes of the tasks is fairly straightforward since in most cases, the tasks are sequences of operations that can be executed as straight-line code. Gathering the hardware runtime and resource statistics is done with the aid of a high-level hardware synthesis and profiling tools [11].

Once estimates of the task implementations have been generated, this information, along with the clustered behavioral description, is passed on to a hardware/software process partitioning and global scheduling optimization tool. This phase of the synthesis evaluates the hardware/software trade-offs, performs the optimization, and generates behavioral descriptions of the hardware and software processes. Although it is not shown in Figure 1, after the optimization step the resulting hardware processes can then be synthesized by a hardware behavioral synthesis tool, and the software process compiled by a conventional software compiler.

The remainder of this section describes the technique used for optimizing the partitioning of tasks into hardware and software processes and the global scheduling of task invocations and process interactions.

2.1 Design state representation

The design state representation must capture the mapping of task invocations to processes (the process partition) and the control flow within each process among task invocations, control structures, and communication operations (the global schedule). In addition, it is also important for the representation to capture data flow among task invocations and communication operations so that the control flow can be modified without altering the semantics. Data
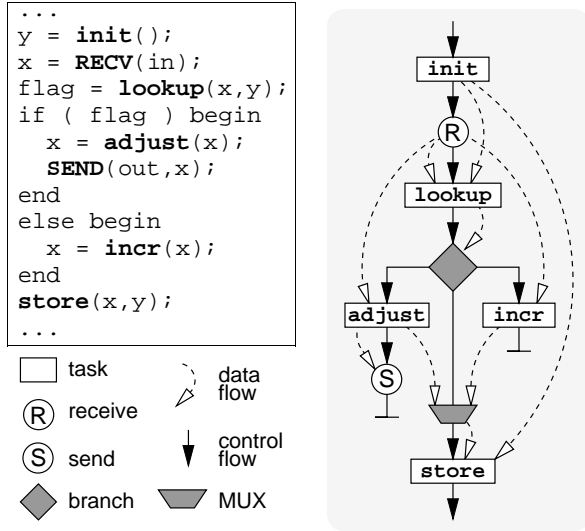
```
...
y = init();
x = RECV(in);
flag = lookup(x,y);
if ( flag ) begin
  x = adjust(x);
  SEND(out,x);
end
else begin
  x = incr(x);
end
store(x,y);
...
```

	task		data flow
R	receive		
S	send		control flow
	branch		MUX

Figure 2: A behavioral description and the corresponding design representation.

flow information is also useful for determining the needed process communication operations when a task invocation is moved from one process to another. To capture the needed data flow information, we use a static single-assignment representation of the system behavior.

Figure 2 shows a fragment of a Verilog behavioral description and a diagram of the our design representation. The behavioral model consists of processes, task invocations, conditional branches, loops, and send and receive operations.

2.2 Design space exploration

To explore alternative amounts of concurrency and hardware-software partitions, transformations are needed that move functionality (task invocations and control structures) from one process to another. The basic transformation we use to accomplish this is *inter-process code motion*. This transformation moves a task invocation or control structure from one process to another, creating any process communication operations necessary for the task to be performed in the new process.

Figure 3 shows an example of what happens to the design representation when a task invocation (in this case, the invocation of task 'b') is moved to another process. The figure shows control sequences in two processes that are executed under the same conditions. The invocation of b is first removed from the control flow of the originating process and communication operations are created for transferring the needed values (Figure 3b). The task invocation and communication operations are then randomly inserted into the control flow of the destination process (part (c) of the figure). Finally, the needed communica-

tion operations are randomly inserted into the control flow of the originating process (Figure 3d). There are similar transformations for moving other behavioral structures such as conditional branches and loops from one process to another.

For the inter-process code motion transformation to be applicable, there must be control sequences in the two processes that are executed under the same conditions. For this reason, it is useful to have a transformation that copies control structures (loops and conditionals) from one process to another. Our code copying transformation is similar to the code motion transformation.

To explore alternate global schedules, it is also necessary to transform the control flow within a process, re-ordering task invocations, process interactions, and control structures relative to one another. This is accomplished in our case by *intra-process code motion* transformations, which remove an item from the control flow of a process then randomly reinsert the item into the same control flow.

Repeatedly applying transformations to parts of the design chosen at random provides us with a way of randomly exploring alternative global schedules and process partitions. In addition to moving code from one process to another, it is also necessary to have the ability to create a new processes, so that alternative amounts of concurrency can be considered. In our case, inter-process code motion has the option of creating a new process and moving code into it.

Because these transformations have an unpredictable effect, it is also necessary to provide some way of undoing them if the effects are not desirable. This is accomplished in our case by a mechanism for noting the changes made to the design state as the transformations progress, and reversing those changes if the transformation needs to be undone.

While the transformations outlined above have an unpredictable effect, some transformations are known to be helpful in all cases. These transformations, which we call *clean-up* transformations, perform modifications such as simplifying complex or redundant inter-process communication, eliminating useless control structures, and destroying empty processes. In our case, the clean-up transformations are performed after every code motion transformation. The clean-up transformations must be done before the effect of the intra- or inter-process code motion is evaluated, since creating a clean-up opportunity is clearly part of the merit of a code motion transformation. However, since the clean-up transformations are contingent upon the code motion transformation being accepted, it is necessary for the clean-up transformations, like the code motion transformations, to be undoable.

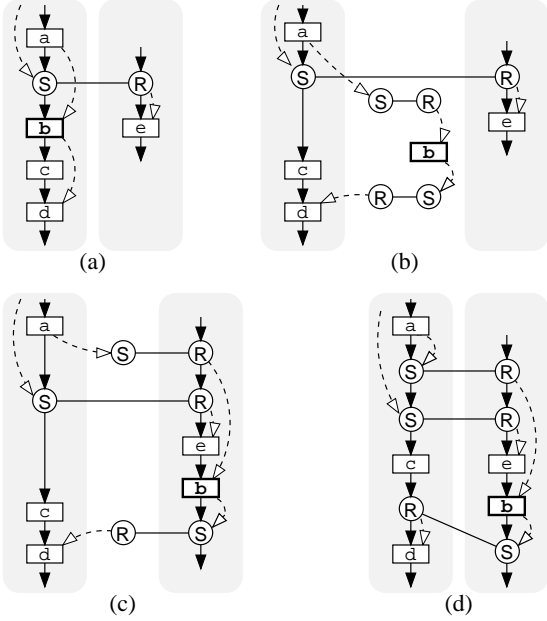All code motion and clean-up transformations must

Figure 3: Example of inter-process code motion.

take care not to create a deadlock situation among the processes. Since process communication operations imply synchronization, it is important that these operations be scheduled correctly.

We address the deadlock avoidance problem by use of the *port/data flow graph*. The port/data flow graph contains a node for each task invocation, control structure, and synchronization operation. The graph contains a directed edge for each data dependence and control flow relation and an undirected edge for each synchronization (between the send and receive operation). Deadlock can be avoided as long as this graph is acyclic. As the code motion and clean-up transformations are performed, the port/data flow graph is examined to ensure that no change to the design induces a cycle in the graph.

2.3   Design state evaluation

The design state is evaluated based on two characteristics: the amount of custom hardware required to implement the design and the execution time (in terms of throughput) of the system.

The amount of hardware required to implement the design can be estimated based on the hardware implementation statistics for each of the tasks that are mapped to hardware processes. The simplest way to estimate the required amount of hardware would be to simply sum up, for each task that is to be implemented in hardware, the amount of custom hardware required to implement that task. However, this strategy does not take into account the possibility that tasks within a process will be able to

share hardware resources. Hardware sharing is an important factor to take into account, since it tends to group similar computations into the same process, reducing the cost of the hardware implementation. To include the possibility of hardware sharing in our estimation we use a multi-parameter estimation technique, similar to that of [10], that is based on the results of behavioral synthesis.

The estimate of the system performance should take into account the impact of each task execution on the overall performance. Tasks that are invoked from within an inner loop, for example, will likely have more impact than others. Determining which functions of the system are the most time-critical, however, is difficult and cannot simply be done at the outset, since transformations may alter which tasks are in the critical path. For this reason, we have chosen to evaluate the performance of the system by performing an abstract simulation after each transformation.

The abstract simulation relies on knowledge of the sequence of directions a branch takes each time it is encountered. Gathering this information, which we refer to as the *execution trace*, can be done before beginning the optimization by simulating the entire system description using a sample input data set and recording the sequence of directions taken by the each conditional branch. Note that the transformations we make to the design do not alter the execution trace.

The abstract simulation is performed by simply summing up the delays required by the processes as they invoke tasks or perform conditional branches. Since the directions of all the branches are known ahead of time, the abstract simulation need only track the computation time required by each task and the delays incurred due to process communication, and not the actual computation being performed. Because of this, the abstract simulation can be done quickly, even for fairly long execution traces.

Basing performance evaluation on information derived from simulating the system description on a sample data set is also the general approached presented in [12]. In our case, however, we wish evaluate the effects of process interaction and global scheduling, and therefore must perform the simulation many times as we explore the design space.

2.4   Optimization

The optimization technique we use is non-deterministic iterative improvement in which the current design state is randomly perturbed, the resulting design state is evaluated, and the perturbation is either accepted or rejected according to some acceptance criteria. If the perturbation is accepted, the new design state becomes the current design state for further iterations of the optimization. Three basic elements are need to implement this type of opti-

mization: a set of design perturbations, a means for evaluating a design state, and a set of criteria for accepting or rejecting design perturbations.

The design space perturbations we use are the non-deterministic intra- and inter-process code motion transformations followed by a series of clean-up transformations described above. We evaluate design points using a cost function that is a weighted sum of the hardware required to implement the design and the throughput the design is able to achieve. We use simulated annealing to define the criterion for accepting or rejecting design perturbations.

3 Results

The technique described in the previous section has been implemented in Co-SAW, the Co-design System Architect's Workbench, which comprises just over 18,000 lines of C code. We used Co-SAW to partition two systems into concurrent hardware and software processes under a variety of constraints. In all cases we assumed that there would be only one CPU available and that the software would be statically scheduled (i.e., there is no runtime software to do scheduling). Even though the CPU can run only one process, we made no assumption about the number of hardware processes that would be present.

The two behavioral descriptions we used in our experiments were 'lzw-des,' which performs LZW data compression followed by DES encryption, and 'jpeg,' which implements the JPEG image compression standard. The 'lzw-des' design example is described in approximately 330 lines of behavioral Verilog. The 'jpeg' design example consists of 400 lines of Verilog.

Table 1 shows several of the results obtained for the 'lzw-des' example. The first column shows the hardware size constraint that was specified (in arbitrary units). The remaining columns show the resulting hardware size, system performance (in terms of cycles per input byte), and numbers of hardware and software processes. The rows labeled "all SW" or "all HW" in the first column are designs done by hand. This table shows that when the hardware size is less than 3500, only incremental gains in performance can be realized by increasing the amount of hardware available. When a sufficient amount of custom hardware is available, however, the optimizer is able to realize a significant performance improvement by creating two fairly independent concurrent processes, one in software and the other in hardware. As the hardware size constraint is increased from 4000, gains in performance are again incremental until 7500 units of custom hardware are available, at which time a quantum improvement in performance can be achieved by placing all of the time-critical computation into hardware. Another step in performance improvement is seen when 8000 units of hard-

HW size constraint	HW size result	performance (cycles/byte)	processes SW	HW
all SW	0	122.9	1	0
500	0	122.9	1	0
1000	614	121.3	1	1
1500	1208	119.4	1	1
2000	1208	119.4	1	1
2500	2426	112.4	1	2
3000	2426	112.4	1	2
3500	2426	112.4	1	2
4000	3794	40.0	1	1
4500	3794	40.0	1	1
4750	4692	39.0	1	2
5000	5012	33.0	1	2
5500	5012	33.0	1	2
6000	5012	33.0	1	2
6500	5012	33.0	1	2
7000	5012	33.0	1	2
7500	7063	16.6	1	1
all HW	7692	16.6	0	1
8000	7790	10.2	1	2
all HW	8189	9.8	0	3

Table 1: Results for the LZW-DES design example.

ware are available, when it becomes possible to further increase the concurrency by creating more independent hardware processes.

The results of the optimization are also affected by the sample data set on which the performance evaluation is based. With the 'lzw-des' example, we have noticed that when the data set exhibits an average to low compression ratio, the DES computation represents the performance bottleneck. In this case, the optimizer attempts to use the hardware resources to implement that part of the system. When the data set exhibits a high compression ratio, however, the LZW compression computation represents the performance bottleneck. The optimizer responds, in this case, by using custom hardware to implement the compression routines.

The results in Table 2 show the experimental results obtained for the 'jpeg' example. The results of the optimization are shown for different hardware size constraints. Again, the optimizer is able to make effective use of the custom hardware resources over a wide range of constraints.

The experimental results we have obtained illustrate the benefit of addressing concurrency optimization and hardware-software partitioning simultaneously. The extra degrees of freedom enable the optimizer to explore a wide range of implementation options, ranging from incremental performance gains achieved by placing selected computations in hardware, to substantial performance gains

HW size constraint	HW size result	performance (cycles/byte)	processes SW	HW
all SW	0	679	1	0
2000	258	672	1	1
4000	2862	442	1	2
6000	5840	315	1	1
all HW	6824	183	0	1

Table 2: Results for the JPEG design example.

achieved by forming concurrent hardware and software processes.

4   Conclusion

Optimization of mixed hardware/software systems of the type we have described can be seen as an instance of multiple-process behavioral synthesis, since the controller and datapath of an off-the-shelf CPU are necessarily separate from the controller(s) and datapaths(s) implemented with custom hardware. Multiple-process behavioral synthesis raises the possibility of altering the concurrency implied by the initial system behavioral description to suit the implementation or to improve system performance. However, several problems of multiple-process behavioral synthesis (process partitioning, global resource allocation, and global scheduling) appear to be interdependent and, therefore, difficult to solve sequentially.

We have described a technique for addressing multiple-process behavioral synthesis as it relates to the co-synthesis of mixed hardware/software systems. We address the interdependency of the synthesis problems by attempting to optimize them simultaneously using iterative-improvement techniques. This enables an optimizer to consider concurrency issues at the same time as hardware-software trade-offs. The result is the ability to explore a wide range of design options, ranging from simply placing computations in hardware to forming concurrent hardware and software processes. Experimental results show that this approach is effective at optimizing the performance of a mixed hardware/software system over a wide range of constraints on the amount of custom hardware available. The experimental results also illustrate that significant gains in performance can be achieved by taking advantage of the concurrency inherent in hardware/software systems.

References

[1] W. Wolf and R. Manno, "High-Level Modeling and Synthesis of Communicating Processes Using VHDL," *IEICE Trans. on Information and Systems*, vol. E76-D, no. 9, pp. 1039–46, 1995.

[2] P. Eles, K. Kuchcinski, Z. Peng, and M. Minea, "Synthesis of VHDL Concurrent Processes," in *Proceedings of EURO-DAC/EURO-VHDL '94*, 1994.

[3] R. A. Walker and D. E. Thomas, "Design representation and transformation in The System Architect's Workbench," in *Proceedings: ICCAD-87*, 1987.

[4] J. W. Hagerman, *Synthesis of Multiple Process Digital Systems*. PhD thesis, CMU, November 1994.

[5] T. B. Ismail, K. O'Brien, and A. Jerraya, "Interactive System-Level Partitioning with PARTIF," in *Proceedings of EURO-DAC/EURO-VHDL '94*, 1994.

[6] J. Henkel, R. Ernst, U. Holtmann, and T. Benner, "Adaptation of partitioning and high-level synthesis in hardware/software co-synthesis," in *Proceedings: ICCAD-94*, 1994.

[7] R. K. Gupta and G. De Micheli, "Hardware-software cosynthesis for digital systems," *IEEE Design & Test of Computers*, vol. 10, no. 3, pp. 29–41, 1993.

[8] D. D. Gajski, F. Vahid, and S. Narayan, "A system-design methodology: executable-specification refinement," in *Proceedings: ED&T Conference*, 1994.

[9] D. E. Thomas, J. K. Adams, and H. Schmit, "A Model and Methodology for Hardware/Software Codesign," *IEEE Design & Test of Computers*, vol. 10, no. 3, pp. 6–15, 1993.

[10] F. Vahid and D. D. Gajski, "Incremental Hardware Estimation During Hardware/Software Functional Partitioning," *IEEE Trans. on VLSI Systems*, vol. 3, no. 3, 1995.

[11] J. K. Adams, J. A. Miller, and D. E. Thomas, "Execution-Time Profiling for Multiple-Process Behavioral Synthesis," in *Proceedings: ICCD-95*, 1995.

[12] W. Ye, R. Ernst, T. Benner, and J. Henkel, "Fast Timing Analysis for Hardware-Software Co-Synthesis," in *Proceedings: ICCD-93*, 1993.