Hardware/Software Partitioning for Multifunction Systems

Asawaree Kalavade and P. A. Subrahmanyam, *Fellow, IEEE*

***Abstract*—We are interested in optimizing the design of multifunction embedded systems such as multistandard audio/video codecs and multisystem phones. Such systems run a prespecified set of applications, and any "one" of the applications is selected at a run time, depending on system parameters. Our goal is to develop a methodology for the efficient design of such systems.**

A key observation underlying our method is that it may not be efficient to design for each application separately. This is attributed to two factors. First, considering each application in isolation can lead to application-specific decisions that do not necessarily lead to the best overall system solution. Second, these applications typically tend to have several commonalities among them, and considering applications independently may lead to inconsistent mappings of common tasks in different applications. Our approach is to optimize jointly across the set of applications while ensuring that each application itself meets its timing constraints.

Based on these guiding principles, we formulate, as a codesign problem, the design and synthesis of an efficient hardware-software implementation for a multifunction embedded system. The first step in our methodology is to identify nodes that represent similar functionality across different applications. Such "common" nodes are characterized by several metrics such as their repetitions, urgency, concurrency, and performance/area tradeoff. These metrics are quantified and used by a hardware/software partitioning tool to influence hardware/software mapping decisions. The idea behind this is to bias common tasks toward the same resource as far as possible while also considering preferences and timing constraints local to each application. Further, relative criticality of applications is also considered, and the mapping decisions in more critical applications are allowed to influence those in less critical applications. We demonstrate how this is achieved by modifying an existing partitioning algorithm (GCLP) used to partition single-function systems. Our modified algorithm considers global preferences across the application set as well as the preference of each individual application to generate an efficient overall solution while ensuring that timing constraints of each application are met. 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.

***Index Terms*— Hardware-software codesign, hardware/software partitioning, multifunction systems, system-level design, video encode/decode.**

Manuscript received January 8, 1998; revised February 12, 1998. This paper was recommended by Associate Editor G. Borriello.

A. Kalavade is with Bell Labs, Murray Hill, NJ 07974 USA.

P. A. Subrahmanyam is with the Computer System Laboratory, Stanford University, Stanford, CA 94305 USA.

Publisher Item Identifier S 0278-0070(98)06769-4.

I. Introduction

THE hardware-software codesign problem has received a lot of attention recently. Typical efforts in the hardware-software codesign for embedded systems assume that the system supports a single application. Thus, the goal is to find the best hardware-software implementation for a *particular* application, say, a video encoder or a graphics controller. However, there is a growing class of embedded systems that needs to execute a *set of applications* rather than just a single application. Such systems fall into two broad categories.

1) Systems that execute multiple applications concurrently, e.g., set-top boxes with concurrent applications like audio, video, and Web browsing.
2) Multifunction systems that support multiple functions or applications, of which only one is executed at any instant. Consider, for example, a multistandard video codec that supports MPEG2, H.261, and JPEG algorithms. Depending on whether the user is watching a movie or conducting a video conference, any one of these applications would run at a given time. Such multifunction systems offer alternatives between various functionalities—the specific alternative is typically selected at run time. Another example is a multisystem cellular phone that supports time division multiple access, code division multiple access (CDMA), and global system for mobility, only one of which is active at a given time, depending on the area of usage. A third example is a multiprotocol data-transmission system that handles different communication protocols such as Ethernet, v.34, etc.

In this paper, we focus on the design of systems that belong to the second category. We refer to such systems as *multifunction* systems. Note that we use the 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 multifunction embedded systems to have mixed hardware-software implementations. Our belief is that such systems can be designed efficiently if they are optimized across the set of applications. That is, the hardware/software partitioning decisions for each application can be made by considering the impact on and of other applications in the set, and not just the timing constraints of the application under consideration. 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 an individual application may lead to a decision to use a specialized (but not very reusable) hardware module, whereas investing the same or comparable area in a programmable processor core may yield a module with a much higher degree of reuse across different applications. Also, a joint consideration makes it possible to exploit the slack in one application and allow a "critical" application to be implemented more efficiently, thereby improving the overall solution. 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, their implementations could be inconsistent. Thus, a hardware implementation may be selected for the DCT in one application while a software implementation may be selected in the other application. For these reasons, we believe that, when designing for a multiapplication set, it is important to consider all the applications in the set simultaneously rather than design for individual applications. Our objective in this paper is to describe a methodology based on this belief and to demonstrate its payoff for concrete applications.

Toward this end, we formulate a codesign problem for the design and synthesis of an efficient hardware-software implementation for a multifunction embedded system. We assume that each application in the system has real-time constraints. The system-level design problem has two constituent subproblems: 1) hardware/software partitioning, which is the problem of mapping and scheduling each of the components in all of the applications in the given set, and 2) synthesis of the hardware, software, and interface components for all the applications in the given set. In this paper, we refer to this system-level design problem as the *multiapplication codesign problem*. While our paper focuses primarily on the partitioning problem, we also touch upon the synthesis problem.

Our approach to solving the partitioning problem is to modify traditional partitioning approaches so as to incorporate the unique features of multifunction systems. We begin by assuming that each application is specified by a directed acyclic graph (DAG) where the nodes represent computations of "coarse" granularity. This level of granularity could represent, say, a DCT on a block of pixels. Nodes that are common across applications are identified based on their functionality and parameters. The common nodes are characterized by several metrics (called commonality measures) like their repetitions, urgency, concurrency, performance/area tradeoff, etc. Applications are ordered according to their relative criticality. These metrics are used by the partitioning tool to make partitioning decisions jointly across different applications. We propose two specific algorithms for partitioning multifunction systems. These methods modify GCLP, an algorithm for partitioning a single application, developed by Kalavade *et al.* [1].

In the first method for multifunction partitioning, called hardware-oriented partitioning (HOP), the mapping of common nodes is biased toward a hardware implementation. The commonality measures are used to determine this bias. The intuition is that by biasing the common nodes toward hardware, the less common nodes may get mapped to software, thereby reducing the overall hardware area and reusing the hardware resource more efficiently across all applications. Note that the commonality measures are used to bias the mappings—not to *assign* the mappings. In other words, commonality measures indicate a preference for a mapping, based on the other applications in the set, but the final mapping decision is made by the partitioning tool. Specifically, the partitioning algorithm takes into account the demands of the application under consideration as well as the bias introduced by the other applications in the set, and attempts to generate a solution that minimizes hardware area while meeting timing constraints for that application. This use of bias is a subtle point. GCLP is a good vehicle to express bias, as we shall see in Section IV-B.

The second method for multifunction partitioning, called consistency and hardware oriented partitioning (CHOP) tries to incorporate the consistency requirement mentioned earlier. In this case, the key idea is to use application criticality to influence the order in which applications are considered for mapping, as well as to propagate mapping decisions across applications. In other words, if a node of the same type as the node being considered in the current application has been mapped earlier by some other application, the mapping decision for the node in the current application is biased in the *direction* of the previous mapping. When the node is considered for the first time, its performance/area measures are used to make the decision local to that application. The intuition here is to give preference to a more "critical" application, and the mapping decisions made for this application are allowed to influence mappings of other not-so-critical applications considered subsequently. For example, consider a low-criticality application with a large deadline. Suppose that a node i in this application can be implemented in two different feasible ways A and B, with A giving a slightly better solution for this application. Also say that a node of the same type as node i was mapped to implementation B by a more critical application. In this case, the mapping of node i can be set to B instead of developing two different implementations for node i. Thus, a suboptimal solution for one application may be selected in order to get a better overall solution. Such an approach is greedy. However, by ranking the applications in the order of their importance, the greediness is applied in a controlled manner. It is also possible for the user to go back and change mappings in applications considered earlier. Subsequent mappings can be recomputed by applying the partitioning tool. Since the partitioning algorithm is quite efficient (quadratic in the number of nodes), such an exploration is easily possible.

Note that the partitioning process computes not just the mapping but a schedule for the execution of each application as well. Once the partitioning process is finished, synthesis tools are used to generate the hardware and software components for each application. Our synthesis approach is based on cosynthesis tools developed by Kalavade *et al.* within the Ptolemy design environment [2]. There are still some issues that need to be solved in the context of synthesis, which we mention in Section V.

The rest of this paper is organized as follows. In Section II, we define the multiapplication codesign problem in more

detail and describe the assumed specification semantics and the architectural model. The overall methodology is described in Section II-C. In Section III, we discuss some of the related work in this area. In Section IV, we describe the two algorithms, HOP and CHOP. The hardware/software cosynthesis approach is described in Section V. In Section VI, we demonstrate the use of the proposed algorithms with the help of an example application set consisting of three video applications. We compare the total system area obtained when each application is considered independently to the total system area obtained by applying the two proposed algorithms. The resultant solution is shown to be 8% smaller with HOP and 38% smaller with CHOP.

II. Problem Definition and Codesign Methodology

A. Specification of Applications

We are interested in the codesign of multifunction systems, where, given a set $AP = \{A_1, A_2, \cdots, A_k\}$ of k applications, only one of these applications is active at a given time. The particular application running at a given time is selected at run time, either determined by the user by selecting a certain application (such as selecting between MPEG2 decode or H.261 in a video codec, depending on whether the user is watching a movie or conducting a video conference, respectively) or determined automatically by system parameters (such as automatically shifting from CDMA to Advanced Mobile Phone System (AMPS) in a multimode cellular phone when the area of use changes). To simplify the problem, we do not consider the design of the control code that selects the application at run time. We focus on the design of an implementation that supports the set of applications such that any one may be active at a given time and the active application meets its timing constraints.

We consider applications that have a periodic repetitive behavior with fixed timing constraints. One iteration through the application A_j is assumed to be specified as a DAG ($G = (N, E)$), where nodes N specify computations and edges E specify data and control precedences between nodes. Each edge e_{kl} is also assumed to be annotated by the number of data samples communicated between nodes k and l. Each application A_j has a timing constraint that specifies the maximum allowable time D_j for executing one iteration of the application. The DAG can be generated from data-flow descriptions, such as in the synchronous data-flow (SDF) domain of Ptolemy [2]. Note that an SDF specification supports iterations (where the number of iterations is known at compile time), delays, hierarchy, feedback, and multirate operations (multirate graphs are typically translated to a DAG by unfolding). The nodes of the DAG are at a "coarse" level of granularity; for instance, typical nodes might include a DCT, finite impulse response (FIR) filter, or quantizer. Although this model of specification limits the class of applications that can be described, we have found that a fairly large number of continuous media applications such as audio/video encode/decode can be described in this model. The benefit of this constrained model is that it permits a static (compile-time) analysis of the graph. A limitation of this approach is that control operations are assumed to be either encapsulated within nodes of the DAG or flattened out. Our approach is to try to optimize the data-flow parts of the designs across multiple applications. An area of future work is in the design of mixed control-data-flow systems.

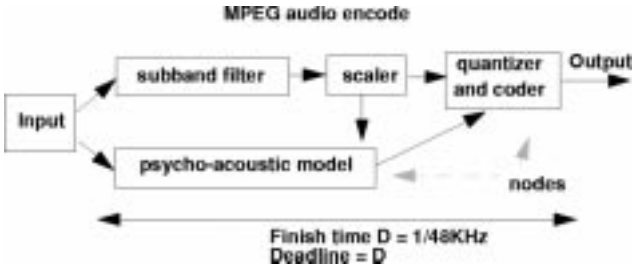


Fig. 1. Specifying a single application. Note the granularity of nodes and the deadline constraint.

Fig. 1 shows the specification of an application. Fig. 2 shows an end system that supports a set of multiple applications. Our objective is to optimize the design of an implementation that supports all of the applications, such that each application when selected to run meets its timing constraints. The design objective is to minimize the cost of the complete system supporting all the applications in the set.

B. Architecture Specification

Before we proceed with a discussion of the design methodology, we summarize the assumed system architecture. The embedded system is assumed to consist of three types of resources:

1) a single programmable processor core that executes the software component of the nodes mapped to software;
2) a set of *fixed-function hardware accelerators*, each of which is designed for a single function in hardware, e.g., motion estimation;
3) *macrofunction hardware modules*, each of which supports a closely related class of tasks differing in parameters or interfaces, e.g., a filter module with programmable taps that supports different FIR filters.

Another example of a macrofunction hardware module is a coprocessor optimized for vector operations that supports DCT and inverse (I) DCT.

In this paper, we assume that a fixed-function hardware accelerator can be synthesized for any node. We describe mechanisms to estimate the area and size of such accelerators in Section II-C and describe methods to synthesize an implementation for them in Section V. We also assume that the macrofunction hardware modules, on the other hand, are selected from a list of *available* modules. In Section IV-A1, we propose a method that may be used to design such a module, but we have not implemented this method.

The end system is also assumed to be constrained by the amount of memory available AS (hence restricting the software size possible) and the maximum hardware area possible AH (based on packing constraints, for example). Fig. 3(a) shows the architecture of the embedded system.

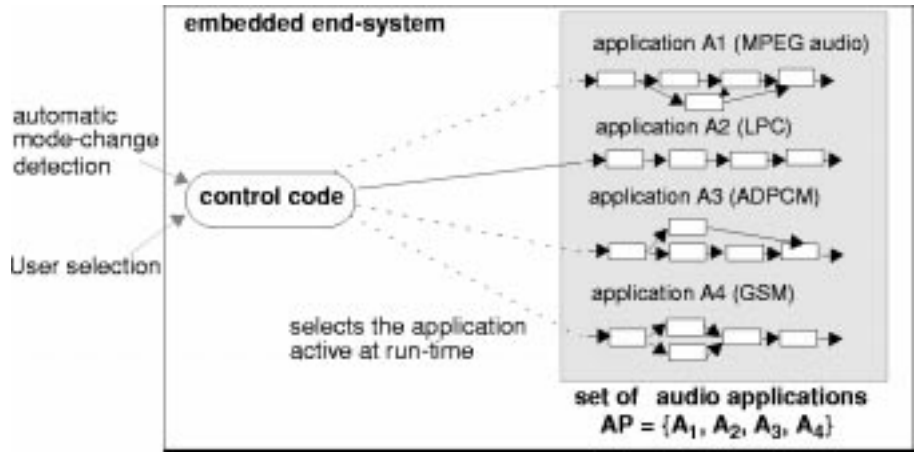
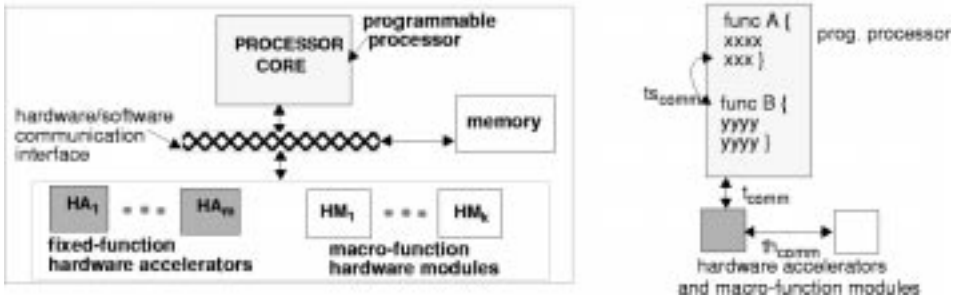


Fig. 2. An end system supporting a set of audio applications. The control code selects the application active at run time. We focus on the design of an implementation that supports all the applications in the set such that each meets its deadlines. We do not consider the design of the control code.



(a) (b)

Fig. 3. (a) Assumed architecture of the end system. (b) The hardware/software interface.

We assume that the communication between the different components can be abstracted by area and time parameters. Specifically, t_{comm} represents the time taken to transfer a sample of data across the hardware-software interface, i.e., t_{comm} is the time to transfer a sample of data between nodes A and B if nodes A and B are mapped to hardware and software, respectively. The communication time depends on the assumed communication model. For example, in a synchronous model, it includes interrupt handlers in addition to the actual data-transfer time. In the case of an asynchronous communication, it is the time to write to a buffer. Similarly, th_{comm} (ts_{comm}) is the time taken to transfer a sample of data when both nodes involved in the communication are mapped to hardware (software). Hardware area and software code size are also associated with each interface. Specifically, ah_{comm} is the hardware area required to implement the hardware end of the hardware-software interface, and as_{comm} is the code size associated with implementing the software end of the interface (code for polling or the specific interrupt handler). Fig. 3(b) illustrates some of the communication parameters. The GCLP discussion in Section IV-B describes the use of these various parameters and constraints within the partitioning process. Mechanisms to implement an asynchronous interface between the hardware and software have been described in [4]. In this paper, we consider a shared bus between the modules. Note that the hardware and software components can execute in parallel. Last, note that other communication mechanisms can also be incorporated within the partitioning tool.

C. Design Methodology

Fig. 4 shows our proposed design methodology for the codesign of multifunction systems. The methodology comprises four main steps: 1) specification of the application set, 2) estimation of node-level execution time and area in hardware and software implementations, 3) hardware/software partitioning, and 4) system synthesis.

1) Specification of the Application Set: The first step is to specify each application in the set. As described earlier, we assume a course-grain data-flow specification. The SDF domain of Ptolemy is used to specify each application. The output of this step is a DAG for each application in the set. An implicit assumption in such a block diagram specification is that each block corresponds directly to a node (i.e., the granularity of partitioning). The partitioning tool maps each node to either hardware or software; it does not break up user-specified blocks. This approach seems reasonable for the coarse-grain data-flow semantics assumed. Alternatively, the applications can also be specified as DAG's, independent of the Ptolemy interface.

2) Estimation of Node-Level Execution Time and Area in Hardware and Software Implementations: The next step is to get estimates of area and execution time for each node in

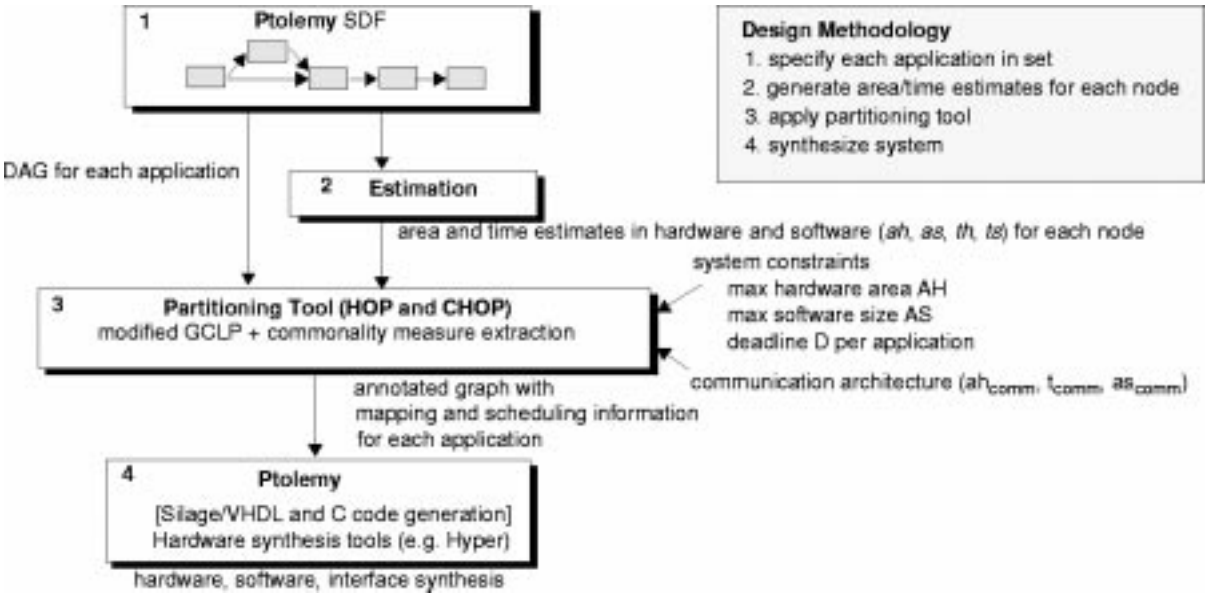


Fig. 4. Design methodology for multifunction system codesign.

each application. Recall that each node could be implemented in software on the programmable processor, in hardware as a fixed-function hardware accelerator, or in hardware mapped to a macrofunction module. Hence, estimates need to be obtained, for each node, for execution time in hardware (th), execution time in software (ts), area when implemented in hardware (ah), and code size when implemented in software (as).

As mentioned earlier, we make a simplifying assumption that the macrofunction modules are selected from a given library of modules. Note that such a macrofunction-module approach is quite reasonable when nodes differ slightly, i.e., in number of iterations, data sizes, word lengths, interfaces, etc. Although designing such a library of modules is nontrivial, this approach holds in the wake of the increasing trend seen in the design community toward design "reuse." For such nodes, the estimates of execution time of a node on such a macrofunction module and the area of such a module are assumed to be known from the library. As the library of such macrofunction modules matures, the estimates can be refined.

Next, we describe methods to obtain estimates of execution time in fixed-function hardware accelerators and software. Before we go into the details of the estimation process, we digress briefly to comment on the accuracy of the estimation process. The usefulness of the generated partition depends on how accurate the estimates are; if the actual values obtained after synthesis are very different from the estimates, the resultant solution could either be infeasible or have an unnecessarily large hardware area. Accuracy of the estimated area and time values is directly proportional to the time spent on generating the estimates; the more accurate the estimate, the longer it takes to derive it. In the limiting case, the best estimate is obtained after actually synthesizing the implementation. Our philosophy in generating the estimates is to get fairly accurate estimates from behavioral specifications, without going through the entire synthesis process. This approach is similar to the estimation methodology described in Appendix B of [4]. The key ideas are summarized here for completeness. Our approach assumes a block diagram description of each application, where each block corresponds to a node in the DAG. We assume that a C and Silage description is available for each node. In our current implementation, the generated Silage code for each node is fed to Hyper [5], a high-level synthesis tool. Hyper generates estimates of execution time (th) and area (ah) for a hardware implementation of the node. The hardware execution time is computed as the best case execution time, i.e., the fastest execution time. This corresponds to the critical path of the control-data-flow graph associated with the node.[1] The hardware area is computed by setting the sample period for the node to be the critical path. Note that we are not restricted to using Hyper for generating estimates. If VHDL descriptions were available for each node (a more reasonable assumption than Silage), synthesis tools such as Behavioral Compiler from Synopsys could be used to generate estimates of area and execution time. We have used the Hyper approach to demonstrate our ideas since we had free access to the tools, the source code, and the experts who designed the tools.

To obtain the estimates for software execution time and size, we first use the code-generation mechanisms of Ptolemy to generate C code [or code for a digital signal processor (DSP)] for each node. Next, this code is mapped to the particular processor under consideration. The implementation of the cosynthesis tools described in [4] assumes the 56 000 DSP from Motorola. Once assembly code is generated, the software execution time (ts) estimates are obtained by using the processor simulator, and code size (as) estimates are

[1] In a separate work, Kalavade *et al*. have addressed the problem of "extended" partitioning [1], where the area and execution time of a node are not single values but instead are represented as an area-delay tradeoff curve. Such a curve represents the spectrum of implementation options for a node within a given mapping. The extended partitioning problem is to select, in addition to a hardware or software mapping, the appropriate implementation for each node from such an area-delay curve.

obtained by running simple scripts on the generated code. For the examples we present in this paper, we used an in-house processor, and software size and execution time estimates were available from the application developers.

Our assumption of requiring Silage (or VHDL) and C code to be available for each node does pose some limitations on the usability of this approach. However, these restrictions seem less severe now (our cosynthesis framework was developed around 1993) given the current trend in commercial system-level system synthesis tools such as COSSAP (from Synopsys) and SPW (from the Alta group of Cadence) to also assume a library-based mapping approach. It has been our observation that the set of library modules becomes increasingly extensive over time, and it becomes more likely that such C and Silage/VHDL descriptions exist for each node.

Note also that our partitioning methodology is not restricted to a block-diagram-based approach. Other specification and cosynthesis mechanisms that start with a unified description of the application can also use our partitioning methodology, as long as a DAG where nodes are annotated with area and time estimates can be generated.

3) Hardware/Software Partitioning: Once the estimates of execution time and size in hardware and software have been obtained for each application, the partitioning tool maps and schedules nodes in each application. Details of the partitioning methodology are referred to Section IV. The output of the partitioning process is an annotated graph indicating the selected mapping for each node and a schedule for the execution of each application.

4) System Synthesis: Last, the annotated application graphs are fed back into the Ptolemy framework for synthesis of the hardware, software, and interface components. The synthesis mechanism is described in Section V. Of course, other synthesis mechanisms that start with such an annotated DAG could be used as well.

D. Problem Definition

The multiapplication codesign problem is formally defined as follows. Given a set $AP = \{A_1, A_2, \cdots A_k\}$ 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. Only 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.

Our partitioning tools try to minimize the area of the extra hardware (macrofunction modules and fixed-function hardware accelerators) while ensuring that each application meets timing constraints. The motivation for this is to use the available software processor as much as possible. The partitioning tool can be used in an iterative design methodology where different processors of varying cost and performance could be used. For each processor, the total system cost can be computed as a sum of the costs of the processor and the additional hardware. A design that minimizes the system cost can be selected. As mentioned earlier, the partitioning process is quite efficient and hence is amenable to such rapid exploration. A limitation of our current partitioning approach is that we are restricted to a single processor. Extending our approach to allow multiple programmable processors in the system is an area of future work.

Given the architecture template shown in Fig. 3 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.

III. Related Work

The hardware/software partitioning problem for a single application has been proved to be NP-hard [4]. The problem of optimizing over a set of multiple applications is at least as hard as the problem of obtaining a design for a single application. 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 related areas. We place these in perspective next.

The application-specific instruction processor (ASIP) synthesis 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 data path and controller for that instruction set is designed. Several bodies of research address this problem. For instance, the optimal instruction set selection problem is formulated as an integer linear program in the PEAS system [6], [7]. Van Praet *et al.* [8] present an interactive approach to selecting the microinstructions in the ASIP. Other approaches include [9]–[11]. 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 ASIP's. 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. Any progress in designing ASIP's can be used to complement the techniques we discuss here. We comment more on this topic in the concluding section.

Potkonjak *et al*. [12] address the problem of combining several concurrent tasks onto a single application-specific integrated circuit (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 bit-width 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.

The problem of designing systems where multiple applications run concurrently has been studied separately by Kalavade and Moghé [13].

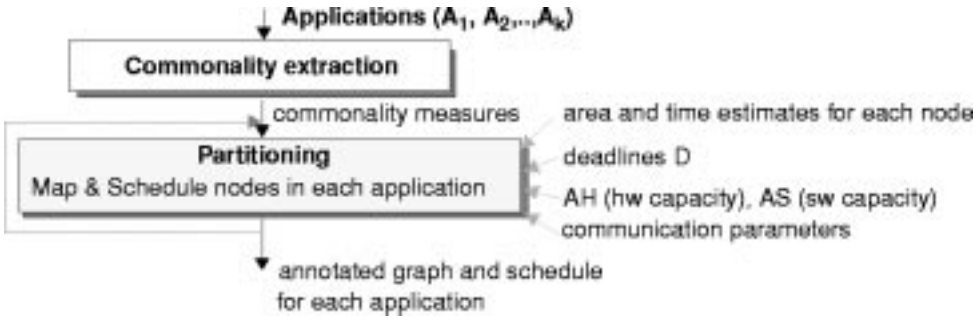


Fig. 5. Partitioning methodology for multifunction systems.

IV. Partitioning Methodology

The proposed methodology for the multiapplication partitioning problem is shown in Fig. 5. The set of specified applications is first analyzed to extract commonality measures across applications. In Section IV-A, we identify some measures, such as repetitions and performance-area tradeoff, which can be used to characterize nodes across all the 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 IV-C, we describe two methods (CHOP and HOP) to partition the nodes in the applications into hardware and software. These methods are based on a previously developed algorithm, GCLP [1], used to partition a single application. The GCLP algorithm is summarized in Section IV-B.

A. Commonality Measures

In this section, we first describe methods to identify "common" nodes, i.e., nodes that are common across different applications. Then we describe different commonality measures and methods to quantify them. These commonality measures are based on several nodal properties.

1) Identification of Common Nodes: The set of common nodes consists of nodes that repeat over different applications in the set and can be implemented on the same resource. As discussed earlier, we assume that nodes can be implemented in three ways: software, hardware on a fixed-function accelerator, and hardware on an existing macrofunction module.

In general, identifying structural and functional matches across different nodes is a difficult problem. In this work, we use a simple method as follows. The user tags all the nodes in all the applications with a tuple: ⟨*root name, parameters*⟩. If the root name and parameter values on two nodes match exactly, the nodes are common and can in fact be mapped to the same resource: either a programmable processor executing the same function or the same fixed-function hardware accelerator. Such a fixed-function hardware accelerator can be synthesized using the Ptolemy mechanisms described in Section V.

If the parameters do not match, we check for an existing macrofunction hardware module that matches with the root name and can support both parameters. Each macrofunction hardware module is assumed to be characterized by a root name and a list of possible parameters. Two nodes that can be mapped to the same macrofunction hardware module are also said to be common and are tagged with the particular module that can support them. An example of a partial match is the two nodes ⟨DCT, one block⟩ and ⟨DCT, eight blocks⟩, where DCT is the function name and the parameters specify the number of blocks operated on for each iteration. Note that a similar approach can also be used to incorporate macrofunction software modules.

Once matching is done, each node is tagged with a *node type*; nodes that match have the same node type. The set of common nodes is the set of nodes that have the same node type.

This root-name-parameter matching approach that we currently use for detecting commonality is quite simplistic. Sophisticated techniques such as template matching can be used to detect matches. We propose a mechanism to do this: a functional description (such as C or Silage or VHDL code) is assumed to be available for each node. Starting with such a high-level description, a control-data-flow graph (CDFG) is generated for each node using techniques commonly used in the high-level synthesis or compiler communities. A list of possible templates characterizing different architecture features (such as FIR filters, infinite impulse response filters, multiply-accumulates, etc.) is maintained. The CDFG associated with each node is then analyzed to detect which templates are present. This can be done using covering techniques such as those proposed by Rao *et al.* [14]. Then the templates associated with nodes can be compared. Nodes that have the same templates are said to be common. A macrofunction hardware module that implements such a set of templates for common nodes can be synthesized. An alternative to the template approach is to compare pairs of nodes to extract regular patterns across them, as proposed by Guerra *et al.* [15]. Common implementations that accommodate regular patterns can be synthesized. Note that this process can be made even more complex by trying to look across node boundaries to detect matching patterns. In the limiting case, such methods to extract commonality tend to the ASIP problem that tries to identify common "instruction patterns."

We have taken a first step in this direction; for future work, we intend to formalize this approach. As a starting point, we believe that restricting to simpler approaches might increase the practicability of our approach. In practice, often the hard-

ware/software selection is limited by available macrofunction hardware modules. In such a scenario, allowing the user to detect possible function matches might suffice. Note that even if the user determines the common nodes, the actual selection of the mapping for each application is not obvious due to the other constraints in the problem. The partitioning tool determines the best mapping while meeting the constraints.

One of our underlying philosophies in automated partitioning is to try to *systematize* the design process commonly used by designers today. We have strived to make the core partitioning tool efficient to permit an iterative and interactive design methodology.

2) Quantification of Commonality Measures: Once the set of common nodes is identified, each common node is analyzed to compute metrics that characterize its properties. We have identified some properties and propose simple techniques for quantifying them. In Section IV-C, we describe the use of these metrics in the partitioning tool. The properties, their quantification, and their intuitive meaning are discussed next.

a) Repetitions of a node (R): The repetitions (R_i) of a node i are computed as the number of occurrences, across all the applications, of nodes of the same type as the type of node i. R_i is simply the number of times node i appears over all the applications. 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, allocating a custom hardware area for it can be justified 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.

b) Performance-area ratio of a node (PA): The performance-area ratio (PA_i) of a node i is measured as the ratio of the performance gain (speedup achieved by implementing 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 selecting a hardware implementation.

c) Urgency of a node (U): 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 critical path in the application graph is the path that has the longest execution time.) 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.

d) Concurrency of a node (C): The concurrency C_i of a node i is the number of "concurrent" instances of the node. One way to compute the concurrency is as follows. Each application is first scheduled assuming infinite resources for each node type. The number of instances actually used at any cycle is computed for each node type for each application. The concurrency of node i is defined as the average of these numbers over all applications. The concurrency of a node can influence the number of instances of the node when it is implemented in hardware. For example, if the concurrency for node i is two, it indicates that, on 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 $\langle R_i, PA_i, U_i, C_i \rangle$ is thus computed for each node. Nodes with the same type have the same commonality vectors. For each property, the values are then normalized across all the nodes. The cumulative effect of the different properties can be incorporated by combining different measures. In Section IV-C, we show how some of these measures are used in the partitioning process.

B. Algorithm for Partitioning Single Independent Applications

In this section, we briefly summarize the GCLP algorithm proposed by Kalavade *et al.*, which is used to partition a single application *independent* of other applications. For more details on the GCLP algorithm, the reader is referred to [1]. GCLP is the kernel of the multiapplication partitioning procedures. In Section IV-C1 and IV-C2, we describe two modifications to the GCLP algorithm that can be applied to the multiapplication codesign problem.

The GCLP algorithm assumes an architecture consisting of a programmable processor and custom hardware accelerators. The application is assumed to be specified as a DAG, similar to that described in Section II-A of this paper. The goal of the GCLP algorithm is to find the mapping and schedule for all the nodes in the given application such that the deadline is met and the area of the nodes mapped to hardware is minimized.

The GCLP algorithm is based on list scheduling, where the graph is traversed from a source node to the sink node and one node is mapped in each step. In contrast to traditional list scheduling, where a single objective function is used to select the mapping of the node, GCLP *adaptively* selects the mapping criterion from among two possible objective functions: minimize finish time of the node or minimize the area of the node. This criterion could change at each step in the algorithm. The motivation for adaptively selecting the optimization objective is as follows. Due to the constrained nature of the mapping problem, minimizing hardware area and meeting timing constraints often present conflicting optimization goals. An objective function that minimizes finish time drives the solution toward feasibility but is likely to be suboptimal in terms of the area. On the other hand, if a node is always mapped such that area is minimized, the final solution may not meet timing constraints. To overcome this problem, the GCLP algorithm adaptively selects one of the optimization objectives at each step, depending on which of the two dimensions is critical at that step. This criticality is computed via a global criticality measure GC. That said, using just the GC measure may lead to locally suboptimal mapping decisions. Sometimes a node can have attributes that render its mapping to either hardware or software more appropriate. These are accommodated by quantifying the local preference of nodes (δ) and using this value to bias the threshold used in GC comparison. Fig. 6(a) summarizes this idea. Let us now look at the details of the GCLP algorithm.

The flow of the GCLP algorithm is shown in Fig. 6(b). 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. During initialization, for each node i, a local attribute δ_i that quantifies the preference of i to a hardware or a

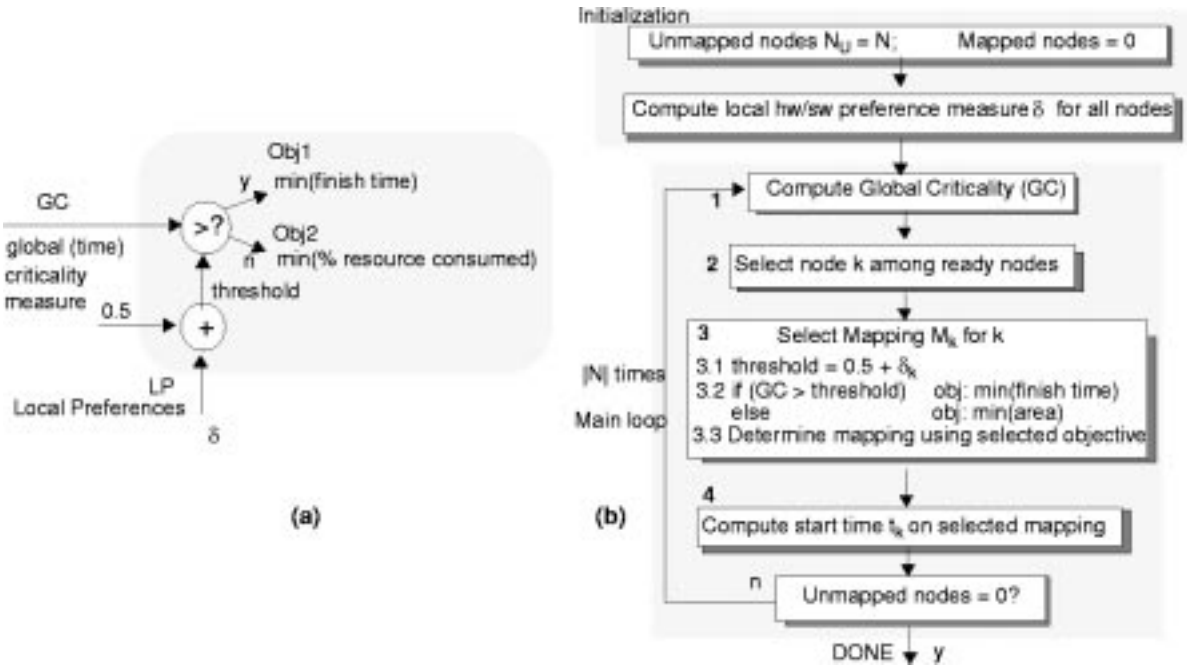


Fig. 6. (a) Key idea of the GCLP algorithm. (b) Flow of GCLP algorithm.

software mapping is computed. These attributes include the area/performance ratio in different mappings, properties such as bit manipulations and precision variance that indicate an affinity to hardware, and factors such as the density of control operations and memory operations that dictate an affinity to software. The attribute δ_i for a node i is computed as a convex combination of these affinity metrics for node i. In this paper, we will not go into more details of these attributes.

The algorithm then maps one node per step. At the beginning of each step, the global time criticality measure GC is computed for that step. 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. GC is computed at each step of the algorithm as a fraction of the as yet unmapped nodes that need to be moved from software to hardware, given the mappings of nodes mapped so far, such that the resultant solution is feasible. A high GC indicates that time is more critical. After computing GC, a node k is picked from among all *ready* nodes (nodes whose predecessors have been mapped) using an urgency criterion. In particular, the node k on the longest path is selected in this step, as shown in step S2) of the algorithm shown at the bottom of the next page. The next step, S3), is to select the mapping for the selected node k. As described earlier, instead of using a fixed function to select the mapping, the GCLP algorithm *adaptively* selects between minimizing total hardware area and minimizing system finish time as the objective to be used to determine the mapping of node k. This adaptive selection is governed by global criticality GC and local preference δ_k of the selected node k, as shown in Fig. 6. The GCLP algorithm selects the optimization objective by comparing GC to a threshold. If GC is greater than the threshold, time is said to be critical, and a mapping that minimizes the finish time of the node under consideration is selected. If GC is less than a threshold, a mapping that minimizes the resources (hardware area) consumed is selected. The threshold may be either supplied by the user or statistically determined over a test set. In our experiments, we assume a threshold of 0.5. Note that the GC measure is also a "look-ahead" measure that tries to minimize the greediness typically associated with serial traversal. As mentioned earlier, the local preference of the node is quantified by δ and is used to bias the threshold used in GC comparison. (This local preference is replaced by the bias introduced by the commonality measures in the case of multiapplication partitioning.)

Using the selected optimization objective, the selected node k is assigned a mapping (M_k). The start time t_k for the execution of node k is then computed using the finish times of all of its predecessors and the communication delay (depending on the relative mappings of predecessors and k). This set of start times defines the schedule for the system. The process is repeated $|N|$ times over all nodes. The algorithm is summarized on the next page.

Some of the key steps in the algorithm outlined are discussed next. After computing GC, a node k is selected for mapping from among all ready nodes in step S2). Ready nodes are those whose predecessors have been mapped. We use an urgency criterion to select node k, i.e., k is selected as the node, from among all ready nodes, that has the longest path to completion. Execution time values are required in order to compute the longest path. The effective execution time of nodes that have already been mapped is determined by their mapping. However, the execution time is not known for nodes that have yet to be mapped. To address this problem, we define the effective execution time $(t_{\text{exec}}(i))$ of an unmapped node i as the mean execution time of the node, assuming it is mapped to hardware with probability GC and to software

with probability $(1 - GC)$. Here we use the notion of GC as a node-invariant probability that an unmapped node will be mapped to hardware. Once the effective execution times are computed, the longest path is computed and node k is selected according to the longest path.

The mapping for node k is selected in step S3). First, the effective threshold is computed by adding the local preference to a default threshold of 0.5. GC is compared to this threshold. If GC is greater than the threshold, time is assumed to be critical, and hence a mapping that minimizes the finish time of node k is selected (objective 1). If GC is less than the threshold, time is not that critical, and a mapping that minimizes the system hardware area is selected (objective 2). Based on the mapping selected, the schedule is computed in step S4). Note that this schedule also includes the communication times. The objective functions are as follows.

Objective 1: $t_{\text{finish}}(k,m)$, where $m \in$ {software, hardware} ($t_{\text{finish}}(k,m)$ is the finish time of node k on mapping m)

$$t_{\text{finish}}(k,m) = \max(\max_{P(k)}(t_{\text{finish}}(p) + t_c(p,k)), t_{\text{last}}(m)) + t(k,m), \text{ where}$$

$P(k)$ = set of predecessors of node $k, p \in P(k)$

$t_{\text{finish}}(p)$ = finish time of predecessor p

$t_c(p,k)$ = communication time between predecessor p and node k

$t_{\text{last}}(m)$ = finish time of the last node assigned to mapping m

= 0 if m corresponds to hardware

$t(k,m)$ = execution time of node k on mapping m.

Objective 2: $((as_k + as_{\text{comm}}^{\text{tot}})/AS) \cdot I(m = sw) + ((ah_k + ah_{\text{comm}}^{\text{tot}})/AH_{\text{remaining}}) \cdot I(m = hw)$, where $I(m = x)$ is an indicator function whose value is one if mapping m is "x."

Objective 1 selects a mapping that minimizes the finish time of the node. A node can begin execution only after all of its predecessors have finished execution and the data has been transferred to it from its predecessors. Note that the communication delays are taken into account; $t_c(p,k)$ includes the time to transfer data to this node and is dependent on the particular communication mechanism and the number of samples transferred. Also, a node cannot begin execution on the software resource until the last node mapped to software has finished execution. This is accounted for by the term t_{last}. Thus, t_{finish} is computed for node k on all mappings m. The mapping that minimizes $t_{\text{finish}}(k,m)$ is selected as the mapping for node k.

Objective 2 uses a "percentage resource consumption" measure. This measure is the ratio of the resource area of a node (nodal area plus communication area) to the total resource area. Recall that the area of node k in hardware and software is ah and as, respectively. The area $ah_{\text{comm}}^{\text{tot}}$ ($as_{\text{comm}}^{\text{tot}}$) takes into account the total cost of communication (glue logic in hardware and code in software) between node k in hardware (software) and all its predecessors. For the hardware resource, the resource area required by the node is divided by the available hardware area ($AH_{\text{remaining}}$), while for the software resource, the resource area is divided by the software capacity AS. Objective 2 thus favors software allocation as the algorithm proceeds. Note that this step also checks to see if the resource required for this mapping still meets the capacity constraint (AS or AH). If objective 2 is selected as

---

Algorithm GCLP
Input: Graph $G = (N, A)$; For each node i in N: ah_i, as_i, th_i, ts_i and a timing constraint D
Output: for each node i in N, Mapping M_i, start time t_i;
Initialize:
I1) Compute values for all nodes.
I2) $N_U = \{\text{unmapped nodes}\} = N$.
Procedure:
while $\{|N_U| > 0\}$ {
S1) Compute GC
S2) Select node k for mapping
  S2.1) Determine N_R, the set of ready nodes
  S2.2) Compute the effective execution time $t_{\text{exec}}(i)$ for each node i
    if (mapping$(i) == 0$) $t_{\text{exec}}(i) = GC * th_i + (1 - GC) * ts_i$ /* unmapped */
    else if (mapping$(i) ==$ hardware) $t_{\text{exec}}(i) = th_i$
    else if (mapping$(i) ==$ software) $t_{\text{exec}}(i) = ts_i$
  S2.3) Compute the longest path longestPath(i), using $t_{\text{exec}}(i)$, for all ready nodes
  S2.4) Select node k with the maximum longestPath for mapping
S3) Determine mapping M_k for k:
  S3.1) threshold $= 0.5 + \delta_k$
  S3.2) If ($GC >$ threshold) m: minimize(objective1); /* minimize(finish time)*/
    else m: minimize(objective2); /* minimize(hardware area)*/
  S3.3) Determine mapping M_k using selected objective.
S4) Compute start time t_k
S5) $N_U = N_U - \{k\}$; Update remaining time;}

the mapping function, then the percentage resource consumed is computed for all mappings. The mapping that minimizes this quantity is selected.

Note that different communication models can be accommodated within GCLP. In this paper, we assume a shared bus. Exclusive access to the bus by different resources is accounted for by modifying the function that computes the communication cost. Specifically, the bus is checked for availability while computing the communication and finish times. This is similar in concept to modeling sequential execution on the programmable processor. For the implementation described in [4], a point to point communication between hardware and software was assumed.

Last, the GCLP algorithm has quadratic complexity in the number of nodes. In the work reported in [1], it is shown that the solution generated by GCLP compares favorably with the optimal solution generated by an exact formulation using integer programming.

C. Partitioning Multiple Applications

We next describe two methods for partitioning multiple applications. These methods use modified versions of the GCLP algorithm. In the first method, HOP, described in Section IV-C1, the commonality measures are used to bias the mapping decisions made by GCLP. In particular, if the repetitions measure (R) of a node is high, the node is biased toward a hardware mapping; otherwise, the performance-area measure (PA) is used to bias its mapping toward one that is most suitable to that node. In the second method, CHOP, described in Section IV-C2, the applications are ordered according to their relative criticality and selected for partitioning in this order. Mapping decisions made when partitioning one application are propagated to the next application. Common nodes are encouraged to share the same mapping, thereby maintaining consistency of mapping across applications.

1) HOP—Hardware Bias for Common Nodes: 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 (and hence has a 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, relative to all nodes in all applications, when implemented in hardware, and if the difference in software and hardware execution times is high (i.e., node has a high PA measure), it might be beneficial to bias the mapping of the node toward hardware. This will 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 tradeoff measure, and ρ and π are user-defined weights. CM_i is normalized over all the nodes in the application. The CM_i measure is used to bias the threshold in the GCLP algorithm. The procedure adopted in HOP for designing for multiple applications is summarized as follows:

Procedure HOP
S1) 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.
S2) Compute the performance-area trade-off measure for each node i as $PA_i = (ts_i - th_i)/ah_i$ where for node i, ts_i and th_i are execution times in software and hardware respectively and ah_i is the hardware area. Normalize PA_i over all nodes to a value between 0 and 1.
S3) Compute for each node i, the commonality measure $CM_i = \rho R_i + \pi PA_i$ using user-specified weights ρ and π. Normalize CM_i to a value between 0 and 0.5.
S4) For each application A_a, run GCLP − HOP.

Algorithm GCLP-HOP
This is a variant of the GCLP algorithm described in Fig. 6, where step 3 in GCLP is replaced as follows:
GCLP-S3) Determine mapping M_k for k:
GCLP-S3.1 threshold $= 0.5 - CM_k$.
GCLP-S3.2 if$(GC >$ threshold$)$
$M_k =$ hardware,
else $M_k =$ software;

The interpretation of step S4) in procedure HOP is as follows. For nodes with a nonzero CM_k 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 (as dictated by GC) as well as interapplication demands (modeled by CM_k). 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.

2) CHOP—Consistency in Mapping Common Nodes: In HOP, we used GCLP-HOP to bias common nodes toward hardware. However, an alternative way of mapping multifunction systems is to maintain a "consistency" in the way the common nodes are mapped. We introduce a second variant of the GCLP algorithm that embodies this particular design principle. The key idea is to map the applications in a particular order and to propagate information about the mapping decisions made. One way to select the order of applications is to use an application criticality measure,

which indicates the relative computational complexity of each application. The propagation of mapping decisions allows information about mapping decisions within an application to be *shared* between applications. Let k be the node being considered. If a node of the same type as k has been mapped earlier, the mapping decision for k is biased in the direction of the previous mapping. Otherwise, the mapping of node k is determined locally. The procedure used in CHOP is summarized at the bottom of the page.

The interpretation of step S4) in procedure CHOP is as follows. Let k be the node being considered at a particular step. If a node of the same type as node k has been mapped by a previously considered application, the goal is to try to preserve that mapping for node k. 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_k favoring a hardware implementation for this node. Similarly, if a node of the same type as node k has been previously mapped to software, the threshold is raised by the repetitions measure R_k, thus favoring a software implementation. Note that the mapping of node i is not *hard coded* 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 mapped to hardware are increased. However, the state of the current application, as reflected by the time criticality measure GC, is also considered. GC is compared to the modified threshold, and if it is greater than the threshold, only then is a hardware mapping selected. Thus, the mapping of common nodes is biased toward a mapping that maintains consistency across all applications while making sure that the application-specific constraints are also taken into consideration. If no node of the type of node k has been mapped before, the PA measure is used to select its mapping. In this case, the measure tries to select the best possible mapping for node k, considering the effect of the current application only. In particular, if PA is high, it indicates a high benefit of a hardware mapping. In such a case, the threshold is lowered to favor a hardware mapping. Of course, if GC is low enough, then a software mapping gets selected.

While selecting the applications, the most critical application (highest AC) is considered first. The nodes in this application are mapped 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.

Once all the nodes in each application have been mapped to either hardware or software, using either of the methodologies described in the previous section, a postmapping optimization step may be applied. This involves swapping some nodes from hardware to software or applying user-controlled fine tuning.

V. System Synthesis

Once the mapping and schedule have been determined, the final system is to be synthesized. Fig. 7 shows the system-synthesis mechanism.

The hardware synthesis process assumes a hierarchical approach, where a layout is first generated for each node mapped to a fixed-function hardware accelerator, and these layout modules, along with the preexisting macrofunction modules,

---

Procedure CHOP
S1) Compute commonality measures R and PA for all nodes, as described earlier.
S2) Compute application criticality (AC)for all the applications in the set. AC_j is computed as $(\Sigma\, ts_i)/D_j$, where ts_i is the execution time of node i when implemented in software, $\Sigma\, ts_i$ is the sum of ts_i over all nodes i in applications j, 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.
S3) Order the applications by AC such that the most critical application (largest AC) is considered first.
S4) For each A_a in this order, run GCLP-CHOP

```
Algorithm GCLP-CHOP
This is a variant of the GCLP algorithm described in Fig. 6, where step 3
in GCLP is replaced as follows.
GCLP-S3) Determine mapping M_k for k:
        GCLP-S3.1 if (computed_shared_mapping(k) ==1)
                  {/* node type mapped before*/
                       if (shared_mapping(k) == hardware)
                           threshold = 0.5 - R_k /* lower threshold */
                       else if (shared_mapping(k) == software)
                           threshold = 0.5 + R_k /* raise threshold */
                  }    else/* no earlier shared mapping */
                           threshold = 0.5 - PA_k.
        GCLP-S3.2 if(GC > threshold) M_k = hardware, else M_k = software
        GCLP-S3.3) Update shared mapping
```

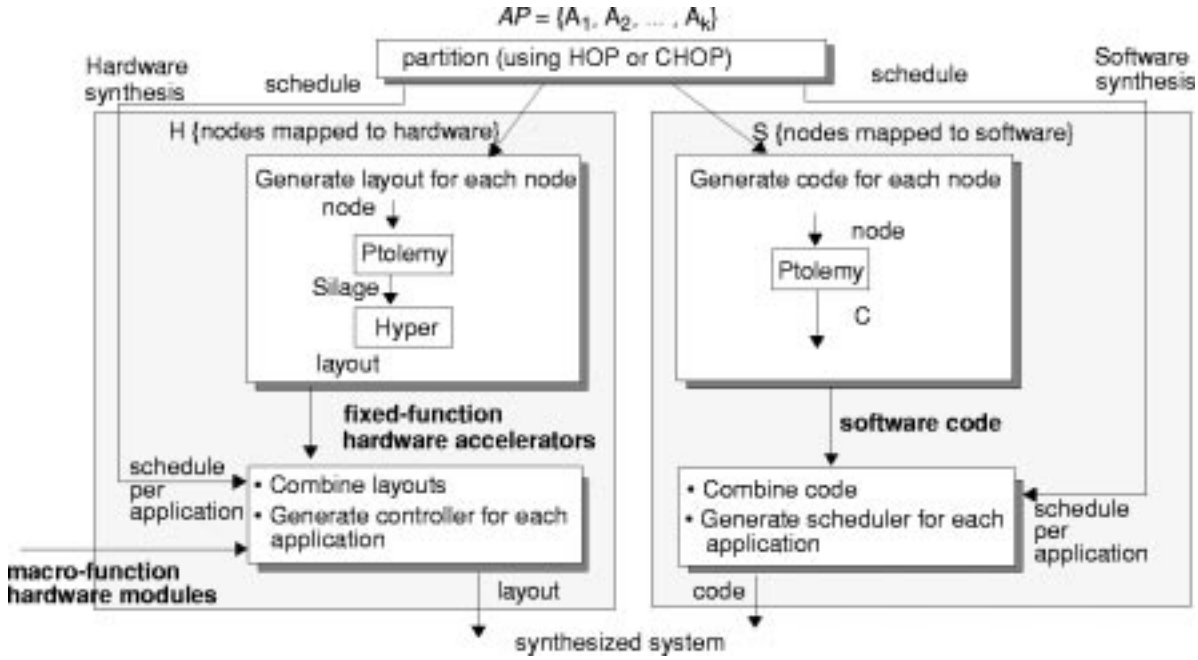


Fig. 7. System-synthesis procedure.

are put together to generate a netlist for the complete system. As mentioned earlier, we do not synthesize macrofunction modules but assume that these are selected from a preexisting set of modules. However, the method outlined in Section IV-A1 can be used to synthesize the macrofunction modules.

The layout for a fixed-function implementation is obtained by using a combination of preexisting tools such as Ptolemy [2] and Hyper [5]. In particular, Silage[2] code is first generated for the node using Ptolemy. This Silage code is then passed through Hyper to synthesize the actual layout. The layouts are then combined, and a controller is generated for each application using the generated schedule.

The software-synthesis process involves generating code for each application. A two-tiered approach, similar to the hardware-synthesis process, is assumed. C code is generated for each node using the predefined modules in Ptolemy. These code modules are then stitched together in the sequence dictated by the schedule (generated by the partitioning tool) to obtain a single code file for each application.

At run time, the software modules corresponding to the selected application are loaded in, and the hardware for the selected application is activated by the controller for that application. This synthesis mechanism has been implemented as part of the Ptolemy environment. More details on this approach can be found in [3] and [4].

VI. 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 VI-A, we describe the relevant details of the applications, architecture, and node metrics. In Section VI-B, we present the solution obtained when each application is considered *independently*. The solutions obtained by using HOP and CHOP are discussed in Sections VI-C and VI-D, respectively. The solutions obtained by using these are found to be superior to (have smaller area than) the solution obtained when each application is considered independently. We have implemented HOP and CHOP and incorporated them into the basic GCLP algorithm. Both HOP and CHOP are efficient algorithms and have a quadratic complexity in the number of nodes. Each run through the partitioning algorithm takes less than one second of CPU time on a Sparc20.

A. Application Set, System Architecture, and Node Metrics

1) Application Set: To illustrate some of the concepts introduced here, we consider the application set consisting of:

- MPEG2 video encode (M2E);
- H.261 decode and encode (H);
- 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). Suppose that we want to design an implementation that runs this set of applications, such that any one application runs at a given time. Fig. 8 shows the graphs for the three applications. One iteration through a graph corresponds to the processing of a macroblock of data[3] in that application. The latency constraint on the graph specifies

[2] Silage is a high-level functional language specifically designed to describe DSP applications. Several research and commercial synthesis systems use Silage as the specification language [for instance, Hyper from the University of California, Berkeley, and DSPStation from Frontier Design, Inc. (previously part of Mentor Graphics)]. The methodology is not restricted to using Silage and Hyper; VHDL and other synthesis tools can be used as well.

[3] A macroblock is a basic data type used in video encoding/decoding. An image is typically divided into blocks of pixels, where a block has 8×8 pixels, and a macroblock consists of four blocks. The number of macroblocks per image depends on the frame size.

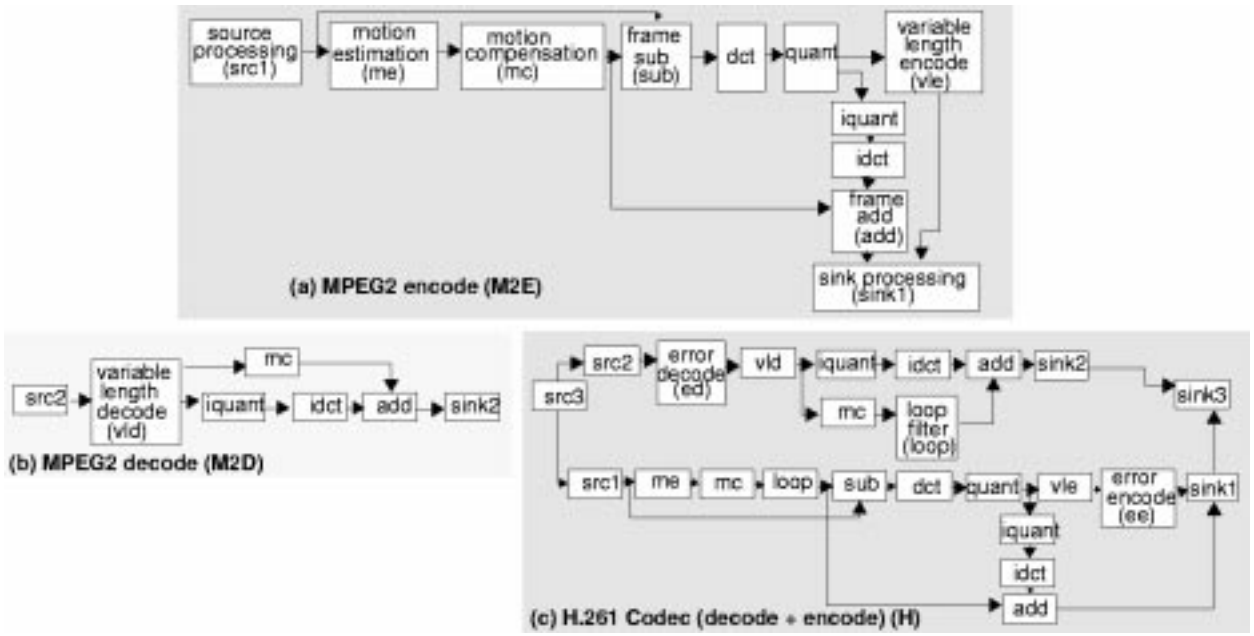


Fig. 8. Application graphs (a) MPEG2 video encode, (b) MPEG2 video decode, (c) H.261 video encode/decode.

TABLE I
APPLICATION SET

Application	frame size	frame rate	latency constraint	# nodes	criticality (AC)
MPEG2 video encode (**M2E**)	CCIR 601	12 fps	6110 cycles	11	3.27
MPEG2 video decode (**M2D**)	CCIR 601	15 fps	8888 cycles	7	1.23
H.261 video encode/decode (**H**)	CIF	15 fps	16666 cycles	24	1.68

the maximum time available to process one iteration. This is computed from the frame size and rate. Table I summarizes the details of the applications in the application mix, their timing constraints, and the application criticality. Note that the MPEG2 encoder and decoder operate at a larger frame size than the H.261 codec. The application criticality for application j is computed as the ratio $\sum ts_i/D_j$, where $\sum ts_i$ is the sum of the software execution times for all nodes i in application j and D_j is the deadline for application j.

2) System Architecture: We assume a system architecture consisting of a single programmable processor, multiple fixed-function accelerators, and coprocessors attached to the processor. The programmable processor can implement all the nodes (the software implementation). The coprocessor represents a special case of the macrofunction hardware modules described earlier. The coprocessor can implement some of the nodes. We assume a single-instruction multiple-data vector coprocessor that can implement the following functions: DCT/IDCT, Quantizer/IQuantizer, 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 or a variable length encoder. Table II summarizes the different resource types available. Fig. 9 summarizes the system architecture. The end system contains one programmable processor and several instances of the coprocessor and hardware accelerators. The hardware accelerators are connected to the processor via a shared bus. The coprocessor is closely coupled to the processor. The communication between different resources is abstracted into communication delays, which are used by the partitioning algorithm.

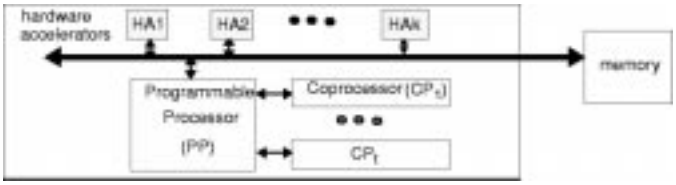


Fig. 9. System architecture assumed in experiments.

Different system implementations are quantified by a "system cost," which is the total hardware area required in addition to the programmable processor. The partitioning algorithm determines the number of coprocessors and the number and types of hardware accelerators, as well as a mapping of nodes in all the applications to these resources (programmable processor, coprocessors, and hardware accelerators) such that each application meets its timing constraint and the overall system cost is minimal.

3) Node Metrics: Software execution times for nodes are measured by a detailed simulation on a media processor

TABLE II
RESOURCE TYPES IN THE SYSTEM ARCHITECTURE. HA: FIXED-FUNCTION HARDWARE ACCELERATOR; CP: MACROFUNCTION HARDWARE MODULE

Type of Resource	Notation	Type of Resource	Notation
Programmable Processor	PP	Error Decoder (Hardware Accelerator)	HA6
SIMD Coprocessor	CP	Error Encoder (Hardware Accelerator)	HA7
Motion Estimator (Hardware Accelerator)	HA1	Encode Source Processing (Hardware Accelerator)	HA8
Variable Length Encoder (Hardware Accelerator)	HA2	Encode Sink Processing (Hardware Accelerator)	HA9
Variable Length Decoder (Hardware Accelerator)	HA3	Decode Source Processing (Hardware Accelerator)	HA10
H.261 Source Processing (Hardware Accelerator)	HA4	Decode Sink Processing (Hardware Accelerator)	HA11
H.261 Sink Processing (Hardware Accelerator)	HA5		

TABLE III
NODE TYPES, R AND PA MEASURES, AND IMPLEMENTATION OPTIONS

Node Type	R_i	PA_i	Implementation options for node	Node Type	R_i	PA_i	Implementation options for node
src1	0.3333	0	PP, HA8	sink1	0.3333	0.130435	PP, HA9
me	0.3333	1.0	PP, HA1	src2	0.3333	0.130455	PP, HA10
mc	1.0	0	PP, CP	vld	0.3333	0.135326	PP, HA3
sub	0.3333	0	PP, CP	sink2	0.3333	0.130435	PP, HA11
dct	0.3333	0	PP, CP	src3	0	0	PP, HA4
quant	0.3333	0	PP, CP	sink3	0	0	PP, HA5
vle	0.3333	0.338315	PP, HA2	loop	0.3333	0	PP, CP
iquant	1.0	0	PP, CP	cd	0	0.202899	PP, HA6
idct	1.0	0	PP, CP	cc	0	0.202899	PP, HA7
add	1.0	0	PP, CP				

developed in our lab. Software code size for each node is also available. Hardware execution times and areas for nodes (for both, fixed-function accelerators and the coprocessor) are abstracted from an actual implementation of a video-conferencing system developed in our lab. Note that several nodes are repeated over different applications (e.g., inverse quantizer appears four times; the motion estimator appears two times). Table III summarizes the repetitions (R) and performance/area (PA) measures for the different types of nodes in the application mix. It also lists the implementation options for each node type; the particular implementation for each node within different applications is selected by the partitioning process.

With this background, we are now ready to report the solutions obtained by different methods. In Section VI-B, the solution obtained by considering each application independently, without considering the other applications in the set, is reported (i.e., using vanilla GCLP). Results with HOP and CHOP are reported in Sections VI-C and VI-D, respectively. It is assumed that each of these solutions uses a single programmable processor. The results are tabulated as follows. For each application in the set, the hardware resources used in addition to the processor are reported. The corresponding system area for each application is the sum of the areas of the hardware resources. Note also that each solution represents a feasible solution for each application. The finish time (and schedule) calculations take into account the time to transfer data across the interface, in addition to the execution times in different implementations. Specifically, since a shared bus is assumed, exclusive access to the bus by different resources is accounted for in the calculations. Last, the row labeled "Set" reports the union of the hardware resources needed for the application set and the total system area for the set of applications.

B. Experiment 1: Independent Mapping

We first run each application independently through the original GCLP algorithm. This represents a naive approach, where each application is designed separately without taking into consideration any of the features of the multiple applications in the set. Table IV summarizes the resultant hardware resources used for each application and the total system area. Note that application H requires two instances of the resource

TABLE IV
RESULTS OF EXPERIMENT 1

Application	Hardware Resources Used	System Cost
M2E	CP, HA1, HA2, HA8	103
M2D	HA3, HA10	25
H	CP, CP, HA1, HA2, HA4, HA5, HA8, HA11	199
Set	**CP, CP, HA1, HA2, HA3, HA4, HA5, HA8, HA10, HA11**	**224**

TABLE V
RESULTS OF EXPERIMENT 2

Case	Application	Hardware Resources Used	System Cost
	M2E	CP, HA1, HA2, HA8	103
	M2D	HA3, HA10	25
	H	CP, CP, HA1, HA3, HA4, HA8	193
Case 2A:CM = R	Set	CP, CP, HA1, HA2, HA3, HA4, HA8, HA10	206
	M2E	CP, HA1, HA2, HA8	103
	M2D	HA3, HA10	25
	H	CP, CP, HA1, HA2, HA4, HA5, HA8, HA11	199
Case 2B: CM = PA	Set	CP, CP, HA1, HA2, HA3, HA4, HA5, HA8, HA10, HA11	224
	M2E	CP, HA1, HA2, HA8	103
	M2D	HA3, HA10	25
	H	CP, CP, HA1, HA2, HA4, HA8	181
Case 2C: CM=R+PA	Set	CP, CP, HA1, HA2, HA3, HA4, HA8, HA10	206

CP. The total area for the mix is computed by summing the areas of the union of the hardware resources used in the three applications. This represents the total system area required to implement the multifunction system.

C. Experiment 2: HOP

We next apply HOP, which uses GCLP-HOP, where the threshold is modified with a linear combination of the R and PA measures (recall that *threshold* $= 0.5 - CM_i$, where $CM_i = \rho R_i + \pi PA_i$). We present three sets of results with (ρ, π) set to (1, 0), (0, 1), and (1, 1), respectively. Table V summarizes the resultant hardware resources used for each application and the total system area for the three cases.

1) Observations: This method uses the principle of moving nodes with high repetitions to hardware. In case 2A, it was found that nodes idct, mc, sub, and dct were mapped to hardware (since they had higher repetitions) as against software in experiment 1. This was achieved by changing the threshold values when mapping these nodes. As a consequence, nodes with lower repetitions such as vld and sink3 got mapped to software. This resulted in a lowered system area. In case 2B, the PA measures do not appear to help in improving the solution over case 1, since the PA measures have the same effect as δ in considering local preferences of nodes. In case 2C, the combined effect of PA and R is the same as the effect of R alone in case 2A.Thus, the repetitions measure seems to help to reduce the system area. For this example, the solution obtained from HOP is 8% smaller than that obtained by considering applications independently.

D. Experiment 3: CHOP

We now apply CHOP, where the applications are considered in a specific order and mapping state is maintained between applications. The idea behind this algorithm is to maintain consistency when mapping nodes of the same type. The applications are considered in the order of decreasing application criticality (i.e., sequence M2E-H-M2D). Table VI summarizes the resultant hardware resources used for each application and the total system area.

1) Observations: CHOP 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. The results obtained so far indicate that CHOP is superior to HOP. This may be attributed to the fact that CHOP incorporates some of the design principles in HOP and also takes the consistency into

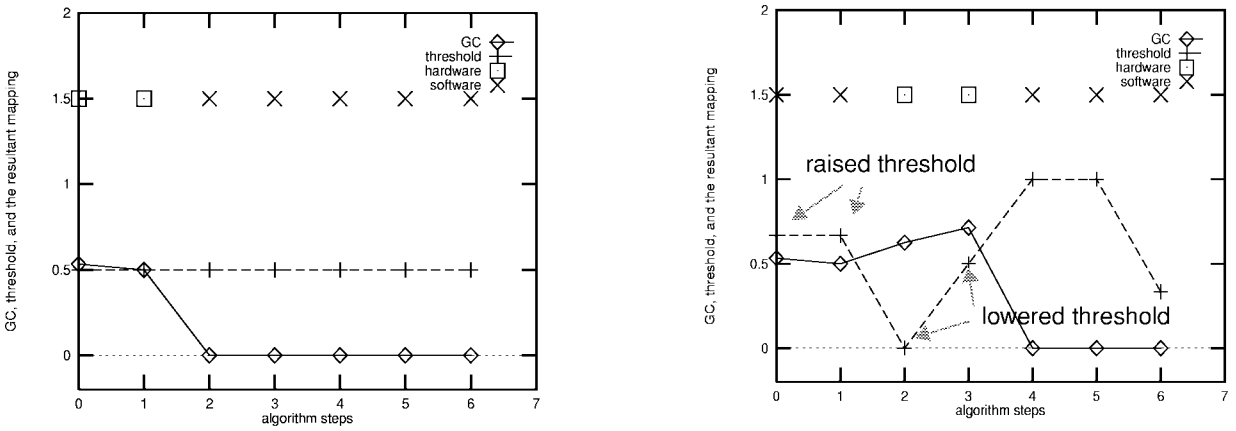


Fig. 10. Algorithm trace for MPEG2 decode (M2D). (a) M2D in experiment 1. (b) M2D in experiment 3.

TABLE VI
RESULTS OF EXPERIMENT 3

Application	Hardware Resources Used	System Cost
M2E	CP, HA1, HA2, HA8	103
H	CP, HA1, HA2, HA4, HA5, HA8, HA9, HA11	139
M2D	CP	65
Set	CP, HA1, HA2, HA4, HA5, HA8, HA9, HA11	139

consideration. We also found that considering the applications in the order of decreasing criticality gives the best solution. This is attributed to the effect that by giving preference to the more critical application first, a better overall solution can be obtained.

E. Algorithm Trace

To give a better insight into the working of the algorithm, a trace of the algorithm flow is shown in Fig. 10 for the M2D application. The graph plots the algorithm step on the X axis and the GC and threshold on the Y axis. The mappings of nodes are as shown. Fig. 10(a) shows the flow when the application is mapped independently, as in experiment 1. Fig. 10(b) shows the flow when the application is mapped using CHOP. In this case, this application is considered last.

In Fig. 10(a), at the first two steps the GC is higher than the threshold and a hardware mapping is selected. In Fig. 10(b), the threshold is raised in the first two steps, biasing the node toward software. This happens because these two nodes have been mapped to software by the applications considered previously. After the first two nodes get mapped to software, the next two nodes (iquant, idct) get biased toward hardware. This shows how the changed threshold helps to change the mappings. Also observe that the solution for M2D by itself is worse than in experiment 1. Specifically, when considered independently, special hardware accelerators were used for this application. Now, due to the bias, the coprocessor is used. However, the advantage is a reduced overall system cost since the same coprocessor is used in all applications.

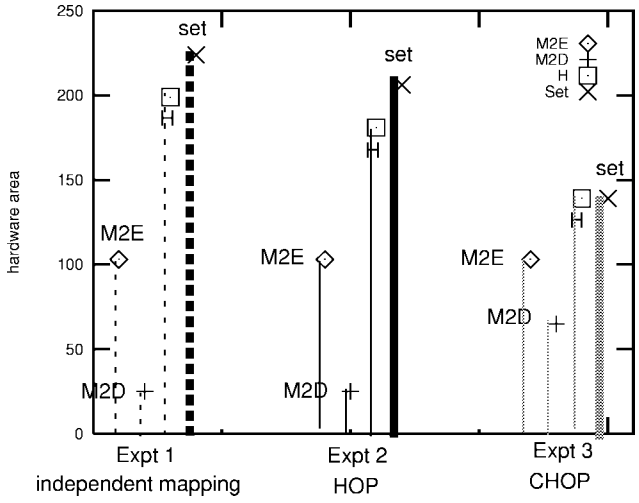


Fig. 11. Comparing hardware area for each application and for the complete set for the three experiments.

F. Comparing the Three Experiments

Fig. 11 summarizes the results obtained from the three experiments. For each experiment, the hardware area for each application and total system area for the set of applications is plotted. The CHOP method seems to give the best solution. Note that accepting a suboptimal solution for M2D in experiment 3 leads to an overall better system solution.

When the solutions from experiments 1 and 3 were compared, we found that there was not much consistency when mapping repeated nodes. In experiment 1, for example, of the four instances of the node "iquant," two were mapped to hardware and two to software. Overall, eight of the repeated nodes had mapping inconsistencies. In contrast, by using CHOP in experiment 3, only one repeated node had different implementations in different applications. This happened because the local and timing constraints had an overriding effect.

VII. 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 prespecified set of applica-

tions. 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 design, we have intentionally formulated the problem in a manner that avoids automatic processor design. Instead, we assume that the processor core and some macrofunction hardware modules are 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.

We presented two methods to solve the problem. The key idea is to analyze the entire set of applications to extract commonalities across different nodes in different applications. We identified several measures that characterize the nodes and defined ways to quantify them. These measures are used 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 (as dictated by the global criticality within the application) as well as interapplication 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 (HOP), nodes that are repeated more often across different applications are biased toward hardware so as to improve the utilization of the specialized hardware accelerators. For the example considered, it was found that this method reduces the overall system area by 8% when compared to the solution obtained when applications are considered independently. In the second method (CHOP), 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 the other applications in an attempt to maintain consistency in mapping common nodes. When mapping a node type for the first time, its PA measure is used to select its mapping. In this way, when the node is being considered for the first time, 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 CHOP incorporates some of the design principles in HOP and also takes the consistency into consideration.

ACKNOWLEDGMENT

The authors gratefully acknowledge P. Moghe for several valuable discussions on the algorithmic issues as well as for his feedback on this paper. They also thank the anonymous referees for insightful feedback on this paper.

REFERENCES

[1] A. Kalavade and E. A. Lee, "The extended partitioning problem: Hardware/software mapping, scheduling, and implementation-bin selection," *J. Design Automat. Embedded Syst.*, vol. 2, no. 2, pp. 226–163, Mar. 1997. [Online]. Available WWW: http://www.bell-labs.com/user/kalavade/papers/pdf/daem-partitioning.pdf.

[2] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *Int. J. Comput. Simulation*, vol. 4, pp. 155–182, Apr. 1994.

[3] A. Kalavade and E. A. Lee, "A hardware/software codesign methodology for DSP applications," *IEEE Design Test Comput. Mag.*, pp. 16–28, Sept. 1993.

[4] A. Kalavade, "System-level codesign of mixed hardware-software systems," Ph.D. dissertation, University of California, Berkeley, CA, Sept. 1995.

[5] J. M. Rabaey, C. Chu, P. Hoang, and M. Potkonjak, "Fast prototyping of datapath-intensive architectures," *IEEE Design Test Comput. Mag.*, pp. 40–51, June 1991.

[6] J. Sato, A. Y. Alomary, Y. Honma, and T. Nakata, "PEAS-I: A hardware/software codesign system for ASIP development," *IEICE Trans. Fundamentals Electron., Commun. Comput. Sci.*, vol. E77-A, no. 3, pp. 483–91, Mar. 1994.

[7] N. Ngoc, M. Imai, A. Shiomi, and N. Hikichi, "A hardware/software partitioning algorithm for designing pipelined ASIP's with least gate counts," in *Proc. 33rd DAC*, June 1996, pp. 527–532.

[8] J. Van Praet, G. Goossens, D. Lanneer, and D. H De Man, "Instruction set definition and instruction selection for ASIP's," in *Proc. 7th Int. Symp. High-Level Synthesis*, Niagara-on-the-Lake, Canada, 1994, pp. 11–16.

[9] H. Ing-Jer and A. M. Despain, "Generating instruction sets and microarchitectures from applications," in *Proc. ICCAD 94*, pp. 391–396.

[10] P. Paulin, C. Liem, T. May, and S. Sutarwala, "DSP design tool requirements for embedded systems: A telecommunications industrial perspective," *J. VLSI Signal Process.*, vol. 9, no. 1/2, pp. 23–47, Jan. 1995.

[11] W. Zhao and C. A. Papachristou, "Synthesis of reusable DSP cores based on multiple behaviors," in *Proc. ICCAD 96*, pp. 103–109.

[12] M. Potkonjak and W. Wolf, "Cost optimization in ASIC implementation of periodic hard-real time systems using behavioral synthesis techniques," in *Proc. ICCAD 95*, pp. 446–451.

[13] A. Kalavade and P. Moghe, "A tool for performance estimation of networked embedded systems," in *Proc. DAC*, June 1998, pp. 257–262.

[14] D. Rao and F. Kurdahi, "Partitioning by regularity extraction," in *Proc. 29th DAC*, 1992, pp. 235–238.

[15] L. Guerra, M. Potkonjak, and J. Rabaey, "System-level design guidance using algorithm properties," *VLSI Signal Processing VII*, J. Rabaey, P. M. Chau, and J. Eldon, Eds. New York: IEEE Press, 1994.

Asawaree Kalavade received the B.E. degree in electronics and telecommunications engineering from the University of Poona, India, in 1989 and the M.S. and Ph.D. degrees in electrical engineering from the University of California (UC), Berkeley, in 1991 and 1995, respectively.

She is a Member of Technical Staff in the Networked Multimedia Systems Research Department, Bell Labs, Murray Hill, NJ. Her current research interests include performance estimation and rapid prototyping tools for the system-level design of networked embedded systems. She previously was with the DSP and VLSI Systems Research Department, Bell Labs, where she led a team of researchers working on the design and implementation of tools for the design of software for a single-chip multiprocessor DSP. Her contributions include defining the software architecture of the system, the design and implementation of a multiprocessor real-time kernel, and the design of a static scheduling framework. She also has developed a fast analytical performance-estimation framework called AsaP. AsaP has been used for the design of networked embedded systems as well as for quantifying the impact of different run-time scheduling policies on multimedia end systems. Her doctoral research focused on several aspects of hardware/software codesign (including partitioning, cosynthesis, and cosimulation). She was part of the Ptolemy project group at Berkeley. She is the author of numerous papers and has consulted for Berkeley Design Technology, Inc.

Dr. Kalavade received the Best Student Award for academic excellence (College of Engineering, Poona) and the Dora Garibaldi Fellowship (UC Berkeley).

P. A. Subrahmanyam (M'84–SM'96–F'97) is a Consulting Professor in the Computer System Laboratory at Stanford University, Stanford, CA. He was with Bell Laboratories Research, most recently involved with the design of the software/hardware architecture of multiprocessor-based systems-on-a-chip for multimedia and wireless applications. His current work relates to 1) the design of, and the design methodology and tools for, a new generation of embedded/networked information appliances and systems on a chip and 2) ways to leverage communication (intranet/internet) frameworks in the design and deployment of these applicances. His research interests span various aspects of computer-aided design, software/hardware architecture, formal methods, hardware-software codesign, and embedded system design. He has authored/coauthored/edited more than four books on formal methods of very-large-scale-integration design and multimedia systems and has authored/coauthored more than 100 journal and conference papers. He has presented several colloquia, invited technical talks, and tutorials at universities, research laboratories, and conferences worldwide and has chaired/served on various technical/conference/National Science Foundation program committees.

Dr. Subrahmanyam has received both the Outstanding Paper Award and Outstanding Presentation Award at ICCD.