An Architectural Co-Synthesis Algorithm for Distributed, Embedded Computing Systems

Wayne H. Wolf, Senior Member, IEEE

Abstract—Many embedded computers are distributed systems, composed of several heterogeneous processors and communication links of varying speeds and topologies. This paper describes a new, heuristic algorithm which simultaneously synthesizes the hardware and software architectures of a distributed system to meet a performance goal and minimize cost. The hardware architecture of the synthesized system consists of a network of processors of multiple types and arbitrary communication topology; the software architecture consists of an allocation of processes to processors and a schedule for the processes. Most previous work in co-synthesis targets an architectural template, whereas this algorithm can synthesize a distributed system of arbitrary topology. The algorithm works from a technology database which describes the available processors, communication links, I/O devices, and implementations of processes on processors. Previous work had proposed solving this problem by integer linear programming (ILP); our algorithm is much faster than ILP and produces higher-quality results.

Index Terms—Co-synthesis, embedded computing systems, hardware/software co-design.

I. INTRODUCTION

T HIS paper describes a new algorithm for the architectural co-synthesis of embedded hardware and software architectures. The algorithm synthesizes a distributed multiprocessor architecture and allocates software processes to the CPU’s in the multiprocessor such that the combined hardware-software architecture is of minimal cost to meet hard deadlines. The synthesized multiprocessor may have an arbitrary topology, as determined by the design requirements; the synthesis algorithm selects appropriate CPU’s and communication channels from a technology-specific library.

Previous work in VLSI CAD, distributed systems, and real-time systems have addressed various aspects of this problem and will provide a valuable foundation to this work. However, most previous work has concentrated on solving for only one or two variables at a time: allocating processes for a given partitioning, scheduling processes for a given hardware configuration, etc. In contrast, our goal is to study multiple-parameter system design: changing the partitioning, scheduling, and allocation of software processes while simultaneously optimizing the underlying hardware which is executing the software.

The algorithm described in this paper constructs a cost-minimal distributed processing system which meets all specified performance constraints and simultaneously allocates and schedules the software processes onto the processing elements (PE’s) in the distributed system. Once the architecture has been designed, other synthesis algorithms or manual design can be used to complete the lower levels of abstraction of the design. A co-synthesis algorithm is useful, beyond its obvious use for architecture design, as a product planning tool. Embedded computing systems are frequently designed as families, with successive generations developed using the previous generation as a starting point. Designers can use a co-synthesis tool to determine the cost and performance of implementations of an incompletely specified proposed product. For example, the co-synthesis tool can help determine whether projected memory costs will allow a particular feature to be added to the next-generation system. Co-synthesis allows the design space to be much more thoroughly explored than is possible than by manual methods.

The next section describes the characteristics of embedded computing systems and the role architectural co-synthesis plays in the design process. Section III describes previous research on problems related to architectural co-synthesis. Section IV describes our formulation of the architectural co-synthesis problem in detail. Section V describes the co-synthesis algorithm and Section VI describes the results of experiments with the algorithms.

II. EMBEDDED COMPUTING SYSTEM DESIGN

The ability to co-synthesize a distributed multiprocessor with a custom interconnect architecture is important to the embedded system designer because many embedded systems are distributed systems. Examples include: a pen-based computer which used four microprocessors, each of a different type [25]; a high-performance signal processing machines, such as radar processors, which are implemented using multiple 32-bit processors along with some custom components; a robot arm controller designed by Srivastava et al. [31] which was implemented by microprocessors connected to a work station bus; 35 mm cameras which use two or more microprocessors; and automobiles such as the Mercedes S-class, which uses over 60 microprocessors; most navigation and communications systems for boats and aircraft communicate via RS-232 serial connections. Distributed embedded computers exhibit a wide range of architectures: loosely coupled systems which communicate via RS-232; microprocessor networks which use higher-speed serial communication systems such as the I2C bus.
An application may be implemented as a distributed system for any of several reasons. In many cases, using several 8-bit microcontrollers is often cheaper than using one 32-bit embedded microprocessor. Even if a more expensive microprocessor is required for some computation-intensive tasks, smaller microcontrollers may be used as device handlers to off-load the main CPU. If the design includes several I/O devices, enough on-board I/O devices or timers may be required that more than one embedded controller is necessary using several microcontrollers which include on-board I/O logic may be the lowest-cost design. Finally, there are cases in which even the largest single CPU cannot handle the computation load and processes must be put on separate CPU’s and/or ASIC’s to meet all the hard deadlines.

An embedded computing system consists of a hardware engine which executes application software [35]. If we limit our design requirements to cost and performance for the sake of argument, the architectural design problem for the hardware engine is to determine how much hardware is needed to make sure that the system meets its deadlines and its soft performance goals, while minimizing the engine’s cost. The computing elements of the hardware engine are PE’s, which may be general-purpose CPU’s, digital signal processors (DSP’s), floating-point units, ASIC’s, etc. In the simplest case of a single-CPU engine, the PE must be fast enough to run all the processes fast enough to meet all their deadlines under the worst-case input combinations. Even in a single-CPU design, the designer is faced with several choices within a CPU family (different bus widths, different clock speeds, different amounts of execution parallelism) as well as choices across families (different architectures and manufacturers). When designing a distributed hardware engine, the designer faces the PE selection choice at each node as well as the choice of communication channels between the PE’s.

However, the hardware engine cannot be designed without simultaneously considering the application software architecture. The allocation of processes to PE’s and the scheduling of those processes and their communications determines the cost of both the PE’s and communications network. Therefore, the software architecture must be optimized simultaneously with the hardware architecture to ensure that arbitrary choices in the software design have not foreclosed attractive hardware design options. Our co-synthesis algorithm allocates processes to PE’s while simultaneously allocating PE’s and communication channels in the hardware engine and it schedules the processes to determine the execution demands of the processes on the CPU’s and communication system.

The input and output of architectural co-synthesis are shown in Fig. 1. An embedded system’s specification includes both functional and nonfunctional elements [10]; a model for the functional specification is a process graph; the nonfunctional specification of interest in this work is a rate constraint, or the maximum time from initiation to termination of the process graph execution. (The inverse of the process graph’s rate is its period.) The product of architectural co-synthesis is a pair of architectures: the hardware engine architecture consists of the component PE’s and their interconnections; the software architecture consists of an allocation of processes to the PE’s, priorities for process execution, and assignment of interprocess communication to physical communication links. Detailed design of the hardware and software components is guided by the structure of these architectures. The goal of architectural co-synthesis is to estimate and predict the results of detailed design decisions well enough to guide the construction of an effective, efficient architecture. Co-design is necessary at several different levels of the design hierarchy. For example, Chou et al. [5] developed an algorithm which synthesizes a complete device interface using a combination of instructions executing on the host CPU and external interface logic. Architectural co-synthesis works at the top level of the design hierarchy; it does not produce a complete implementation of the hardware and software components of the system, but it does a combined hardware-software architecture for the complete system.

The design of a hardware–software (HW/SW) architecture for an embedded system requires consideration of a great deal of information. Each software process executes at different speeds on different PE’s. PE types also vary in their available communication topology and bandwidth and their on-chip devices. Architectural design requires considering the utilization of PE’s, interprocess communication, component cost, and other factors. While simple systems can be designed on the back of an envelope, sophisticated systems which execute more complex functions and which have stricter performance and cost constraints require tools which can efficiently search through architectural choices to find an efficient design.

A useful architectural co-synthesis algorithm must be able to handle incomplete functional specifications. There are many reasons why a designer may not be able to completely specify the function to be executed on the hardware engine: while some software may be lifted from previous designs, new
III. PREVIOUS WORK

Related previous work includes studies of architectural partitioning, hardware-software partitioning, hardware-software co-synthesis, and distributed system scheduling and allocation.

Architectural partitioning algorithms model the design as a marked graph and partition the graph into smaller subgraphs to optimize performance and interconnect cost. Partitioning algorithms rely primarily on the structure of the graph during optimization. APARTY [19] is an architectural partitioner targeted mainly to hardware designs thanks to its emphasis on a large number of relatively small operators; it uses a hierarchical clustering algorithm. PARTIF [18] is an interactive partitioning tool based on the SOLAR design representation of communicating processes.

Hardware–software partitioning algorithms implement a system from an architectural template: a CPU and a custom ASIC communicating over a bus, as illustrated in Fig. 2. These algorithms either move operations from hardware to software to minimize cost, as does the algorithm of Gupta and De Micheli [17], or moves operations from software to hardware to satisfy performance goals, as does the algorithm of Ernst et al. [11] Vahid et al. [34] use a multi-phase partitioning algorithm to improve the results of hardware–software partitioning.

SIERA [31] is a template-based synthesizer of board-level controllers; the system description is successively refined by having one generator build a system from components created by smaller generators. At each stage of abstraction, two templates—one for hardware and another for software—describe the basic structure of the unit to be synthesized. Generators are used to create detailed designs of the components, which may themselves invoke generators at the next-lower level of abstraction. Properties of the specification are used as parameters to the generators. SIERA does not feed back results of hardware or software synthesis to modify the architecture generated by the template. SIERA has been used to design a robot arm controller, among other projects.

Some recent work in co-synthesis has developed techniques for synthesizing engines with arbitrary topologies. In the SOS algorithm [23], Prakash and Parker formulated the hardware engine design problem as a software process set implemented on a hardware architecture and solved the resulting problem using mixed integer linear programming (MILP) techniques. D’Ambrosio and Hu [8] simulated a set of processes to determine the feasibility of a given hardware-software architecture on a single-CPU system. Barros et al. [2] use a clustering algorithm to synthesize an engine and allocate processes to the engine.

Research on the scheduling and allocation of processes in distributed systems is relevant to our research. Distributed systems scheduling algorithms assume that the processes have already been partitioned and that the hardware architecture of the distributed system is given. Leinbaugh and Yamani [20] developed algorithms to bound the amount of time required to execute a set of processes on a distributed system. Ramanritham et al. [24] developed a heuristic scheduling algorithm for real-time multiprocessors. Sih and Lee developed a multiprocessor scheduling algorithm called declustering [30] which reclusters processes on CPU’s to trade off concurrency for interprocess communication time.

Distributed systems allocation algorithms assume that the processes have been allocated and the topology of the distributed computing engine is given. Stone [33] developed the first algorithm for allocation of processes to processors on distributed systems. Dasarathy and Feridun [9] developed extensions for real-time constraints. Chu et al. [7] developed heuristics for taking interprocess communication times into account during the allocation process. Chu and Tan [6] developed heuristics to include precedence relations between processes during optimization; they used process size as an approximation for important precedence relationships. Shen and Tsai [27] used a graph matching heuristic to allocate processes; their algorithm minimized interprocessor communication and balanced system load. Gopinath and Gupta [15] applied a combination of static and dynamic techniques to improve processor utilization. They statically analyze process code and assign two predictability/unpredictability and monotonicity/nonmonotonicity values to each process, which they use to move less-predictable code earlier in the schedule. They then use software monitors to keep track of actual execution time of processes and adjust the schedule on-line.

Our co-synthesis algorithm has several advantages over previous work. Unlike distributed system optimization algorithms, we do not assume that the topology of the hardware engine is given. Unlike template-driven schemes or hardware–software partitioning algorithms, our algorithm can synthesize an engine with an arbitrary topology. It also selects the proper PE type at each node. We will see that the difficulty of predicting performance of a process before the PE it will run on has been selected is the greatest single problem in the design of our co-synthesis algorithm. Unlike architectural partitioning algorithms, it evaluates system performance based on non-graph-theoretic properties of the design; we schedule and
allocate processes to evaluate the hardware engine architecture. D’Ambrosio and Hu consider only architectures with a single CPU. Barros et al. take advantage of properties of the Unity language to simplify analysis of the system specification.

The work of Prakash and Parker is closest to our own, since they co-synthesize a distributed engine of arbitrary topology. However, our algorithm is heuristic and can therefore produce a solution very quickly. We believe that a heuristic algorithm is an important step in the development of architectural co-synthesis algorithms because the development of heuristics both offers the promise of much shorter execution times and helps identify the key relationships between subproblems and further refine the solution method. The experimental results of Section VI demonstrate that our heuristic algorithm gives results comparable to those generated by ILP on small problems and can handle larger problems due to its low execution times. Our algorithm also allocates devices along with processes. When PE’s include on-chip devices, it is important to allocate those with processes: a process and its device must be on the same chip, and the number of on-chip devices required may expand the number of PE’s of a given type available, allowing different allocations of processes to PE’s.

IV. PROBLEM FORMULATION

As illustrated in Fig. 3, the co-synthesis process takes as input the system specification along with a set of technology parameters; it produces both the hardware architecture, consisting of PE’s and communication channels connecting them, and the software architecture, which gives the allocation of processes to PE’s, the scheduling of processes, and the scheduling and allocation of communication on channels. The formulation of our problem can be divided into three components: the problem specification; the definition of the hardware and software architectures; and the technology-dependent descriptions of the hardware and software components.

A. Problem Specification

The problem specification (which Prakash and Parker called the task model) includes both the functional and nonfunctional requirements, as shown in Fig. 4. A process is the atomic unit of specification. A process may begin execution when all its inputs arrive and emits its outputs when execution terminates. The execution time for a process depends on the type of PE to which it is allocated, as described in Section IV-C. A process graph defines the system’s functional requirements: is a directed acyclic graph whose nodes represent processes and whose edges (data dependencies) represent data communication between processes and devices. The system of processes described by the process graph is repeatedly executed. Each edge is labeled with the size of the datum transmitted in units of bits; data is transmitted at each termination of the process. In the example, $s_{13}$ specifies the number of bits transmitted from process $P_1$ to $P_3$ in a single communication. An input device is represented by a node with no inward edges and an output device by a node with no outgoing edges. Prakash and Parker model processes which can start executing before all of their inputs have been received using additional variables to model the execution times at various stages of processing; we prefer to represent such processes as a network of processes, with some processes receiving initial inputs and passing partial computations onto later processes. Similarly, we can model system inputs which arrive after execution has begun by adding pseudo-processes which delay time-zero-arrival inputs by the appropriate amount.

The minimum rate at which the process graph must be executed is a nonfunctional requirement of the system. The rate is measured from the process graph inputs to the computation of the last output of the process graph. The inverse of the rate is the maximum period of one process graph execution. The rate is a hard constraint—it must be satisfied for the implementation to be feasible. We refer to schedule of processes which ensures that all processes complete with the period as a feasible schedule.

B. Architecture Model

The architecture of the implementation consists of the software architecture and hardware engine architecture. The hardware engine architecture is built from three kinds of primitives: PE’s, I/O devices, and communication channels. The hardware architecture is specified as a labeled hypergraph: nodes represent PE’s or devices, with labels giving the type of component represented; communication channels are specified by hyperedges (since a communication channel may connect more than two devices, an edge may connect more than two nodes).

The software architecture implemented by co-synthesis consists of processes and interprocess communication (IPC) channels (also known as communication channels). Each software
process in the implementation corresponds to a process in the specification. Interprocess communication links are implemented over communication channels (capped with software interfaces) for processes executing on different PE’s; IPC links are implemented by shared memory communication mediated by the operating system when the communicating processes execute on the same PE. The software architecture may be described in terms of the problem specification: the process allocation is given by a mapping from the nodes in the process graph to processors in the hardware engine architecture; the communication allocation is given by a mapping from edges in the process graph to communication channels in the hardware architecture.

C. Technology Description

A technology database describes the characteristics of the hardware and software component types. A communication channel connects to a PE or a device through a port. A channel type’s technology model includes three parameters: the cost (in dollars, ECU’s, etc.) of the hardware required to implement a port, the channel’s throughput, and the maximum number of ports which can be connected to the channel. A device type’s technology model includes the type of channel to which the device is connected and the cost of the device. The technology model for a PE type describes its manufacturing cost, its internal devices, and its communication ports. Manufacturing cost is a monetary value which can include the cost of printed circuit board real estate required by the PE and other indirect costs, so long as the indirect cost per PE is fixed. Any on-chip devices and communication ports have their costs included in the total PE cost, so that using an on-chip device or port is cheaper than adding an external equivalent.

The technology database must also specify the implementation characteristics of the processes. Execution times for individual processes are specified by the designer. The technology model gives the maximum execution time of a process for each PE type on which it can run. The execution time can be found by measurement of code running on a processor or through analysis techniques such as those of Park and Shaw [21]. The execution time of a function implemented on an ASIC can be estimated using high-level synthesis techniques like those of Henkel and Ernst [12]; if an ASIC preemptively executes several processes, the techniques of Potkonjak and Wolf [22] can be used to determine worst case performance and to synthesize the shared data path. A process’s worst case execution time may also be estimated by the designer if the process has not yet been implemented. If a process does not have a bounded execution time, then it cannot be used to meet a hard rate constraint. It is possible, however, to use an average execution time to represent the process and treat the rate as a soft constraint.

V. ENGINE/SOFTWARE ARCHITECTURE CO-SYNTHESIS

A. Summary of the Algorithm

Our architectural co-synthesis algorithm’s primary objective is to meet the rate constraint and the secondary objective is to minimize total implementation cost

\[
\sum_{i \in \text{CPU's}} \text{cost} \,(\text{CPU}_i) + \sum_{j \in \text{devices}} \text{cost} \,(\text{device}_j) + \sum_{k \in \text{channels}} \text{cost} \,(\text{channel}_k).
\]  

Co-design includes four major procedures [35]: partitioning of the functional description into a set of processes; scheduling the processes; allocating the processes to PEs; and mapping PE’s into particular component types. Our synthesis algorithm assumes that the description has been partitioned into processes and it implements a process on a single PE. The most important steps to architectural co-synthesis are allocation and mapping; scheduling information serves to compute the most accurate PE utilization to guide allocation.

Synthesis is easiest when scheduling, allocation, partitioning, and mapping can be cleanly separated. However, these steps are much more closely tied in co-synthesis than in high-level synthesis because process execution times vary in small increments, depending on the mapping to a particular type of PE. The circular reasoning required due to the relationship between mapping and the other two tasks can be easily seen: we want to schedule processes and allocate them to PE’s so as to maximize PE utilization and minimize communication costs; but the execution time of a process, which determines utilization, is not known until the process is not only mapped to a particular node in the multiprocessor network but also mapped to a particular type of PE; and we cannot allocate and schedule processes until we know their execution times. The greatest challenge in the development of an efficient co-synthesis algorithm is to break this circular process in allows the algorithm to find a good hardware–software architecture.

We have chosen to break the circularity by first performing an initial allocation and mapping which has enough resources to ensure that all the specification’s deadlines are met, then refining the design to reduce system cost. The co-synthesis algorithm is designed to satisfy design criteria in this order: first, satisfy all deadlines; second, minimize PE cost; third, minimize communication port cost; and fourth, minimize device cost. The algorithm assumes that I/O operations have been assigned to different physical ports; the algorithm allocates devices to minimize system cost, but it does not try to combine several logical I/O operations onto one physical device. The techniques of Amon and Borriello [1] can be used to schedule I/O among and determine the number of physical I/O devices required.

Synthesis proceeds through five major steps.

1) Generate an initial solution: allocate processes to PE’s such that all tasks are placed on PE’s fast enough to ensure that all deadlines are met; schedule the processes to determine process exclusivity and communication rates.
2) Reallocate processes to PE’s to minimize PE cost.
3) Reallocate processes again to minimize inter-PE communication.
4) Allocate communication channels.
5) Allocate devices, either internally to PE’s or externally to communication channels.

While we have described these steps as scheduling or allocation for descriptive purposes, we schedule to test feasibility during several of the allocation phases. The sequence of synthesis steps is designed to refine the design from a feasible one to a minimal-cost architecture.

We found that several elements were important in achieving high-quality results. This synthesis heuristic first tries to minimize PE cost, then communication channel cost, and finally device cost; this sequence of optimizations reflects the relative importance of these components of cost in the examples that we have studied. Our experiments have shown that Step 2 is the most important for achieving minimum cost, so that step performs an iterative refinement; other steps in the algorithm are greedy, as described below. Our algorithm moves several processes at a time when re-allocating. In our experiments with various heuristics, we found that moving several processes at a time in order to eliminate or cost-reduce a PE is also very important to quickly finding a good architecture. Finally, since Step 2 carries most of the burden of reducing the PE cost of the system, it is important to use an optimization strategy which goes beyond simple greedy allocation. As described in detail below, we alternate between trying to eliminate excess PE’s and balancing the load across the existing PE’s in order to perform a more complete search of the design space.

The operation of the algorithm is illustrated in Fig. 5. The process specification includes four processes, an input operation, and an output operation; the input and output operations each require their own devices. The initial allocation of processes to PE’s is conservative, requiring three powerful, expensive processors (i960s) and one slower, cheaper processor.
The scheduling procedure performed in Step 1 reveals that processes $P_1$, $P_2$, and $P_3$ are mutually exclusive in time and thus can share a priority in a single PE; that change is made in Step 2. Step 3 tries to minimize communication cost by swapping $P_3$ and $P_4$, since $P_3$ communicates a larger volume of data with $P_4$. The final steps in the algorithm allocate the communication link between the two PE’s and the I/O devices which implement the input and output operations.

We refer to the process graph and rate requirement as the specification, the current hardware architecture as the engine, and the current software architecture as the process allocation. A useful measure in guiding heuristic choices is the utilization of a PE, which we define here as the ratio of execution time required by currently allocated processes to the specified period; note that this ratio will never be greater than one in a feasible solution.

This algorithm concentrates on microprocessor-based architectures. The algorithm can be extended to synthesize into ASIC’s as well as CPU’s by using a more sophisticated timing model for the PE’s. In a CPU, only one process can run at a time, but an ASIC can support two processes which execute concurrently. The performance estimation procedures of this algorithm could be extended to handle multiple processes executing simultaneously on a PE. Other hardware-software partitioning algorithms, like those of Ernst et al. [11], make similar simplifying assumptions, such that the CPU and ASIC do not execute concurrently.

**B. Steps in Co-Synthesis**

We will describe the steps in more detail using pseudocode. The first two steps assume that sufficient communication capacity will be available to support the chosen allocation of processes to PE’s.

Step 1 first performs an initial allocation of processes to PE’s—the algorithm assigns only one process to a PE and chooses the fastest type of PE available to execute that process. It then schedules processes on the initially allocated hardware engine; if scheduling does not find a feasible schedule for this system (assuming zero communication time) then the problem does not have a feasible solution in the given technology. The scheduling procedure used throughout the co-synthesis algorithm is shown in Fig. 6. During Step 1, since the initial allocation has just been selected, the scheduler is called to find an initial feasible schedule. Our goal is to find a feasible schedule quickly, not necessarily an optimal schedule, and multiprocessor scheduling is NP-complete [14]. However, since we separately determine the allocation of processes to PE’s and our process graphs are acyclic, we can use a simple scheduling approximation as the core of our algorithm. We use Dijkstra’s algorithm [13] to find the longest path through the process graph. However, the longest-path algorithm operating only on the precedence constraints in the process graph will not give a feasible schedule, since processes allocated to the same PE must be scheduled to have nonoverlapping execution times. The allocation of processes to PE’s is given. We approximate the solution to the multiprocessor scheduling problem by inserting extra constraints to force an order of process execution. We use the last computed schedule to choose the order in which the processes on a PE should be executed. As shown in the figure, an ordering constraint is generated to keep the processes executing in the same order and ensuring that the later process’s start time is no sooner than the earlier process’s completion time. (During Step 1 of the algorithm, although we do not have a previous schedule, we also do not need to generate any ordering constraints since each PE is allocated only one process.)

During Step 2, we reallocate processes to minimize PE cost, still assuming that communication is unrestricted. At a single step in the iterative procedure, we try to remove all processes from a lightly loaded PE so that it can be removed from the current engine. The improvement step procedure is outlined in Fig. 7. We consider the PE’s from least to most utilized. Given a PE $c$, we first identify processes which it may be feasible to move to another, existing PE in the engine. We must check the feasibility of the schedule to determine whether the processes can in fact be moved, so we use utilization as an approximation. We may move different processes to various different PE’s in the engine. Some processes may not be movable, so we try creating one additional PE to hold all the leftover processes; that PE is chosen as the cheapest...
void PE_cost_reduction_step(spec,engine,allocation) {
    sorted_PEs = sort_PEs_by_utilization(engine,least_first); /* sort with least-utilized first */
    foreach PE c in sorted_PEs {
        reallocatable = processes in allocation(c) which can be executed on
        other existing PEs;
        /* reducible is the set of all processes allocated to PE c which cannot be moved
        to another process */
        reducible = allocation(c) - reallocatable;
        xtype = cheapest PE type which can implement all processes in reducible;
        /* new allocation moves some processes of c to other, existing PEs and moves
        the remaining processes to a new, cheaper PE */
        new_allocation = old allocation with reallocatable and reducible modifications;
        if (size(reallocatable | reducible) == size(allocation(c)) /* can move all processes */
            && schedule(spec,new_allocation) <= period) {
            if (size(reducible) > 0)
                engine.add_PE(new_PE(xtype)); /* add the new, cheaper PE */
            remove_PE(engine,c); /* get rid of the old PE */
            redistribute(new_allocation); /* actually reallocate the processes */
        }
    }
}

Fig. 7. The PE cost reduction step procedure.

void iteratively_reduce_PE_cost(spec,engine,allocation) {
    lastcost = large_value;
    first = YES;
    do {
        if ((first) lastcost = cost(engine);
            first = NO;
            PE_cost_reduction_step(spec,engine);
            pairwise_merge(spec,engine);
            balance_load(spec,engine);
        } while (cost(engine) < lastcost);
}

Fig. 8. The iterative PE cost reduction procedure.

type which implements all the processes. Using a single,
cheap PE to hold leftover processes is a good heuristic for
minimizing hardware engine cost, since it allows noncritical
processes to migrate to cheaper PE’s without creating too
many additional PE’s. If some processes in c cannot be moved
so some combination of existing PE’s and the newly created
PE, we move on to the next PE for reallocation. We then
check the feasibility of this allocation by rescheduling the
processes (using, of course, the execution times for the new
allocation). If the new allocation is feasible, we move the
processes, delete the now-empty c, and add the new PE if
one was necessary.

The iterative PE cost reduction procedure is summarized
in Fig. 8. This procedure first applies the basic PE cost
reduction step, then tries to merge process sets by replacing
two PE’s with a single PE of smaller total cost, and finally
reallocates processes in the existing engine to balance load,
and continues so long as the engine cost is reduced. The
pairwise merger and load balancing procedures are shown in
Fig. 9. Pairwise merging complements the other optimizations
by identifying a single PE which can cover a larger set of
processes. Load-balancing iteratively tries to move processes
from the most heavily utilized PE’s to less-utilized PE’s;
however, the procedure does not remove more than half the
PE’s load as a heuristic to avoid eliminating this PE. Our
experiments show that the load balancing step helps move
processes to less-expensive PE’s.

Step 3 reallocates processes to reduce communication re-
requirements, since allocation decisions in Step 2 were made
without consideration of the required communication rates.
This procedure, which is outlined in Fig. 10, does not ex-
plicitly take channel capacities into account, but only tries
to reduce total communication between PE’s. The rate at
which interprocess communication occurs is fundamentally
determined by the process graph, but the rate at which com-
munication between PE’s occurs depends on the allocation
of processes to PE’s. Before allocating channels, Step 3
reallocates processes to minimize inter-PE communication
rates. The algorithm considers processes one at a time, starting
with the process p which conducts the most interprocess
communication. The nearest-neighbor processes to p in the
process are identified. Given a neighboring process q, the
algorithm either move directly q onto p’s current PE or to
swap q with some other process which currently resides on
p’s PE.

Step 4 actually allocates the communication channels be-
tween the PE’s. The type of channel used for each link is
determined by the types of ports supported by each PE and
device and port cost. The procedure, illustrated in Fig. 11,
identifies communications which span PE’s and allocates
channels as necessary. Any data dependency which connects
processes allocated to different PE’s requires a communica-
tions channel. Each data dependency edge is labeled with the
size of datum transmitted at each execution, as was described
in Section IV-A. From that information the required data
rate can be computed. We assume that the communication
is scheduled at the termination time of the process at the
source of the data dependency. If the communication can be
permitted on an existing channel, taking into account both
the rate and the schedule, it is allocated to the channel,
otherwise another channel is created. The number and types
of channels which can be created for the PE are specified
void pairwise_merge(spec, engine) {
    for (i = 0; i < size(PEs(engine)); i++) {
        for (j = i+1; j < size(PEs(engine)); j++) {
            mergedprocs = merge(processes(ith_PE(i, engine)),
                                processes(ith_PE(j, engine)));
            repl_types = PE types which implement all processes in mergedprocs;
            if (size(repl_types) == 0) continue;
            cheapest = PE(cheapesttype(repl_types));
            if (cost(cheapest) > cost(ith_PE(i, engine)) + cost(ith_PE(j, engine))) continue;
            new_allocation = old_allocation with i and j processes moved to cheapest;
            if (schedule(spec, new_allocation, last_schedule(engine)) <= period)
                reallocate(new_allocation); /* actually reallocate the processes */
        }
    }
}

void balance_load(spec, engine) {
    sorted_PEs = sort_PEs_by_utilization(engine, most_first); /* sort with most-utilized first */
    foreach PE c in sorted_PEs
        do {
            p = process in c not yet considered;
            x = less-utilized PE in engine which can implement p;
            new_allocation = old_allocation with p moved to x;
            if (schedule(spec, new_allocation, last_schedule(engine)) <= period)
                reallocate(new_allocation); /* actually reallocate the processes */
        } while ((less than half of processes in c moved) ||
               (more processes can be moved));
}

void reallocate_for_communication(spec, engine, allocation) {
    /* sort from most to least output communication rate */
    sorted_processes = sort_processes_by_communication(spec);
    foreach process p in sorted_processes {
        this_cpu = assignment(allocation, p); /* convenient alias for CPU to which p is assigned */
        /* nearby is the set of processes adjacent to p in the data flow graph */
        nearby = processes adjacent to p in process_graph + [p];
        new_allocation = allocation;
        foreach process q in nearby {
            if (q can be implemented on this_cpu)
                new_allocation = q moved to this_cpu;
            else if (q can be swapped with a process r currently allocated to this_cpu)
                new_allocation = q swapped with r;
        }
        if (schedule(spec, new_allocation, last_schedule(engine)) <= period)
            reallocate(new_allocation); /* actually reallocate the processes */
    }
}

Fig. 9. The load balancing and merging procedures.

Fig. 10. The communication minimization procedure.

as devices in the technology description. Step 5 may fail to produce a feasible design if communication channels cannot be allocated to handle the required inter-PE communication. We have not encountered this difficulty in practice, but if it were encountered, it would be possible to reallocate processes and possibly allocate new PE’s to make the communication network feasible.

At this point, process allocation is complete, and the last step of the algorithm is straightforward. Step 5 allocates devices around the engine to ensure that each process has a communication channel to the devices it reads and writes. To minimize cost, on-chip devices are used when possible.

We have analyzed the complexity of this algorithm. Let \( n \) be the number of processes, \( d \) the number of data dependencies in the process graph, \( p \) the number of PE’s, \( P \) be the maximum number of PE types which can implement any one process, \( c \) be the number of channels, and \( x \) be the number of devices specified. Since we use the scheduling procedure of Fig. 6 as a building block in several steps, we will analyze it first. Adding execution ordering constraints requires \( O(n^2) \) time, and Dijkstra’s algorithm also requires \( O(n^2) \) time. The allocation phase of Step 1 is linear in the number of processes; the scheduling procedure used in Step 1 requires \( O(n^2) \) time. The PE cost reduction procedure of Fig. 7, which is used as
void allocate_channels(spec, engine, allocation) {
    D = set of dependencies d in spec.process_graph where d.source and d.sink
    are allocated to different PEs;
    foreach dependency d in D {
        ca = PE to which d.source is allocated; cz = PE to which d.sink is allocated;
        if ((existing channel connects ca and cz) &&
            (existing channel has capacity))
            allocate d to existing channel;
        else {
            allocate new channel of compatible type;
            allocate d to new channel;
        }
    }
}

Fig. 11. The communication channel allocation procedure.

TABLE I

<table>
<thead>
<tr>
<th>example</th>
<th># processes</th>
<th>period</th>
<th>implementation cost</th>
<th>CPU time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Wolf</td>
<td>P&amp;P</td>
</tr>
<tr>
<td>pp1</td>
<td>4</td>
<td>2.5</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>14</td>
<td>13</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7</td>
<td>7</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>0.05</td>
</tr>
<tr>
<td>pp2</td>
<td>9</td>
<td>5</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>12</td>
<td>12</td>
<td>1.1</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>1.6</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>5</td>
<td>5</td>
<td>1.1</td>
</tr>
</tbody>
</table>

the inner loop of Step 2, is dominated by the scheduling check
performed on each PE, and has complexity $O(p^2n^2)$. The load
balancing procedure of Fig. 9 has the same complexity and
the pairwise merger procedure has complexity $O(p^2n^2)$. The
full, iterative PE cost reduction procedure will be executed
at most $p^2n^2$ times, since we must either eliminate a PE or
replace it with a lower-cost PE at each step, so the complete
procedure requires $O(p^2n^2)$ time. Step 3, which reallocates
to minimize communication cost, has complexity $O(p^2n^2)$ since
an iteration may have to examine all the PE’s to determine
the system connectivity. Step 5 is linear in the number of
communication devices.

VI. RESULTS

We have implemented our architectural co-synthesis algo-

rithm, which required about 6000 lines of C++, using the
NIH Class Library. Execution times for processes in all our
examples were either measured from separately implemented
C code or were estimated. All of our experiments were run on
SGI Indigo workstations.

We used examples from related co-synthesis research and
from software engineering texts to evaluate our algorithm: pp1
and pp2 are Prakash and Parker’s examples 1 and 2 [23], the
two examples they used to test their ILP formulation; cfuge
is the centrifuge example of Calvez [3]; dye is the dyeing
machine example of Selic et al. [26]; juice is the juice plant
example of Shlaer and Mellor [28]. For all these examples,
we used the process graph structure given by the authors. For
the examples of Prakash et al. and D’Ambrosio et al., we
were also able to use those authors’ technology parameters:
process execution times for various PE’s and component costs.
For the other examples, we estimated process execution times
based on the description of the functions to be executed
and based our cost values on the catalog, costs of known
components.

Table I compares our results to those of Prakash and
Parker’s algorithm, with pp1 synthesized for four different
specified periods and pp2 synthesized for five different
periods corresponding to the point-to-point interconnection
experiments of Prakash and Parker. The data for Prakash
and Parker’s experiments is taken from their paper. In two
cases, pp1 with period = 3 and pp2 with period = 8, our
algorithm found a slightly higher cost implementation. In
all other cases it found the same implementation as Prakash
and Parker’s algorithm, resulting in the same implementation
cost. The iterative search of Step 2, which alternates between
cost reduction and load balancing, is essential to obtaining
high-quality results. Our algorithm required substantially less
execution time than theirs. Even though the execution times
are reported for different processors, the large difference in
execution time required for heuristic versus ILP solutions
to co-synthesis suggest that heuristic algorithms become
attractive even for relatively small examples.
Table II summarizes the results of our experiments with the other examples. The table summarizes the hardware and software architectures of the synthesized implementations by listing the PE’s, channels, and process allocation. Fig. 12 shows the architecture generated for the dye example as an illustration of the results of our algorithm. Synthesis placed the three driver routines for the external devices in the smaller HC11 and the remaining processes in an i486. Our technology file specified that the driver processes could be implemented only on an HC11, while the other processes could be implemented either on an i386 or i486. Our algorithm determined through scheduling that all the processes could not be implemented on an i386 and chose to use one i486 rather than two i386s to reduce cost. It also determined that all the driver processes could be feasibly executed on a single HC11. The i486 required an external serial port, while the HC11’s internal serial port could be used to implement that side of the connection to the communications channel. This example has the largest number of processes because object-oriented specifications tend to include many small processes. Most of the CPU time required for synthesis went into iterative optimization for PE cost (Step 3 in the algorithm). In the juice example, we specified that the ctank and abatch processes could be implemented only on the 68000 but that the other processes could be implemented on either type of CPU. The algorithm chose to cluster as many processes as possible on the 68000 to minimize interprocess communication delays.

VII. CONCLUSIONS

Architectural co-synthesis is an important tool for the embedded system designer. Architectural decisions made early strongly influence the ease and even the feasibility of the detailed hardware and software implementations. The choice of a hardware-software architecture requires balancing many factors: what operations can be implemented on each available PE type; PE cost; allocation of processes to PE and utilization; interprocess communication requirements; communications capacity; and device cost, among others. An
architectural co-synthesis algorithm is essential for quickly assessing the design space, particularly when system architects and customers must evaluate candidate architectures to help refine the system’s specification.

Our co-synthesis algorithm does not assume an architectural template. As a result, it can synthesize the range of designs to which embedded system architects have been accustomed to being able to implement. The heuristic nature of the algorithm makes it significantly more efficient than mathematical programming, making it feasible for system architects to use the algorithm to generate a number of system specifications and component choices. The algorithm handles more sophisticated component models, taking into account not only CPU times but also the savings afforded by integrated devices and ports.

Our heuristics show that the complex relationship between allocation, scheduling, and mapping in co-synthesis can be managed by allocating first for deadline feasibility, then refining the design to reduce its cost. Our experiments show that the heuristics applied by our algorithm find the optimal solution in many cases and close-to-optimal solutions in many others.

Much work remains to be done in architectural co-synthesis as well as the co-analysis and verification tasks which support architectural design. However, we believe that this co-synthesis algorithm is an important step in developing an understanding of the tradeoffs involved in the design of complex, multiple-processor embedded computing systems.

REFERENCES


