EMBEDDED SYSTEM CO-DESIGN
TOWARDS PORTABILITY AND RAPID INTEGRATION

G. BORRIELLO, P. CHOU AND R. ORTEGA
Department of Computer Science & Engineering
University of Washington, Seattle, WA [USA]

1. Introduction

Embedded system designers, in varied industry segments that include consumer electronics, automotive control, and medical equipment, are facing increased pressure to create products quickly and inexpensively. Moreover, they must create product families that offer customers a wide-range of cost, function, and performance tradeoffs. Another trend is the increasing level of integration, performance, and programmability achievable in off-the-shelf integrated circuits including microprocessors, programmable logic, and devices such as LCDs, network interface controllers, and speech generators. Designers find using these devices to be advantageous because of their low cost and the way they facilitate rapid realization of designs not only for prototyping but for production as well. In fact, with embedded controllers now found in everything from automobiles to smart credit cards, many products have declining lifetimes that make custom integrated circuits a less economically viable option. Thus, the coupling of these two trends is leading to new dominant issues in embedded system design: rapid integration of new components and portability of design specifications to more than one realization. Rapid integration is needed to take advantage of emerging devices (processors, sensors, actuators), re-design of portions of existing designs, and to support rapid prototyping. Portability is needed to keep up with continually shifting technology as well as the development of product families. Ideally, a single description of design functionality should have a long lifetime and be realized in a myriad different ways.

The job of embedded system designers has also changed. The traditional separation of hardware and software responsibilities is no longer workable. The pressures of the market make it impossible to separate tasks among different groups and expect adequate communication to take place. In addition, due to the design trends mentioned above, designers not only must worry about the correctness and

cost effectiveness of their implementation but also have a need to explore a large design space of potential solutions. Yet, no integrated computer-aided design tools are available to help designers with this task in an effective and efficient manner. The design needs to be quickly defined and simulated and then mapped onto the cheapest combination of components. Unlike general-purpose computers, embedded systems are designed and optimized to provide specific functionality. Thus, some of the most time consuming and error-prone tasks in embedded system design are precisely the detailed mapping of the abstract functional specification onto the target components. In fact, the process is so time-consuming that many designers fix the target architecture and system components well before a complete evaluation of the final system and perform only one mapping. This often leads designers to *over-design* their systems with faster processors or larger capacity logic devices than really needed, thereby increasing the cost. If the target architecture were to prove inadequate due to performance or capacity constraints, designers would face a costly re-mapping process. Over-design is currently a necessary evil as it helps to avoid delays late in the product design cycle.

It is clear that design exploration tools are sorely needed to automate the mapping process and thus provide faster feedback on design decisions. Many design automation tools and frameworks have been proposed to address a few of these problems [1]. These tools either look at high-level specifications but do not assist with the actual implementation, or they help with individual parts of the implementation but do not provide a system view. Examples of the former include behavioral simulators and formal specification languages while examples of the latter include compilers, board layout tools, and logic synthesis systems. Recently, tools for dealing with the hardware and software portions of the system have been proposed, but these have not addressed the system integration issues that dominate the design cycle. Nor have they addressed issues of portability of the design, namely the reuse of specifications of all or part of a design so that it can be remapped to new target components. For all practical purposes, designers still use tools almost identical in style to those used twenty years ago.

There are many domains in which embedded systems are applied. Tools clearly can be specialized for a particular domain to take advantage of its specification methods, design styles, and dominant technology. Examples of these include digital signal processing systems implemented on a single chip using core DSP processors. The focus of this work is on control-dominated systems operating under hard-real-time constraints that may be quite fine grained (as in communication protocol implementations). We assume they are constructed from commercial off-the-shelf components and can thus more easily benefit from rapid prototyping and emulation. The embedded systems of interest are centered around one or more processing elements that execute software, with hardware accelerators for time-critical functions. These processors control a collection of peripheral devices and/or communication interfaces, and the whole system is connected to-

gether by glue logic, which is typically implemented in field-programmable gate arrays. This class of systems includes embedded systems for consumer, robotic, communication, and medical applications.

An example to illustrate many of the points brought out in this introduction can be taken from our work in designing a collection of low-cost ubiquitous computing devices based on and omni-directional IR signaling mechanism developed at Xerox PARC [16]. The objective is to connect an entire collection of devices to a wireless infrared local area network. These devices include transceivers connecting workstations, laptops, and palmtops; identification tags for people and objects; receivers that can be used to remotely control objects in the office environment; and wearable devices. Each device must implement the same (or a subset of the same) communication protocol but within very different cost and performance constraints. For example, the cost of identification tags must be very low if they are to be used for objects as small and inexpensive as books, while the cost of a PC-Card-based transceiver for a laptop can use more expensive components that support more robust and efficient communication. The essential elements of this example are that we require the ability to implement similar functionality (data transfer protocol) onto very different target architectures and be able to explore connecting a varied collection of peripheral devices (the objects to be sensed or controlled) to our transceivers.

The remainder of this chapter first turns to the issues of portability and integration in more detail to set the state for a discussion of the new generation of computer-aided design tools that are needed. It concludes with a description of Chinook, a first tool in this new class, and some examples of the types of systems that Chinook can currently handle.

2. Portability

Making a design specification portable has many advantages, ranging from improved documentation and maintainability to more rapid re-targeting of the implementation. With the tools available today for embedded system design, many design decisions are thoroughly embedded in the specifications. These include decisions about how device drivers are constructed, the software architecture of interrupts, threads, and processes, and the real-time constraints that must be satisfied by the design. When a design is to be re-mapped to a new set of components, be it a new processor or new devices, it is difficult to extract the design tradeoffs that were made and re-evaluate them. More likely, designers of the new system will either reuse as much as possible of the old design, thereby ignoring new optimization opportunities, or redo the entire design from scratch while attempting to reconstruct the constraints used to derive the original specification.

In the first case, not having an implementation-neutral specification methodology inhibits the designer from documenting design decisions. Many of these

decisions are made in the rush to implementation or even in a debugging phase. The expectation that adequate documentation can be written during this process is unrealistic. Designers may even be unaware that the fix they are implementing is actually motivated by a design constraint. In the second case, re-implementing the design from a *tabula rasa* requires the new designers to relearn many details already acquired by the previous design team. These can be high-level partitioning tradeoffs or low-level device driver implementations. With ever shortening design cycles, re-investing in such a potentially large learning curve is usually not an option.

A methodology must be developed to enhance the portability of designs. A design specification must begin with an unbiased functional description. Mixing design constraints into the functional description is what leads to buried design decisions as the description is refined in a particular implementation context. Design constraints must be separate and be of a declarative nature. Design constraints may include cost and power budgets and timing constraints. Our primary concern is with timing constraints as these have the biggest impact on the implementation details of the design once the major components and their architecture have been selected.

At the lowest levels are the signaling conventions that must be used to connect the system to its peripheral devices and communication interfaces. There are many fine-grained timing constraints at this level, ranging from setup and hold times to data transfer rates and reaction times. These constraints determine how the device or interface is going to be connected to a controlling processor or processors. It may be directly controlled, with a processor generating and sensing the signaling events, or the processor may be too slow or too busy with other functions such that a hardware controller or a dedicated processor may be more appropriate. An example of this occurred in the design of a data acquisition instrument's front panel designed at a local electronic instrumentation equipment manufacturer. The many devices there (*e.g.*, buttons, probes, displays) had to be connected modularly to the data collection and display processing system. It turned out that it was cheaper to insert another processor to control these devices, and implement a serial connection to the data collection unit, than it would have been to include a very wide connector with more bandwidth. These decisions should not make their way into the specification of the design.

At higher levels, performance or reaction time constraints determine the scheduling of processes and the overall software architecture, including inter-processor communication schemes. The trend today is to use real-time kernels and multi-thread packages to deal with concurrency and deadlines. However, these systems have a very coarse-grained model of timing and impose a notable overhead in the scheduling of processes. Moreover, they impose a particular model of scheduling that may not match the application at hand. For example, most systems support periodic processes but do not directly support the fact that most constraints need

only be respected when the system is operating in certain modes and can be ignored at other times, thereby freeing up processor cycles. Frequently, designers account for the overhead of context switching by the operating system and its scheduling model directly in the code that is generated. For example, they may dynamically create and destroy processes or alter the constraints to compensate for overhead. These fixes greatly diminish the portability of the specification. Timing constraints should not be embedded in the specification but explicitly stated in a declarative manner so that they are clear to both future users of the specification and a synthesis system that will determine how best to schedule the processor's activities.

3. Integration

System integration is an area that can benefit greatly from CAD support. It entails determining the hardware and software required to interconnect system components and enabling them to communicate. Furthermore, the communication channels may impose their own constraints due to the protocol that must be respected (*e.g.*, a bus or local area network) and the communication may have to be completed under timing constraints derived from performance requirements. In general, it is a very time consuming task given the many details that must be worked out precisely right before anything can be made to work.

The problems of system integration are intimately tied to those of portability. What benefits portability concerns will most likely also help in system integration. The situation today is that design specifications are fragmented into many different forms: software (often in C) for each of the processors in the system; low-level software (often in assembly) for device drivers; schematics or hardware description language files for the hardware elements and glue logic; and a netlist that details how all the pieces fit together. Unfortunately, the collection of tools that deal with these separate parts are unaware of each other.

A designer must put together a design in a piece-meal and incremental fashion. First, some glue logic is designed to connect a processor to a device. Second, the software for the processor is completed to the point that it can perform some basic communication with that device. Incrementally, the software is expanded to communicate with another processor or device. Some software can then be developed for that other processor and the two can communicate. Now, the designer can return to the first processor and add more functionality or an interface to another device. Unfortunately, these changes may interact in unexpected ways with the software that was previously working, and may even render it no longer functional. For example, a timing constraint may no longer be satisfied due to a processor doing too many other things in parallel, or a previously existing communication path to a peripheral device is no longer available. These are precisely the types of bugs that cause headaches in the integration and debugging phase of

a project.

Handling these situations is quite difficult with today's tools. The separateness of the tools means that simple things such as the naming of signals is not centralized. A signal generated by a processor may have one name in software, another name in the device driver, yet another in the glue logic that connects it to another processor, where it has yet another set of names there. Larger problems exist when a situation as the one above arises. It is important for a designer to be able to debug software on multiple heterogeneous processors simultaneously and have the respective debuggers be aware of synchronization points, intervening glue logic, and shared memory between the processors. It is clear that designers need a single environment in which they can describe their design and in which consistency can be maintained automatically.

Designers need a tool that facilitates migrating functionality easily between processors or even into dedicated hardware. The designer should be able to specify how components should communicate (*e.g.*, point-to-point, common bus, shared memory) and have the tools generate the necessary glue logic and modified software drivers to appropriately forward parameters and data. This type of facility would remove one of the highest barriers to design space exploration, namely, the cost of simulating, prototyping, and evaluating a new partition or mapping for a design.

4. A New Generation of Tools

Given the problems in portability and integration outlined above, there is currently an excellent opportunity for a new generation of embedded systems design tools that address these bottlenecks. These tools will enable designers to specify a system's functionality and then map that same specification to a wide variety of target architectures. From the feedback available from a detailed realization, designers will then be able to make more informed design tradeoffs. In this section, we will outline the suite of tools that will be required. Development of these tools will entail the adoption of a new design methodology, a necessary step before the benefits of automation can be gained. Specifically, currently fragmented design teams must be made to work together in producing a single unified specification of the entire design.

First and foremost, it must be possible to specify an entire embedded system in a single specification language. This is an excellent goal but probably unrealistic given the wide range of devices. For the purposes of this chapter, we restrict our discussion to the digital portions of the design and assume that any analog components can be packaged into *devices* with a digital interface. It is critical that a language be chosen that does not bias the implementation. For example, some synchronous programming languages (with their zero delay semantics) require a user to already schedule much of the code. The specification should include only

high-level calls to peripheral devices and interfaces through a procedural abstraction. Ideally, the designer will use procedure calls (from an API made available from a device library discussed below) for each of the device's functions. Timing constraints will be specified between labeled I/O operations, as these are the only observable events of the system. Support should be available for rate, latency, and response time constraints.

The debugging of this specification can proceed in two directions that go hand-in-hand: simulation and verification. Simulation is essential for validating that the behavior specified is what was desired. Timing will necessarily have to be ignored at this stage as simulating timing precisely will involve almost as much effort as synthesis (therefore, timing properties are best checked at that point). For now, correctness will be defined by I/O behavior. The device library permits a mock-up of the design by aliasing the procedural I/O to device models such that during simulation it will instantiate the appropriate user interfaces in windows on the screen. Library models for interfaces should also include a way of specifying files for input and output. For example, a local area network interface will include a means of reading a file that contains a packet and save a packet sent by the system to a file. Verification offers opportunities for complete checking of the design independent of a particular scenario. The specification language must be made translatable into the appropriate formalism for a verification system (*e.g.*, finite-state machines).

The target architecture can be specified declaratively in the system specification. It should instantiate all the major components in the system: peripheral devices, communication interfaces to the environment, processors, and their basic interconnections, such as a common bus between some of the components or serial point-to-point connection between others. The specification should also include an assignment of functionality to these major components as well as a specification of which portions of the system should be implemented in custom hardware.

The device library will include a wide variety of standard components and should have facilities for easily adding new elements. The models for each device consist of the physical interface (*i.e.*, pins and form factor), the logical interface (*i.e.*, the signaling events required to implement each transaction supported by the device), behavioral simulation model for each transaction (including a visual representation), and a structural, pin-accurate simulation model to be used after synthesis. The structural model can be built on top of the behavioral model by the addition of an FSM for each transaction. The FSM is essentially a watchdog that looks at the device pins to determine if a particular transaction is being performed and then calls the corresponding behavioral simulation routine.

Once the target architecture is specified, tools should be available to transform the specification automatically so that it makes use of the communication paths specified and communicates the necessary data over them. Communication paths

between components will be turned into device I/O calls. This is important in globally optimizing the generation of glue logic and providing a similar abstraction to device I/O as a first step in synthesis. Data that must be communicated between partitions will serve as parameters for these communication I/O calls. Parameter passing mechanisms will need to be inserted into the specification and these could include hardware registers, shared memory, interrupts, *etc*. This is a critical step to automate as it permits enhanced design space exploration by freeing the designer from specifying all the details of interconnections and communication in detail thus permitting experimentation with many different partitions easily.

Interfaces must be synthesized between the components of the system. Device protocol specifications from the library will be customized automatically to the capabilities of the processors to which they will be connected. For example, a slow processor may require external hardware to implement a fast transaction with a device while a faster processor may be able to handle it directly. In essence, device drivers will be automatically synthesized and customized. This is one of the keys to portability. Designers will not need to write the detailed code of these software routines that change with every change in target architecture or interconnection scheme. Treating communication paths in the same way as devices permits their customization to be done similarly.

What remains is the synthesis of the hardware and software components that implement the functionality of the system. The hardware blocks will be synthesized using behavioral synthesis tools that can handle a rich set of timing constraints. Although these are certainly not available today in general enough form, they should be in the near future. The software blocks will be generated by a compiler specific to the processor at hand. The preparation of the source file to be compiled involves not only the translation of the specification but includes scheduling of time critical and concurrent functions. Code scheduling techniques (including serialization, multi-threading, and interrupts) will be applied to ensure that all timing constraints are met. In essence, these will derive a customized operating system kernel for each processor.

Once synthesis is complete, the design can then be simulated at the structural level (accurate to the cycle level) with the software running on models of the processors. At this point, debuggers and profilers can be used to help evaluate the design. Debuggers must make a larger leap to the original source code as it not only has been transformed by a compiler but it was also modified by the interface and glue logic synthesis process. Debugging a collection of processors and custom hardware can now be done within a unified name space. This will prove invaluable in identifying interactions between processing elements. Profilers will be needed to measure processor and bus utilization, precise timing of I/O events to check timing constraints, and to check on execution times of sections of code. The information gathered can serve to guide the designer towards a different partitioning or different communication scheme.

Some obvious tools are not mentioned above. Most important of these are automated partitioners and software estimators. We believe that architectural-level partitioning will continue to be done by designers. For the class of systems we are focusing on, the problem is much too ill-structured and includes so many variations that it will be some time before partitioning and communication synthesis can be automated completely without sacrificing many design opportunities. Our approach is to make partitioning easy for the designer by eliminating the need to specify all the interconnection logic and software.

Software estimators and retargetable compilers are another class of tools that will be very important, but we are relying on other research groups to generate these tools. For software scheduling and architecture synthesis it will be critical to have tight bounds on the execution time of code fragments and understand the relationship between code that may be running in parallel. The effects of instruction buffers and caches make this problem very difficult and we are currently ignoring these features of embedded processors. Retargetable compilers are also a requirement as designers may want to explore customized processors. Compilers must also permit more controllability of their optimization process and be able to offer guarantees regarding the execution time of code fragments, possibly foregoing some optimization opportunities.

The above has described a set of tools that support a new design scenario for embedded systems. The focus of both the research and industrial communities has been on solutions to point problems. These range from better debugging environments to formal verification techniques. However, they ignore the need for tools with a total system view and that can help with the time consuming tasks of porting a design to a new target, modifying a partitioning, and integrating disparate components so that they communicate properly.

5. Chinook

At the University of Washington, we are taking a first step towards the tool suite described in the previous section. Our approach to the co-synthesis of real-time reactive embedded systems is embodied in *Chinook*, a tool that generates complete design specifications given a single high-level specification of the desired system functionality. Several features distinguish Chinook from other work in this volume. Each is motivated by the observations and concerns outlined in the previous sections. Thus, Chinook is intended for control-dominated designs constructed from off-the-shelf components and addresses portability and integration concerns that will permit more design space exploration by automating tasks that are error-prone or cumbersome.

The following elements of the Chinook system are where the principal innovations lie. It is important to note that what makes Chinook unique is the combination of these elements rather than any single one as each is certainly addressed

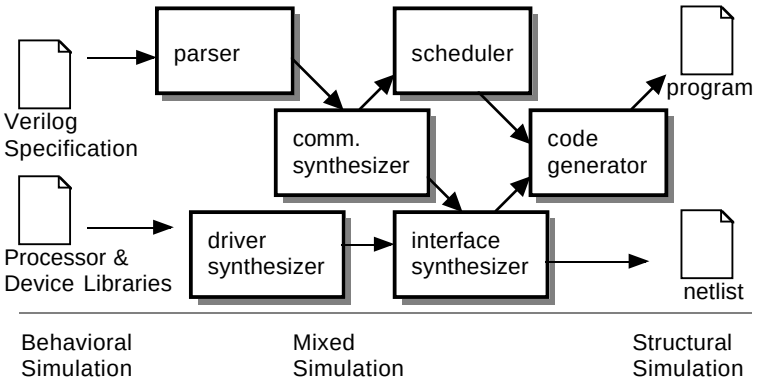

Figure 1. The Chinook Co-Synthesis System

by many other researchers.

- **Single specification.** A designer writes one specification in a single specification language with explicit timing/performance constraints rather than separate netlist, hardware description, and software languages all with implicit constraints. This is key to the retargetability and maintainability of the design.
- **One simulation environment.** The high-level specification of the design can be simulated directly to help debug the designer's intent as well as operational aspects of the design. The final synthesized result, and any intermediate steps, can be simulated in the same environment and augmented with additional tools (*e.g.*, debuggers and profilers for software).
- **Comprehensive software scheduling.** Chinook synthesizes the appropriate software architecture for the timing requirements of the system: low-level partitioning to ensure signaling constraints are satisfied (possibly by synthesized hardware modules), static fine-grained scheduling to tailor device drivers, and customized dynamic schedulers and interrupt handlers.
- **Interface synthesis.** Interface hardware and software between system components (including peripheral devices as well as multiple processors) is automatically synthesized with appropriate changes reflected in interprocessor communication and device drivers.
- **Complete information for physical prototyping.** Chinook generates a complete netlist for assembling the system and complete code for its processors to run. After co-synthesis, the system is ready to be assembled and evaluated in its intended environment.

The Chinook co-synthesis system consists of the parser, the processor/device library, the device-driver synthesizer, the interface synthesizer, the communication

synthesizer, the scheduler, and the simulator (see Figure 1). The parser accepts a system description in annotated Verilog. In addition to a behavioral specification, it also contains a structural specification that instantiates the principal components of the system, including processors, peripheral devices, and standard interfaces. The device library contains detailed generic specification of device interfaces (in the form of timing diagrams and Verilog code) and models for their simulation (in C and Tcl/Tk). For processors it contains specifications of their interfaces as well as timing schemas for software run-time estimation [13]. The device-driver synthesizer compiles the timing diagrams and Verilog device drivers into customized code for the given processor and makes low-level partitioning decisions to meet signaling constraints. The interface synthesizer allocates I/O resources to connect a processor to the peripheral devices it will control, and customizes the access routines to reflect these assignments. The communication synthesizer generates the hardware and software needed for interprocessor communication. With all resources allocated, the scheduler generates code to meet real-time constraints in software. Chinook also outputs the netlist, including the necessary glue logic, to construct the desired system.

Chinook does not attempt several tasks. It does no high-level partitioning of functionality between hardware or software or between processors. Instead, it assumes that designers involved in design exploration are in a better position to make these assignments at the module and/or task level. Instead, Chinook provides a mechanism for designers to easily partition and repartition the specification. In its focus on real-time reactive systems, and due to lack of mature and complete software performance analysis tools, it assumes that caches are not employed.

5.1. SPECIFICATION

The single Verilog file provided as input to Chinook contains both behavioral and structural constructs. The behavioral style imposed by Chinook enables the expression of real-time reactive behavior as well as facilitating partitioning. The structural component merely lists the processors, peripheral devices, and communication interfaces that will be used. That is, the principal components of the system to which the designer would like to evaluate a mapping of the desired functionality. Chinook expects the designer to *tag* tasks and modules with the processor that is preferred for their implementation. The implementation of untagged modules/tasks is assumed to be in software. This separation of functionality from components allows the designer to quickly explore the design space by instantiating different processors and alternative peripheral devices without modifying the behavioral specification. All interactions with the devices and interfaces are specified using a procedural abstraction layer. As long as two interfaces (*e.g.*, SCSI and PC-Card) support the same access routines (*e.g.*, `read` and `write`) they can be easily interchanged.

To model the reactive behavior of control-dominated applications, we organize the control states of the system as a set of *modes*. Each mode defines a behavioral regime, that is, how the system should respond to its inputs. A mode also defines a scope for a set of timing constraints that must be satisfied while the system is within that mode but not necessarily when it is operating outside of it. Inter-modal constraints are used to describe response times when the system transitions from one mode to another. Modes are very similar and are inspired by the hierarchical states of [10] in that they can capture both sequential and concurrent behavior.

Chinook allows the specification of real-time requirements in terms of minimum and maximum separation between I/O events, namely events between system components or between the system and the environment. At the low level, the constraints may correspond to setup and hold times, or simply the sequencing constraints between successive I/Os. At the high level, min/max separation can also be used to express *response times* to system inputs and *rate* constraints on performance [7].

In a given mode, the system's responses are defined by a set of *handlers*. Conceptually, they are event-triggered routines, but their activation conditions are checked by a time-triggered loop. Handlers respond by causing a mode transition to generate I/O events. A handler consists of a trigger condition and a body. The trigger condition is an event expression consisting of inputs from the environment and other handlers. When the trigger condition evaluates to true, the handler body is executed. For example, a network interface chip may signal that a message is pending and this triggers a handler to read that message. Note that the handler body can be in software, hardware, or a combination of the two, depending on its tag and the ability of the processor to meet the timing constraints in the handler. From a specification point of view, a handler is executed atomically, but may be interleaved by the scheduler.

5.2. SCHEDULING

Embedded systems have timing constraints at different levels. Their interaction with the devices and the environment must respect not only low-level signaling constraints but also performance requirements such as rate and response time constraints. To satisfy these high-level constraints, designers have used *process-based* scheduling techniques based on operating systems concepts [12, 2]. These techniques are coarse-grained, priority-driven, and dynamically preemptive. They assume that the processor does not perform I/O directly (*i.e.*, does not manipulate I/O pins and does not have to deal with details of device timing as these are handled by device interface units) and the processes are only loosely dependent on each other if at all. Since all timing constraints are coarse-grained, overhead incurred by the executive during preemption can be dismissed. However, many embedded systems must perform direct I/O and meet fine-grained timing con-

straints. These constraints are much more difficult to meet because the scheduler cannot afford to incur much, if any, run-time overhead, and at the same time must handle uncertainties in the execution delays. Instead, Chinook statically schedules all low-level I/O and high-level operations as grouped in modes. A customized dynamic scheduler may be generated for the larger modes (*i.e.*, those at the top of the mode hierarchy).

Chinook uses a static, nonpreemptive scheduling algorithm to meet min/max timing constraints on fine-grained operations with delay ranges [4]. It determines a serial ordering for the operations, and inserts delays to meet minimum constraints, if necessary. Because the complexity of the problem is NP-hard, we employ heuristic ordering functions to help the exact algorithm quickly find a valid and short schedule. Experimental results show that our approach consistently outperforms and can handle a wider range of scheduling problems than the algorithm of [9].

At the high level, rate constraints are specified on a reference event between successive iterations, and response times are constraints on the time it takes to do a mode transition. In statically scheduling the software, Chinook first converts handlers within a mode into a single handler containing their bodies, possibly using unrolling, and then schedules this single partially-ordered handler by interleaving [7]. Note that a mode transition may be triggered by one of the handlers before other handlers run to completion, and the scheduler must maintain the integrity of all handler states. We use the dual of critical regions, what we call *safe points*, to specify explicitly when it is safe to effect a mode transition safely (rather than the implicit method of critical regions which requires the user to specify when transitions are not safe) [3]. All parallel handlers must reach their safe points before a mode transition is allowed to take effect thus guaranteeing that the system will be left in a consistent state.

5.3. INTERFACE SYNTHESIS

Interface synthesis is the realization of communication between components via both hardware and software elements. Chinook handles a wide range of interface synthesis problems [5, 6]. At the lowest level, Chinook synthesizes device drivers directly from timing diagrams. It generates customized code for the particular processor used, and separates out the portions that cannot be implemented in software by synthesizing the required external hardware. For processors with general purpose I/O ports, Chinook employs an efficient heuristic for connecting devices and processors using minimal interface hardware. For processors without I/O ports, Chinook automatically implements the interface using memory-mapped I/O, by allocating address spaces and generating the required bus logic and instructions.

These synthesis solutions require knowledge about the interfaces of the processors and the devices, which are captured in the libraries. A processor is de-

fined by its I/O resources, built-in functionality (*e.g.*, serial-line controller, timer, etc.), and detailed architecture templates (*e.g.*, down to the specific resistors and capacitors required for power-up reset). A device description contains interface information including ports and skeletal access routines that encapsulate timing diagrams. After successful interface synthesis, Chinook updates the access routines by binding the device ports to the processor's I/O ports or memory bus, and taking into account any intervening glue logic that it may have synthesized. By managing these connectivity details and generating the interface across the hardware/software boundary, the interface synthesizer completes the design and enables simulation and evaluation at the final implementation level.

5.3.1. *Driver Synthesis from Timing Diagrams*

At the most detailed level, device interfaces are described in data sheets in the form of timing diagrams. They show the sequences of signaling events that make up I/O transactions across the interface. These timing diagrams are usually annotated with timing requirements, timing delays, and timing guarantees. The first of these three are requirements imposed on the user of the interface, while the second two are timing promises made by the device as long as the user conforms to the requirements. When new devices are added to the device library, these constraints and their corresponding timing diagrams are entered via an interactive editor [8]. Chinook parses these timing diagrams and synthesizes the device driver code by choosing a linear schedule of controller events, and inserting additional interface glue logic where necessary [15].

5.3.2. *I/O Port Allocation*

Many processors used in embedded systems include I/O ports that can be used to directly sense and manipulate the processor's environment. These ports can be accessed from software like registers, thus providing a low-cost and straightforward interfacing mechanism. Chinook provides a port allocation scheme that uses minimal amount of glue logic. Furthermore, device access routines are customized to reflect the assignments of pins [5]. The key idea is that an I/O port may be able to service multiple devices without glue logic and without performance penalties. These devices have interfaces that are able to isolate themselves from the shared bus, and become active only when the appropriate control signals, or *guards*, enable them. Thus, a guarded interface of a device can share the same I/O port with other devices because their interfaces cannot be active at the same time. If necessary, the port allocator inserts glue logic to add guards to previously unguarded interfaces, so that they can share busses. Chinook can also synthesize ports to create a new interfacing point for additional devices. Hardware is synthesized to create a new port on the processor's memory bus. This module decodes addresses and translates them into control signals to read and write the new I/O pins.

5.3.3. *Memory-Mapped I/O*

When I/O ports are too inefficient (due to multiple instructions to manipulate their values or too much additional hardware) or are unavailable (as is the case for higher-performance processors), Chinook synthesizes the interface using memory-mapped I/O [6]. Many parts, processors as well as peripheral devices, are designed with memory-mapped I/O in mind. They contain built-in address matching logic and can be connected to the memory bus with little or no glue logic. Other components without built-in address comparators can still be connected with little or no glue logic, depending on the available address space the user reserves for I/O. Devices are allocated portions of the address space of the processor controlling them. If the allocation is done intelligently (*i.e.*, using one-hot, binary, or Huffman encodings when possible) the amount of address matching logic required can be minimized.

Memory-mapped I/O is also a preferred method of interprocessor communication and can be used to support both point-to-point and shared memory schemes. If we are to allow a designer to explore mapping of functions to multiple processors, then the mapping tools must automatically synthesize the interprocessor communication hardware and software. Essentially, the view from one processor is that the other processors are just more peripheral devices requiring their own device drivers.

5.4. COMMUNICATION SYNTHESIS

Requirements for faster response times and increased modularity frequently guide embedded system designers to employ multiple processors. These processors are often heterogeneous as cost and modularity concerns drive designers to tailor processors to specific functions. CAD support is non-existent for these types of systems. There are not even debuggers to support concurrent development of programs on two identical processors. Designers find heterogeneous multiple processor systems the most difficult to debug and thus constrain designs unnecessarily just to make debugging tasks tractable.

Chinook provides support for interprocessor communication by synthesizing the hardware and software needed to transfer data between processors. A designer tags the procedures and modules with the processor that should be used to implement them. Chinook then determines the data that must be transferred and the mechanism to use for those transfers, including the interconnections between the processors, glue logic, and buffers and memory.

In meeting timing constraints, Chinook will adjust the interface between the software running on the processors. Consider the case of a fast processor communicating with a slow one. Handshaking with the slow processor may cause the fast one to violate its constraints. Buffers can reduce the load on the fast processor by eliminating direct handshaking. The communication becomes non-blocking and

data may be processed in bursts.

5.4.1. *Interprocessor Communication Synthesis*

When considering communication in multiple processor systems, many new issues arise including predictability, interconnect topology, access to peripheral devices, and communication protocols. The interconnect topology could be bus-based, point-to-point, or a hybrid scheme. A peripheral device may only be accessible via a designated processor or many processors may have shared access. The communication protocol may be contention based or statically scheduled, blocking or non-blocking, and master-slave or peers. Each choice has impacts on performance, predictability, and the complexity of scheduling and hardware required. Chinook supports most of these choices, but by default uses a model suitable for real-time control-dominated applications. It is based on non-blocking communication among peers with designated peripheral processors. The interconnect may be either point-to-point or bus-based.

A handler communicates with the environment through device driver calls and with other handlers via messages. A message is an event that triggers another handler with an optional data value. Intraprocessor messages are implemented with shared variables. Interprocessor messages are transmitted via communication channels synthesized with elements from a communication library that contains buffers, FIFOs, arbiters, and interconnect templates. Given a partitioning of handlers provided by the user, Chinook will synthesize communication channels to satisfy timing and resource constraints. Once the communication components are chosen, they are connected to the respective processors using the interfacing techniques in section 5.3. If there are multiple communication channels between processors, each channel may be mapped to its own physical connection or they may share connections.

5.4.2. *Migration between Processors*

Keeping in mind Chinook's focus on aiding the designer's exploration of the design space, it is important that the designer be free to easily allocate functionality to different processors. Through assignment tags in the high-level specification, a designer can rapidly change the partitioning of functionality – between two processors, or between a processor and a direct hardware implementation. Because Chinook synthesizes interprocessor communication channels and optimizes their use, this task is greatly simplified for the designer. No longer does the designer need to radically alter code running on one processor and then propagate the changes to the others while keeping track of all the potential implications on timing requirements and resource access. These adjustments are made automatically by Chinook.

Migrating functionality is divided into three parts: input parameter sending, control sequencing, and output parameter receiving. Input and output parameters

are mapped to latches or memory locations which are connected to the processor using the interfacing techniques discussed earlier. The control sequencing may simply be moved to another processor or be moved to hardware where it will be instantiated as a finite-state-machine and data-path. The general solution to this requires behavioral synthesis but is quite straightforward in most cases involving I/O. The original software is replaced with routines that pass the inputs, kick-start the hardware or the software handler on the other processor, and then read back the outputs [6].

5.5. SIMULATION

The design can be simulated at different levels of detail. The initial specification is compatible with behavioral Verilog and is simulated without exact timing or detailed I/O. As the synthesis steps refine abstract communications and operations into more concrete signals and components, outputs from intermediate design steps and the final implementation can also be simulated with cycle-level accuracy.

The simulator uses the Verilog-XL Programming Language Interface [14] to communicate with peripheral device models. The device models may be written in Verilog, C, or Tcl/Tk and make X-window calls to visually represent the simulated device. Each device model exports the same API (application program interface) for simulation and synthesis. To simulate the specification during the early stages of the design, the API is bound to a behavioral simulation model. For example, a SCSI device exports a `send` routine. During simulation, the user may pop-up a window containing the various fields of a SCSI packet. After creating a new packet, the designer selects the send option which calls the `send` routine. This enables the user to simulate the environment of the system being designed in a consistent manner. During structural simulation of the system, the device's pin interface is modeled by running multiple FSMs to recognize all possible I/O sequencings in parallel. The FSM that matches the given I/O invokes the corresponding behavioral routine to simulate the device's reaction to the given waveform.

Chinook uses RTL-level processor models for simulating the final system implementation. The processor model, written in C, interprets the same machine code that runs on the actual processor. At this stage, it is possible to execute the software with a debugger (although this is the synthesized code and not the original Verilog source). The binary code is disassembled and the registers, program counter, stack, internal memory, and built-in devices are visible in the processor status window. The processor model faithfully reproduces, within cycle-level accuracy, the appropriate waveforms on the processor's pins. We are currently investigating more efficient simulation techniques that will permit cycle-level accuracy for I/O operations but will run compiled code for the computationally intensive

portions of the software. This is an important problem as the slow speed of detailed simulations is currently a major bottleneck in design-space exploration.

6. Examples

Several embedded systems have been designed using the Chinook tools. The following examples show the type of complexity that the current version supports. They are a portable electronic phonebook, a PC-Card-based logic analyzer, a node controller for a distributed system, and a mobile defibrillator.

6.1. PORTABLE ELECTRONIC PHONEBOOK

The Portable Electronic Phonebook was originally designed by senior undergraduate students. Taking their implementation, we reverse-engineered a high-level specification, which was run through the Chinook tools. The generated solution required less hardware than the original implementation due to the interface synthesis algorithm. We were able to simulate the entire system at the behavioral and structural levels to validate the design. After building this application in hardware according to the generated netlist, the system operated correctly upon applying power.

6.2. PC-CARD-BASED LOGIC ANALYZER

The Logic Analyzer Card is a data acquisition card that can be inserted into a laptop or palmtop (we use an Apple Newton). It can be configured to look for a particular trigger condition on its 15 sampling wires and capture up to 8192 samples of data into internal memory. The memory contents are then forwarded to the Newton for display. The device can also be configured from the Newton. The card employs a microcontroller to handle communication with the PC-Card interface and an FPGA to perform the data sampling. There are two paths to memory: writes of samples controlled by the FPGA and reads from the microcontroller through the FPGA which also implements the required address latches. The solution generated by Chinook includes a small amount of external hardware to handle handshaking on the PC-Card interface (the processor is not fast enough to respond with a WAIT signal to a data request) and a module synthesized as external hardware (implemented in the FPGA) to handle the sampling of data. We were able to simulate the entire system at the behavioral and structural levels to validate the design.

6.3. MAGIC

The MAGIC (Memory and General Interconnect Controller) is a custom node controller for the FLASH architecture [11]. It communicates with a processor,

network, I/O devices, and DRAM. We modeled this architecture with three handlers, one for the processor requests, one for the network requests and one for the I/O requests. Since the DRAM does not initiate activity, it does not require its own handler. All communication with the DRAM occurs via device driver calls. We used the MAGIC application to experiment with using a common API for different peripherals. The specification was written so that it is easy to select a SCSI or Ethernet network interface chip. This demonstrates that designers can easily explore different high level options and observe their ramifications on other parts of the system. Now that we have both SCSI and Ethernet chips and drivers in the device library, it is straightforward to implement other systems that require these protocol chips. Using the results synthesized by Chinook, we performed our experiments with the simulator.

6.4. A MOBILE DEFIBRILLATOR

The purpose of the mobile defibrillator is to revive heart-attack victims with a powerful electrical shock. We consider the digital control subsystem containing an extensive interface including display of ECG waveforms, voice synthesis, digital audio recording, and PC-Card non-volatile storage. Because of the difficulty of guaranteeing that all timing constraints would be respected, the commercial version of this application was designed with a microcontroller and an ASIC. We are currently exploring solutions using reprogrammable components.

7. Conclusion

With the increasing availability of inexpensive and powerful microprocessors and FPGAs, designers of embedded systems are faced with more implementation choices than ever and given less time to realize their designs. Unfortunately, computer aided design tools are not tracking these trends. The Chinook co-synthesis system facilitates design space exploration and automates many aspects of system integration. These are often the most time-consuming and error-prone tasks in the embedded system design process.

Design space exploration is enabled by the use of a single system specification that captures the reactive real-time behavior of the system and appropriately abstracts interactions with the environment to enhance retargetability. Since timing requirements are critical for many embedded applications, Chinook uses static scheduling to guarantee their satisfaction by construction. Several interface synthesis techniques are employed to interconnect system components. The necessary interface hardware and software is generated automatically and minimal glue logic is introduced. At a higher level, Chinook facilitates easy migration of functionality among processing elements and manages the communication requirements between processors. This enables designers to rapidly evaluate different architectural templates and partitionings. Simulation is supported throughout the design

cycle from the initial behavioral specification through the final structural implementation. Chinook's output consists of a netlist, logic specification, and code for each processor – all the elements needed for the construction of the complete system.

We have used to Chinook to synthesize several embedded systems including an electronic phonebook, SCSI interface to a VLSI chip tester, hand-held logic analyzer, and an infrared network transceiver. We are currently experimenting with its use in evaluating the design spaces for an automatic defibrillator and a multiprocessor I/O subsystem. Future work includes developing synthesis methods for more efficient communication using higher level knowledge about the dataflow and control dependencies of the handlers. For instance, routing data around a processor may reduce processor load and yield higher performance at the cost of additional hardware. Ongoing work includes making Chinook more robust and more integrated, especially between the scheduler and compiler/estimator. In addition, we are investigating techniques to permit partitioning between software running on a workstation/PC and functionality in a peripheral device, which is an embedded system on a board attached to the system bus or other standard interface such as serial-line or PC-Card slot.

This chapter has attempted to motivate the need for a new generation of tools that directly attack the problems of portability and integration of embedded system designs. The Chinook project is making the first steps in this direction.

References

1. W. P. Birmingham, A. P. Gupta, D. P. Siewiorek. Automating the Design of Computer Systems. *IEEE Transactions on CAD of Integrated Circuits and Systems*, Vol. 9, No. 5, May 1993.
2. D. Cathey. All Things Considered... Important Factors in Choosing a Real-Time Development System. *Real-Time Magazine*, 2nd quarter 1993.
3. P. Chou and G. Borriello. Software Scheduling in the Co-Synthesis of Reactive Real-Time systems. In *Proceedings of the Design Automation Conference*, June 1994.
4. P. Chou and G. Borriello. Interval Scheduling: Fine-Grained Software Scheduling for Embedded Systems. In *Proceedings of the Design Automation Conference*, June 1995.
5. P. Chou, R. Ortega, and G. Borriello. Synthesis of the Hardware/Software Interface in Microcontroller-based Systems. In *Proceedings of the International Conference on Computer Aided Design*, November 1992.
6. P. Chou, R. Ortega, and G. Borriello. Interface Co-Synthesis Techniques for Embedded Systems. In *Proceedings of the International Conference on Computer Aided Design*, November 1995.
7. P. Chou, E. A. Walkup, and G. Borriello. Scheduling for Reactive Real-Time Systems. *IEEE Micro*, Vol. 14, No. 4, August 1994.
8. B. Gladstone. Specification of Timing in a Digital System. *ASIC and EDA*, August 1993.
9. R. K. Gupta and G. De Micheli. Constrained Software Generation for Hardware-Software Systems. In *Proceedings of the Third International Workshop on Hardware/Software Co-design*, September 1994.
10. D. Harel. StateCharts: a Visual Formalism for Complex Systems. *Science of Programming*, 8, 1987.

11. J. Kuskin et al. The Stanford FLASH Multiprocessor. In *21st Annual Internation Symposium on Computer Architecture*, 1994.
12. A. K. Mok. The Design of Real-Time Programming Systems Based on Process Models. In *Real Time Systems Symposium*, 1984.
13. C. Y. Park. *Predicting Deterministic Execution Times of Real-Time Programs*. PhD thesis, University of Washington, 1992. Technical Report 92-08-02, Department of Computer Science & Engineering.
14. *Programming Language Interface Reference Manual*. CADENCE Design Systems, Inc., 1992.
15. E. Walkup, G. Borriello. Automatic Synthesis of Device Drivers for Hardware/Software Co-design. International Workshop on Hardware-Software Co-design, October 1993 and as University of Washington, Department of Computer Science and Engineering, Technical Report 94-06-04, June 1994.
16. M. Weiser. Some Computer Science Issues in Ubiquitous Computing. *Communications of the ACM*, Vol. 36, No. 7, July 1993.