A Clustering Technique to Optimize Hardware/Software Synchronization

Junyu Peng
CECS, UC Irvine
Irvine, CA 92697
Tel: 949-824-8059
Fax: 949-824-8919
e-mail: pengj@cecs.uci.edu

Samar Abdi
CECS, UC Irvine
Irvine, CA 92697
Tel: 949-824-8059
Fax: 949-824-8919
sabdi@cecs.uci.edu

Daniel Gajski
CECS, UC Irvine
Irvine, CA 92697
Tel: 949-824-4155
Fax: 949-824-4155
e-mail gajski@uci.edu

Abstract— In this paper we present a scheme for reducing the amount of synchronization overhead needed between components, after HW/SW partitioning, to preserve the original control flow of the specification. Since traffic between components is expensive, our scheme can significantly enhance the performance of the system implementation. Our optimization technique dynamically groups the tasks in the specification such that synchronization for different tasks can be shared. The grouping depends on the partitioning decision, and hence, is performed during the generation of the partitioned model. We apply our grouping algorithm for various partitions on system level models of industry standard designs. The experimental results show significant reduction in synchronization overhead compared to the unoptimized model.

I. Introduction

In our system design methodology, we start with a executable specification model that encapsulates only the functional requirements of the design. Such an executable model is a hierarchical collection of tasks, referred to as *behaviors*. The behaviors are composed hierarchically using standard control flow constructs found in System Level Design Languages (SLDLs). The lowest level behaviors that cannot be further decomposed are referred to as *leaf* behaviors. The HW/SW partitioning problem is to decide which behaviors are to be implemented in hardware and which ones are to be implemented in software, such that all design constraints are satisfied.

The HW/SW partitioned model is usually a master-slave implementation, where SW is the master of the bus and invoking the tasks on the HW unit. The invocation of HW tasks is performed using synchronization messages over the system bus. In a simplistic implementation, the SW must invoke every leaf level behavior running on the HW. If the granularity of the model is fine and if there are several control dependencies across the components, then the resulting synchronization would be a huge overkill. The focus of our work is to minimize this expensive synchronization by sharing it for groups of leaf behaviors.

The rest of the paper is organized as follows. In section II, we give an overview of related work. In section III,

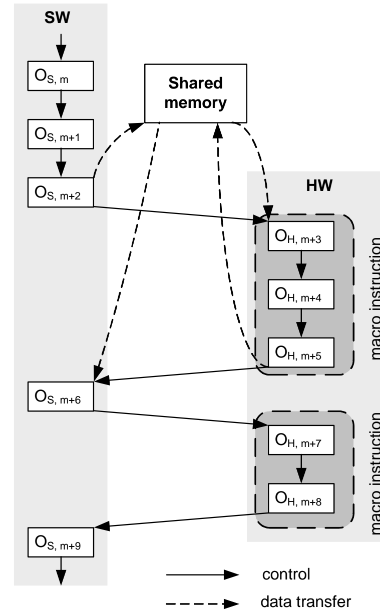

Fig. 1. Hardware/Software communication model

we give some definitions and a motivational example. In section IV, we describe our optimization algorithm. In section V, we present experimental results on an industrial scale design example to demonstrate the effectiveness of our technique. Finally, we draw conclusions and summarize our work.

II. Related Work

The foundation of our work is the communication scheme for HW/SW partitioned model, presented by Henkel in [2]. In Henkel's approach, the communication schema for a simple hardware/software architecture as shown in 1. Essentially, the software executing on the processor may invoke the execution of macro instructions on the hardware unit. In the first step, the processor writes the data needed by the HW unit to a shared memory. The data also contains the ID of the targeted macro instruction in HW. The HW behavior, then, checks for the ID and executes the corresponding macro instruction and stores the result in the shared memory. Each time

the hardware is invoked, the system bus is used. In order to avoid this frequent bus access, they compose the macro-instruction in a manner such that the total communication overhead is minimized. Their approach basically attempts to hierarchically group partitioned objects based on the control structure of the input specification. Therefore, if all objects of a branch construct are mapped to hardware, they will only need to synchronize the hierarchical object instead of each of them individually. For instance, as shown in Figure 1, the three sequential tasks, $m+3$ through $m+5$ are composed into one big macro instruction so that only one call is needed to execute all of them. Similarly, objects belonging to the same loop body and if-then-else constructs can be grouped to form larger macro instructions. However, their approach did not address arbitrary control flow between tasks.

The hardware/software partitioning problem has been researched extensively in the past [1] [3] [5]. Quite a few of the approaches consider the communication overhead after partitioning as a primary metric. Knudsen [4] proposed a coarse grain CDFG model that is suitable for hardware/software partitioning. Vahid [6] quantitatively defines communication as one of the closeness metrics for system-level functional partitioning. The closeness metrics can be used to guide clustering of tasks for partitioning. There are some similarities between our work and the work described in papers by Henkel and Knudsen. Essentially, a graph based data structure of the functional model is developed and then transformed based on prediction of partitioning results.

III. Deriving Control Flow Structure

A. Control Flow Graph

In our approach, the SLDL model is first translated into a *control flow graph*. The source code of the SLDL model contains common control constructs like **if-else**, **goto** and **break**. The corresponding control flow graph contains nodes, symbolizing a basic block and edges, representing control dependencies. Note that our control flow graph does not contain any special nodes that denote entry or exit for regular control structures like **loops** or **if-then-else** constructs.The partitioning information is incorporated in the graph by assigning different colors to the nodes.We will use the convention of coloring the nodes mapped to the hardware as black color and those to software as white.

B. Synchronization Abstraction

Typically, synchronization is realized using a rendezvous communication between the relevant points on each component. At each of these *synchronization points*, a handshake between the SW processor and the HW unit is performed using mechanisms such as status registers, interrupts or messages over abstract channels. For the example shown in Figure 1, a total of 4 messages are exchanged between SW and HW. We can identify this by counting the edges that go across the SW and HW partition boundaries. We will measure the amount of synchronization in terms of number of messages (NoM), which may depend on how many times each node is executed. Therefore two behaviors in a loop body, mapped to different components will exchange as many synchronization messages as the loop count.

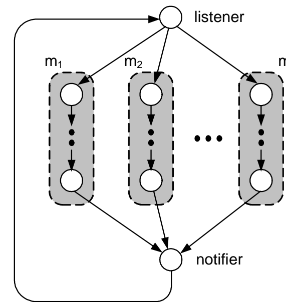

Fig. 2. Control flow graph of hardware co-processor.

C. Control Flow Template for HW

Our template for the control flow graph for the hardware unit is shown in Figure 2. The *listener* node waits for a message from the software processor to start the execution of one of the macro instructions mapped to the hardware. The synchronization message contains the ID of the macro instruction that needs to be executed. Once the macro instruction has finished executing, the control reaches the *notifier* node, which notifies the SW of the completion of the hardware call and the control returns to the *listener* node for the next hardware call.

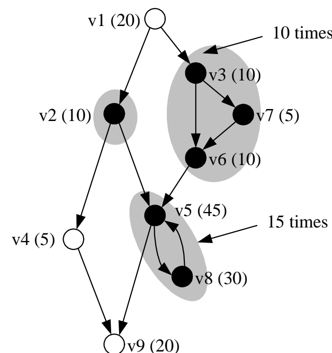

Fig. 3. CFG node clustering for reducing synchronization messages.

D. Motivational Example

For the simple example shown in Figure 1, we have seen that the grouping of smaller blocks into two larger macro instructions helps reduce NoM exchanged between SW and HW. If the 5 blocks, mapped to hardware, were synchronized individually, a total of 10 messages would be

needed. Clearly, a lot of these messages would be redundant, since only 4 synchronization messages are needed for the same execution. For this example, the grouping is straightforward since all blocks execute sequentially. However, control flow in a typical application is more complex.

A complex control flow example is shown in Figure 3. If we synchronize each black node individually, NoM is 220. Remember that we must also take loop counts into account while calculating NoM. If we were to cluster the nodes mapped to HW into three larger groups as shown in Figure 3, the NoM count goes down to 70 without changing the template structure on the HW side. Our goal is to have minimum possible number of node groups, each group having a single control entry and exit.

IV. Optimal clustering of CFG nodes

A. Valid Group

The validity of the groups depends on the following three conditions:

1. All nodes in the group have the same color;
2. **Single entry**: in each group, only one node, called *entry*, has edge(s) from outside the group;
3. **Single exit**: in each group, only one node, called *exit*, has edge(s) going outside of the group.

The first requirement is straightforward since grouping is local to each component. The single entry requirement ensures that the decision to execute a block of code (represented by the non-entry node in a group) is taken within the HW unit. Hence, the control does not need to be sent back to the SW to execute this internal node. A similar argument can be made for the single exit policy. If the control flow from a node cannot not lead to a node in SW, then the control may not return to SW. In the worst case, each node by itself is a valid group and can thus be synchronized individually.

The procedure to derive a minimum set of valid groups is shown in Algorithm B. In order to satisfy the first requirement that the nodes in the same group have the same color, the algorithm first constructs a *polar graph* by keeping only the black nodes which are mapped to hardware. A *source* and a *sink* node are used to replace the edges between white nodes and black nodes.

B. Algorithm for optimal clustering of CFG nodes

The set of nodes that have edges from the *source* must be *entry* nodes of separate groups. Similarly, the set of nodes that have edges to the *sink* must be *exit* nodes of their respective groups. In addition, if a node can be reached by more than one *entry* nodes, then this node must be in a different group. This is because every node can be in one group only and each group can have only one *entry* node. Conversely, if a node has edges to more than one *exits*, then this node must be in a different group as

Algorithm 1 Cluster nodes

1: add *source* and *sink* to G;
2: **for all** $e_{u,v} \in$ G **do**
3:   **if** u is black and v is white **then**
4:     make u an exit by adding $e_{u,sink}$;
5:   **else if** u is white and v is black **then**
6:     make v an entry by adding $e_{source,v}$;
7:   **end if**
8: **end for**
9: remove all white nodes and their edges;
10:
11: **for all** vertex v in BFS order from *source* **do**
12:   **if** v is reachable from multiple entries **then**
13:     remove all incoming edges to v;
14:     make v a new entry if it is not yet;
15:     make v's predecessors new exits;
16:   **end if**
17: **end for**
18:
19: **for all** vertex v in BFS from *sink* **do**
20:   **if** v can reach multiple exits **then**
21:     remove incoming edges to v;
22:     make v a new exit if it is not yet;
23:     make v's successors new entries;
24:   **end if**
25: **end for**

well. Our algorithm does a reachability analysis for each node in a breadth-first-search order. We incrementally remove edges between nodes that have been mapped to different groups. The algorithm terminates when all nodes have been assigned to a group and there are no edges remaining across groups.

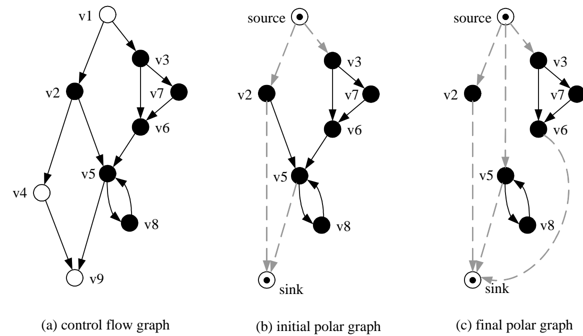

Fig. 4. Example of the Algorithm.

C. Example

A walkthough of our algorithm is shown for the CFG example in Figure 4. In the first step, a polar graph is derived by introducing *source*/*sink* and eliminating the white nodes ($v1, v4, v9$) and their edges as shown in 4 (b). The *source* and *sink* nodes can be distinguished from behavior nodes by a dot in their center. Note that $v2$ and $v3$ are *entry* nodes since they have incoming edge from

Partitions		(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
Before clustering	# nodes	4	5	5	3	3	4	3	5
	# messages	1630	2388	3260	1467	1516	2176	1524	2 282
After clustering	# groups	2	3	3	2	2	2	3	2
	# messages	815	1306	1956	815	1084	872	1524	815
Reduction in # messages		50%	45%	40%	44%	28%	60%	0%	64%

TABLE I
Synchronization message reduction through grouping.

source, while $v2$ and $v5$ are *exit* nodes because they have edges to the *sink*. Broken arrows are used to distinguish edges from or to *source* or *sink* node, respectively.

The next step of the algorithm is to perform reachability analysis for all nodes in a breadth-first order, starting from the *source* node. We do not take any action when $v2$, $v3$, $v6$ and $v7$ are visited since each of them is reachable only from one *entry* node. However, $v5$ can be reached from both $v2$ and $v3$. Therefore, we remove the two incoming edges $e_{v2,v5}$ and $e_{v6,v5}$. At the same time, we make $v5$ a new *entry* node by adding edge $e_{source,v5}$. Similarly, $v6$ is made into a new *exit* node by adding edge $e_{v6,sink}$. It must be noted that $v2$ was already an *exit* node. Due to the removal of some edges, $v8$ is now reachable only from a single *entry* node, $v5$.

The final step of the algorithm is to do a similar reachability analysis,this time starting from the *sink* node. For example, one of the possible visiting order for the nodes is: $v2, v5, v6, v8, v7, v3$. It is easy to see that no new groups are created since every node can only reach one *exit* node. The final graph, shown in 4 (c), contains three disconnected groups. These groups satisfy all the validity requirements defined in Section XSA.

V. Experimental Results

A tool based on the the clustering algorithm was implemented in C++. The tool takes the colored control flow graph as its input and produces the set of groups that satisfy the requirements in Section A. Experiments were performed with a GSM Voice Codec application used in cellular phones. The functional specification consisted of more than 9000 lines of SpecC code. The nodes of the corresponding control flow graph represent the 13 behaviors in the specification. Simulations were performed for the specification model using 3.26 second long voice samples. The clustering algorithm was tested for 8 different SW/HW partitions. For each of the 8 solutions, we obtained the NoM for un-optimized model by synchronizing each of the hardware nodes individually. We then used the clustering technique to group hardware nodes. The results are shown in Table I. We see a NoM reduction ranging from 28 to 60 percent over a variety of partitioning decisions.

VI. Conclusion

In this paper, we presented a technique for reducing SW/HW synchronization overhead. This technique can handle specification with irregular control constructs, such as **goto** and **break**. The communication overhead was minimized by sharing synchronization for groups of behaviors mapped to a component. The gain was measured as the reduction in number of synchronization messages exchanged between software and hardware components. Our experimental results demonstrate that our technique is very effective in reduction of synchronization overhead. This technique can be easily integrated as a post processing step in HW/SW partitioning of system level models.

References

[1] R. Gupta and G. DeMicheli. Partitioning of functional models of synchronous digital systems. In *Proceedings of the International Conference on Computer-Aided Design*, pages 216–219, November 1990.

[2] J. Henkel. An approach to automated hardware/software partitioning using a flexible granularity that is driven by high-level estimation techniques. *IEEE Transactions on Very Large Scale Integration(VLSI) Systems*, 9(2), April 2001.

[3] Kalvade and E. Lee. The extended partitioning problem: Hardware/software mapping, scheduling and implementation-bin selection. In *International Workshop on Rapid System Prototyping*, June 95.

[4] P. V. Knudsen and J. Madsen. Graph based communication analysis for hardware/software codesign. In *Proceedings of the International Workshop on Hardware-Software Codesign*, September 1999.

[5] Z. Peng and K. Kuchkinski. An algorithm for partitioning of application specific systems. In *Proceedings of the European Design Automation Conference*, pages 316–321, March 1993.

[6] F. Vahid and D. Gajski. Closeness metrics for system-level functional partitioning. In *Proceedings of the European Design Automation Conference*, pages 328–333, September 1995.