Hardware / Software Partitioning for Multi-function Systems

Asawaree Kalavade
Bell Labs, Holmdel, NJ 07733.
kalavade@bell-labs.com

P. A. Subrahmanyam
Bell Labs, Holmdel, NJ 07733.
sabra@bell-labs.com

Abstract

We are interested in optimizing the design of multi-function embedded systems that run a pre-specified set of applications, such as multi-standard audio/video codecs and multi-system phones. Such systems usually have stringent performance constraints and tend to have mixed hardware-software implementations. The current state of the art in the hardware/software codesign of such systems is to design for each application separately. This often leads to application-specific sub-optimal decisions and inconsistent mappings of common nodes in different applications.

We use these as the guiding principles to formulate, as a codesign problem, the design and synthesis of an efficient hardware-software implementation for a multi-function embedded system. Our solution methodology is to first identify nodes that represent similar functionality across different applications. Such “common” nodes are characterized by several metrics. These metrics are quantified and used by a hardware/software partitioning tool to map common nodes to the same resource as far as possible. We demonstrate how this is achieved by modifying a traditional partitioning algorithm (GCLP) used to partition single applications. The overall result of the system-level partitioning process is (1) a hardware or software mapping and (2) a schedule for execution for each node within the application set. On an example set consisting of three video applications, we show that the solution obtained by the use of our method is 38% smaller than that obtained when each application is considered independently.

1. Introduction

Many of the recent research efforts in hardware-software codesign for embedded systems focus on the design of embedded systems optimized for a particular application. However, there is a growing class of embedded systems that need to execute a set of applications rather than just a single application. Such systems fall into two broad categories. The first consists of systems that execute multiple applications concurrently, e.g., set-top boxes with concurrent applications like audio, video, and web browsing. The second category includes multi-function systems that support multiple functions or applications of which only one is executed at any instant, depending on external factors. Such multi-function systems offer alternatives between various functionalities — the specific alternative is typically selected at run time. Consider, for example, an audio decoding system that decodes several audio algorithms such as AC3, MPEG2, ADPCM, LPC, etc. Any one of these decoding algorithms may be active at a particular time, based on the service quality requirements and the network bandwidth availability. Another example is a video encoding system that runs MPEG2, H.261, JPEG, etc. Depending on whether the user is watching a movie or conducting a video conference, one of these applications would run at a given time. In this paper we focus on the design of multi-function systems. Note that we use terms “function” and “application” interchangeably.

Due to time-to-market pressures as well as the inherent suitability of some parts of the applications to either hardware or software, it is quite common for such embedded systems to have mixed hardware-software implementations. The current state of the art in the codesign of these systems is to separately design for each application. Using this methodology, the resulting solution consists of separate implementations for each such application, possibly on the same chip or board. Although designing for each application independently reduces the design time, the resulting solution might have a larger system area. This is because optimizing the implementation for each application independently often leads to application-specific implementation decisions that do not necessarily yield the best overall cost-performance results. For example, focusing on individual applications might lead to a decision to use specialized (but not very reusable) hardware modules, whereas investing the same or comparable area in a programmable processor core might have yielded a module with a much higher degree of reuse across different applications. Further, since the applications in the mix are often related, there are often several commonalities between the applications. For example, a discrete cosine transform (DCT) function may occur in several video applications. When applications are considered independently, a hardware implementation may be selected for the DCT in one application, while a software implementation might be selected in the other application. However, from the viewpoint of optimizing the system as a whole, it might be beneficial to maintain consistency in mapping common nodes. For these reasons, we believe that, when designing for a multi-application set, it is important to consider all the applications in the set simultaneously, rather than designing for individual applications.

We use these as the guiding principles to formulate, as a codesign problem, the design and synthesis of an efficient hardware-software implementation for a multi-function embedded system. In Section 2, we define the multi-application codesign problem. In Section 3, we discuss some related work. In Section 4, we describe two methods to solve the problem. In Section 5, we discuss the use of the proposed methods with the help of an example application set consisting of three video applications. We compare the system area obtained when each application is considered independently, to the system area obtained by applying the two methods. The resultant solution is 8% smaller when the first method is used and 38% smaller with the second method.

2. Problem Definition

We begin by assuming that we are given a set $AP = \{A_1, \ldots, A_N\}$,
any one application may be active at run-time and its timing constraint that specifies the maximum allowable time $D_j$ for executing one iteration of the application.

The system architecture model assumed in this paper consists of three types of resources: (1) one or more programmable processor cores that execute the software component of the nodes mapped to software, (2) a set of hardware accelerators that implement the functionality of the nodes mapped to hardware, and (3) coprocessors, attached to the programmable processors, that accelerate the execution of certain nodes. We assume a template-based architecture, where the interface between the hardware and software components is selected by the designer from among a set of predefined mechanisms. The costs associated with these communication modules are abstracted. The actual number of processor cores and the types and numbers of hardware units used in the final implementation is determined by the system-level design process, as described in Section 4.

The multi-application codesign problem is formally defined as follows. Given a set $AP = \{A_1, A_2, \ldots, A_k\}$ of applications, where each application $A_j$ has a timing constraint $D_j$, design an implementation that can support all the applications from the given set. Any one application may be active at run-time and its timing constraints should be met. The design objective is to minimize the overall hardware area.

Given an architecture template and a set $AP$ of applications, the codesign task consists of determining (1) the mapping of nodes, in all the applications, to either hardware or software, and (2) the schedule for the execution of the nodes within each application. The mapping is used to construct the overall system architecture. The schedule is used to generate the software for each application. When a particular application runs on the final system, it follows its particular execution schedule.

3. Related Work

While we are not aware of any work that directly addresses the problem as we have defined it above, there have been several efforts in the related area of ASIP (Application Specific Instruction Processor) synthesis [1][2][3][4][5]. The ASIP design problem is to design a domain-specific processor by selecting the optimal “instruction set” for a class of applications. Typically, the class of applications is analyzed to find the most commonly used instructions and a datapath and controller for that instruction set is designed. The problem we are interested in here is to design a system-level hardware-software architecture optimized for a class of applications. We focus on finding commonalities between applications at a higher level of granularity than is typically considered when designing ASIPs. We do not address the design of the programmable processor itself, but focus on the optimal mapping of components of the applications to the processor or custom hardware accelerators.

Potkonjak et al. [6] address the problem of combining several concurrent tasks onto a single ASIC instead of designing a separate ASIC for each task. They discuss an iterative algorithm that combines tasks onto a single ASIC, based on their bitwidth requirements, register counts, source and destination locations, etc. We are concerned with a different problem here, that of selecting the best hardware or software implementation for each node in each application such that the overall system cost is minimized. Also, we are concerned with nodes at a higher level of granularity.

4. Solution Methodology

The proposed solution methodology for the multi-application codesign problem is shown in Figure 1. The set of specified applications is first analyzed to extract commonality measures across applications. In Section 4.1, we identify some measures, such as repetitions and performance-area trade-offs, which can be used to characterize nodes across all applications. We also present simple methods to quantify these measures and provide an intuitive explanation of how they can be used in the partitioning process. The hardware-software partitioning tool is then applied to each application. The partitioning algorithm uses the commonality measures to bias the mappings of nodes that are common across applications. In Section 4.3, we describe two methods to partition the nodes in the applications into hardware and software. These methods are based on a previously developed algorithm, called GCLP [7], used to partition a single application. The GCLP algorithm is summarized in Section 4.2.

4.1. Commonality Measures

The general approach for computing the commonality measures is as follows. First, the “common” nodes, i.e., nodes that represent similar functionality across different applications, are identified. A simple approach to identify common nodes is to have the user tag all nodes in all applications with a root name and parameters. Rootnames are matched across all applications. Parameters are used to detect exact or partial matches. An example of a partial match is that between the two nodes <DCT, 1 block> and <DCT, 8 blocks>. Each node is then tagged with a node type: nodes that match have the same node type. In the above example, both nodes can run on the same resource with minor changes to the control logic and are said to have the same node type. Alternatively, more sophisticated techniques such as template matching [8] can be used to detect matches. The set of common nodes consists of nodes that repeat over applications in the set.

![Figure 1 Solution methodology.](image-url)
Next, each node is analyzed to compute measures that characterize the node. We have identified some properties and propose simple techniques to quantify them. In this section, we describe the use of these metrics in the partitioning tool.

1. (R) Repetitions of a node: The repetitions ($R_i$) of a node $i$ is computed as the number of occurrences, across all the applications, of nodes of the same type as the type of node $i$. The repetitions measure can be used in two ways. A node that appears more frequently (high $R$ value) can be given a priority for a custom hardware implementation. In other words, if a node occurs many times in different applications, implementing it in custom hardware makes sense since it gets reused over many applications. Alternatively, the repetitions measure can be used to maintain consistency in mapping; when mapping nodes with a high $R$ value all its instances can be mapped to either hardware or software.

2. (PA) Performance-area ratio of a node: The performance-area ratio ($PA_i$) of a node $i$ is measured as the ratio of the performance gain (speedup achieved by implementing node $i$ in hardware) to the area penalty to be paid for the hardware implementation. Nodes with a higher $PA$ ratio indicate a higher benefit in being mapped to a hardware implementation.

3. (U) Urgency of a node: The urgency ($U_i$) of a node $i$ is computed as the number of times a node of the same type as that of node $i$ appears on the critical paths for the different applications. A node that appears on the critical path in a larger number of applications is more “urgent” and can be given preference for a faster implementation.

4. (C) Concurrency of a node: The concurrency $C_i$ of a node $i$ is the number of “concurrent” instances of the node. $C_i$ can influence the number of instances of node $i$ when it is implemented in hardware. For example, if the concurrency for node $i$ is 2, it indicates that, on an average, two instances of node type $i$ are active at any time and hence we can include two hardware accelerators for node type $i$.

A commonality vector $<R_i, PA_i, U_i, C_i>$ is computed for each node. Nodes with the same type have the same commonality vectors. For each property, the values are normalized across all the nodes. The cumulative effect of the properties can be incorporated by combining different measures. In this section, we show how some of these measures are used in the partitioning process.

4.2. Partitioning Single Independent Applications

In this section, we briefly summarize the original GCLP algorithm [7], which is used to partition a single application independent of other applications. In Section 4.3.1 and Section 4.3.2 we describe two modifications to the GCLP algorithm that can be applied to the multi-application codesign problem.

The flow of the GCLP algorithm is shown in Figure 2. $N$ represents the set of nodes in the graph. $N_U$ is the set of unmapped nodes at the current step. $N_U$ is initialized to $N$. For each node $i$, a local attribute $\delta_i$ that quantifies the preference of $i$ to a hardware or a software mapping is computed. The algorithm then maps one node per step. At the beginning of each step, the global time criticality measure $GC$ is computed. $GC$ is a measure of how critical time is at that step, i.e., given the nodes already mapped so far and the required finish time of the application, $GC$ is indicative of the slack available at that instant. Unmapped nodes whose predecessors have already been mapped and scheduled are called ready nodes. A node is selected for mapping from the set of ready nodes using an urgency criterion. $GC$ and $\delta_i$ of the selected node $i$ are then used to select the mapping objective as shown. Using this objective, the selected node is assigned a mapping ($M_i$). The mapping is also used to determine the start time for the execution of node ($t_i$). This set of start times defines the schedule for the system. The process is repeated $|N|$ times over all nodes.

4.3. Partitioning Multiple Applications

We now describe two methods for partitioning multiple applications. These methods modify the GCLP algorithm.

4.3.1 Method A

In this method, the partitioning algorithm is modified to incorporate two basic factors that influence the mapping of a node when multiple applications are considered:

1. If a node is repeated in several applications (high $R$ measure), it might be beneficial to bias its mapping toward hardware. By sharing a hardware implementation for repeated nodes, other less frequently appearing nodes may get mapped to software, leading to an overall reduction in hardware area.

2. If a node takes a small area when implemented in hardware, relative to all nodes in all applications, and if the difference in software and hardware execution times is high (high $PA$ measure), it might be beneficial to bias the mapping of the node toward hardware. This may free up the software resource for nodes that might otherwise be expensive in hardware.

These two factors can be put to use when considering the mapping of multiple applications. We define for each node $i$, a commonality measure $CM_i = \rho R_i + \pi PA_i$, where $R_i$ is the repetitions measure, $PA_i$ is the performance-area trade-off measure, and $\rho$, $\pi$ are the weights assigned to the two factors.

![Figure 2 Flow of GCLP Algorithm](image-url)
ρ and π are user-defined weights. CM_i is normalized over all the nodes in the application. As shown in procedure A below, the CM_i measure is used to bias the threshold in the GCLP algorithm.

**Procedure A**

1. Identify “common” nodes and count the number of instances of each common node over all the applications. The value of the repetitions measure R_i is obtained by normalizing the number of instances with respect to the largest number thus obtained; this yields a value for R_i between 0 and 1.
2. Compute the performance-area trade-off measure for each node i as PA_i = (ts_i - th_i) / ah_i, where ts_i and th_i are the execution times of node i in software and hardware respectively and ah_i is the hardware area for node i. Normalize PA_i over all the nodes to a value between 0 and 1.
3. Compute for each node i, the commonality measure \( CM_i = \rho R_i + \pi PA_i \) using user-specified weights \( \rho \) and \( \pi \). Normalize \( CM_i \) to a value between 0 and 1.
4. For each application \( A_a \), run GCLP-A(\( A_a \)).

**Algorithm GCLP-A**

This is a variant of the GCLP algorithm described in Figure 2, where step 3 in GCLP is replaced as follows:

3. Determine mapping \( M_i \) for i
   3.1 \( \text{threshold} = 0.5 - CM_i \)
   3.2 if \( GC > \text{threshold} \) \( M_i \) = hardware, else \( M_i \) = software

The interpretation of step 4 is as follows. For nodes with a non-zero \( CM_i \) value, the threshold is reduced to below its default value of 0.5. The GC at that step of the algorithm is compared to this modified threshold. If the GC is above the threshold, a hardware mapping is selected. By lowering the threshold with the commonality measure, a hardware bias is introduced. If GC is below the threshold, a software mapping is used. The mapping decision for common nodes is thus made by considering the combination of application-specific requirements (dictated by GC) as well as inter-application demands (modeled by \( CM_i \)). Nodes that are not common get mapped such that feasibility is met, while still attempting to minimize the total area.

**4.3.2 Method B**

This method attempts to maintain a “consistency” in the way common nodes are mapped. The key idea is to map the applications in a particular order and to propagate information about the mapping decisions made. The propagation allows information about mapping decisions within an application to be shared between applications. The procedure for method B is summarized below.

**Procedure B**

1. Compute commonality measures R and PA for all nodes, as described earlier.
2. Compute application criticality (AC) for all the applications in the set. AC is computed as \( \Sigma ts_i/D_j \), where ts_i is the execution time of node i when implemented in software, and D_j is the required finish time for application A_j. The smaller the ratio, the fewer the hardware resources required to implement A_j and the lower the criticality of the application.
3. Order the applications by AC such that the most critical application (largest AC) is considered first.
4. For each \( A_a \) in this order, run GCLP-B(\( A_a \), \( shared \_ mapping \))

**Algorithm GCLP-B**

This is a variant of the GCLP algorithm described in Figure 2, where step 3 in GCLP is replaced as follows.

3. Determine mapping \( M_i \) for i:
   3.1 if \( \text{computed\_shared\_mapping}(i) == 1 \) { /* node type mapped before */
       if \( \text{shared\_mapping}(i) == \text{hardware} \)
         \( \text{threshold} = 0.5 - R_i /* lower threshold */ \)
       else if \( \text{shared\_mapping}(i) == \text{software} \)
         \( \text{threshold} = 0.5 + R_i /* raise threshold */ \)
   } else /* no earlier shared mapping */
   \( \text{threshold} = 0.5 - PA_i \)
3.2 if \( (GC > \text{threshold}) \) \( M_i \) = hardware, else \( M_i \) = software
3.3 Update shared mapping

The interpretation of step 4 is as follows. Let \( i \) be the node being considered at a particular step. If a node of the same type as node \( i \) has been mapped by a previously considered application, the goal is to try to preserve that mapping for node \( i \). This is accomplished by biasing the threshold in the direction of the previous selection. Thus, if the earlier mapping was hardware, the threshold is lowered by the repetitions measure \( R_i \), favoring a hardware implementation for this node. Similarly, if a node of the same type as node \( i \) has been previously mapped to software, the threshold is raised by the repetitions measure \( R_i \), thus favoring a software implementation. Note that the mapping of node \( i \) is not hardcoded to the mapping selected previously; instead, it is “biased” toward that mapping. By lowering the threshold for a node that has a previous mapping in hardware, its chances of getting mapping to hardware are increased (see 3.2). However, the state of the current application, as reflected by GC, is also considered. A hardware mapping is selected only if GC is greater than the modified threshold. Thus the mapping of common nodes is biased towards a mapping that maintains consistency across all applications, while making sure the application-specific constraints are also taken into consideration. If no node of the type of node \( i \) has been mapped before, the PA measure is used to select its mapping. In this case, this tries to select the best possible mapping for the node, considering the effect of the current application only.

The order in which applications are selected is important in this method. One possible approach is to select the most critical application first and map its nodes in a way that best meets the timing constraints. Other nodes then follow the mapping decisions made by the more critical applications, unless their local preference strongly dictates otherwise. We are also experimenting with other orderings of applications.

**5. Numerical Results**

In this section we report the results obtained for designing an implementation for a system running a set of video applications. In Section 5.2, we present the solution obtained when each application is considered independently. The solutions obtained by using methods A and B are discussed in Section 5.3 and Section 5.4 respectively. The solutions obtained by using the methods proposed in this paper are found to be superior (have smaller area) to the solution obtained when each application is considered independently. We have implemented the GCLP algorithm and its
proposed variants. Each run through the partitioning algorithm takes less than a second of CPU time on a Sparc20.

5.1. Application set and system architecture

We consider the application set consisting of: (1) MPEG2 video encode (M2E), (2) H.261 decode and encode (H), and (3) MPEG2 video decode (M2D). Such a mix is representative of the different functions running on an add-on card in a laptop PC, where the user may be watching a movie (MPEG2 video decode), or conducting a video conference (H.261 encode/decode), or transmitting video data (MPEG2 video encode). Table 1 summarizes the details of the applications in the application mix and their timing constraints. Several nodes are repeated over the different applications (e.g.: the node Inverse Quantizer appears 4 times, the node Motion Estimator appears 2 times).

We assume a system architecture as shown in Figure 3, consisting of a single programmable processor (PP) and multiple coprocessors (CP) and hardware accelerators {HW}; the actual number of coprocessors and the number and types of hardware accelerators are determined by the partitioning algorithm. The programmable processor can implement all the nodes, representing the software implementation. The coprocessor can implement some of the nodes. We assume a SIMD vector coprocessor that can implement the following functions: dct/idct, quantizer/iquanti zier, Frame Add/Sub, Motion compensation, and Loop Filter. The hardware accelerators implement specific functionality. We assume hardware accelerators for nodes such as a motion estimator, and a variable length encoder. Table 2 summarizes the different resource types available. The software execution times for the nodes are measured by a detailed simulation on a media processor. The hardware execution times and areas for the nodes are abstracted from an actual implementation of a video conferencing system developed in our lab. Different system implementations are quantified by a “system cost”, which is the total hardware area required over and above the programmable processor.

5.2. Experiment 1: Independent Mapping

We first run each application independently through the original GCLP algorithm. Table 3 summarizes the resultant hardware resources used for each application and the total system area. The total area for the application set is computed by summing the areas of the union of the hardware resources used in the three applications. Note that H requires two instances of CP.

5.3. Experiment 2: Method A

Recall that in method A, \( \text{threshold} = 0.5 - CM_{ij} \), where \( CM_{ij} = \rho R_i + \pi PA_i \). We present three sets of results with \((\rho, \pi)\) taking

<table>
<thead>
<tr>
<th>Case</th>
<th>App.</th>
<th>Hardware Resources Used</th>
<th>System Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2A)</td>
<td>CM = R</td>
<td>Set CP, CP, HW1, HW2, HW1, HW3, HW4, HW8, HW10</td>
<td>206</td>
</tr>
<tr>
<td>(2B)</td>
<td>CM = PA</td>
<td>Set CP, CP, HW1, HW2, HW3, HW8, HW10, HW11</td>
<td>224</td>
</tr>
</tbody>
</table>

Table 4: Results of Experiment 2
values (1,0), (0,1), and (1,1). Table 4 summarizes the results. Recall that the goal of this method is to map nodes with high repetitions to hardware.

In case 2A, nodes idct, mc, sub, and dct are mapped to hardware (since they have higher repetitions), against software in experiment 1. This is achieved by changing the threshold values when mapping these nodes. As a consequence, nodes with lower repetitions such as vld and sink3 get mapped to software. This results in a 8% lower system area. In case 2B, the PA measures do not help improve the solution over experiment 1, since PA measures alone have the same effect as $\delta$ in the original GCLP. In case 2C, the combined effect of PA and $\delta$ is the same as the effect of $R$ alone. Thus, the use of repetitions measure in method A seems to reduce the system area by 8% in this example.

5.4. Experiment 3: Method B

Recall that in this method, applications are considered in a specific order and mapping state is maintained between applications. The goal is to maintain consistency when mapping nodes of the same type. The applications are considered in the order of application criticality (sequence M2E-H-M2D). Table 5 summarizes the results.

<table>
<thead>
<tr>
<th>Application</th>
<th>Hardware Resources Used</th>
<th>System Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>M2E</td>
<td>CP, HW1, HW2, HW8</td>
<td>103</td>
</tr>
<tr>
<td>H</td>
<td>CP, HW1, HW2, HW4, HW5, HW8, HW9, HW11</td>
<td>139</td>
</tr>
<tr>
<td>M2D</td>
<td>CP</td>
<td>65</td>
</tr>
<tr>
<td>Set</td>
<td>CP, HW1, HW2, HW4, HW5, HW8, HW9, HW11</td>
<td>139</td>
</tr>
</tbody>
</table>

Table 5: Results of Experiment 3

Method B gives a much better solution than that obtained in either of the first two experiments. In particular, the solution is 38% smaller than that obtained when applications are considered independently. This may be attributed to the fact that method B incorporates some of the design principles in method A and also takes the consistency into consideration. Note that the solution for M2D by itself is worse than in experiment 1. The advantage is a reduced overall system cost since the same coprocessor is used in all applications, instead of special hardware accelerators that were used for this application when considered independently.

6. Conclusion

We have formulated, as a codesign problem, the design and synthesis of an efficient hardware-software implementation for an embedded system that runs a pre-specified set of applications. The goal is to design an implementation that can support all the applications from the given set. Any one application may be active at run-time and its timing constraints should be met. The design objective is to minimize the overall area of the system.

Although this problem can be viewed as one that involves an ASIP (application specific instruction processor) design, we have intentionally formulated the problem in a manner that avoids automatic processor design. Instead, we assume that the architecture is selected from among a set of given templates and that the processor core is available. Further, the applications are assumed to be specified at a "coarse" level of granularity. These assumptions have been made with a view to reducing the complexity of the problem, while still allowing for solutions that are useful in practice.

The key idea of our approach is to analyze the set of applications to extract commonalities across nodes in different applications. We identified several measures that characterize the nodes and defined ways to quantify them. These measures are used in the partitioning process to bias the mapping of a node. In general, the mapping decision for common nodes is made by considering the combination of application-specific requirements (dictated by the global criticality within the application) as well as inter-application demands (modeled by the commonality measures). Nodes that are not common across applications get mapped in a way such that feasibility is met, while still attempting to minimize the total area. In particular, we presented two methods to partition such application sets. In the first method, nodes that are repeated more often across different applications are biased towards hardware, so as to improve the utilization of the specialized hardware accelerators. In the second method, applications are considered in a specific order for partitioning, dictated by the relative criticality of applications. Mapping decisions made in one application are shared with other applications in an attempt to maintain consistency of mapping common nodes. When mapping a node type for the first time, its PA measure is used to select its mapping. In this way, the best possible mapping is selected for the node, considering the effect of the current application only. Based on the experiments carried out so far, this method appears to be superior to the first method (the resultant solution is 38% smaller). This is attributed to the fact that method B incorporates some of the design principles in method A and also takes the consistency into consideration.

7. References