Codesign of Embedded Systems: Status and Trends

ROLF ERNST
Braunschweig University of
Technology

EVER-INCREASING EMBEDDED-SYSTEM design complexity combined with a very tight time-to-market window has revolutionized the embedded-system design process. The concurrent design of hardware and software has displaced traditional sequential design. Further, hardware and software design now begins before the system architecture (or even the specification) is finalized. System architects, customers, and marketing departments develop requirement definitions and system specifications together. System architects define a system architecture consisting of cooperating system functions that form the basis of concurrent hardware and software design.

Interface design requires the participation of both hardware and software developers. The next step integrates and tests hardware and software—this phase consists of many individual steps. Reusing components taken from previous designs or acquired from outside the design group is a main design goal to improve productivity and reduce design risk.

Figure 1 shows the structure of a design process, highlighting the hardware and software design tasks. A concurrent design starting with a partially incomplete specification requires close cooperation of all participants in the design process. Hardware and software designers and system architects must synchronize their work progress to optimize and debug a system in a joint effort. The ear-

ly discovery of design faults, a prerequisite for hitting the market window, is a central requirement to that cooperation.

A heavy burden is placed on the system architect, who must make decisions based on predicted technology data. To this end, reliable design estimates are essential. Today, such estimates are based on experience and reused components.

Reuse depends on libraries. Libraries of system functions have a higher reuse potential than libraries of physical components with a fixed layout (they become obsolete as technology progresses). The challenge is to support the migration of system functions between different technologies and between hardware and software without a redesign.

What I have described thus far is already a hardware-software codesign process to some extent, but it still lacks a unified approach. This unified approach is the aim of computer-aided hardware-software codesign.

Modeling and verification

A major problem in the design process is synchronization and integration of hardware and software design. This requires permanent control of consistency and correctness, which becomes more time consuming with increasing levels of detail. In hardware-software cosimulation, software execution is simulated as running on the target hardware. Since gate- as well as register-transfer-level (RTL) hardware simulations are too slow for

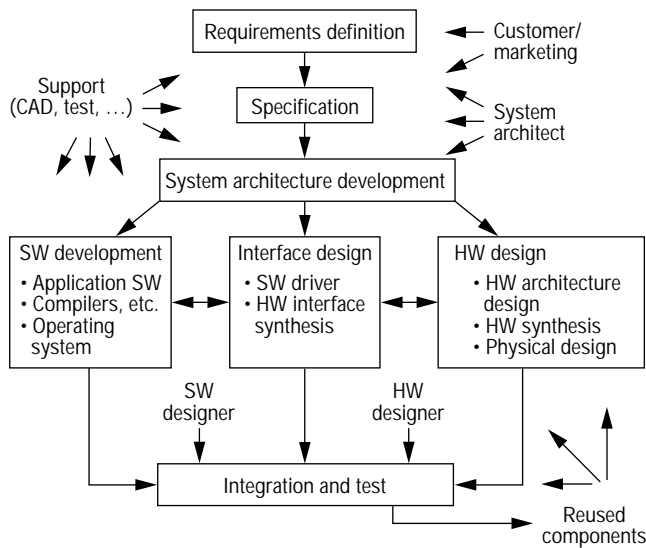New methodologies and CAD tools support an integrated hardware-software codesign process.

Figure 1. Embedded-system design process.

practical software simulation, abstract execution models are needed.

For that purpose, the processor is modeled at a higher level of abstraction than the implemented hardware part. The cosimulation problem[1] lies in coupling the different models to make the hardware simulation sufficiently accurate. In the worst case, the processor and hardware obtain arbitrated access to the same bus and memory. Accurate timing in this situation requires adapted memory and bus models and new simulation strategies. An example of this approach is the cosimulation tool Seamless CVS from Mentor Graphics. CVS uses a target processor model (an instruction set simulator) and bus models that abstract processor interaction with the memory depending on concurrent access from hardware and software.

The main issue is the availability of a library containing processor and memory models provided by the electronics design automation (EDA) vendor or the core processor provider.[2] Library standardization efforts, such as VSI Alliance, might be able to overcome some of the model compatibility problems.

A more abstract approach reduces the processor execution to the pure functional behavior without timing and only models the interface timing behavior. The software runs on any target platform, preferably the host workstation or PC. Software execution couples to the hardware model via a simulator-specific hardware-software communication protocol that requires a designer to perform software enhancement. Only hardware interface models are necessary, thus alleviating the library problem. On the other hand, while the software function and the interface timing can be analyzed, timing and performance analyses are restricted to the hard-

ware part (for example, the tool Eaglei from Viewlogic).

The current cosimulation approaches work best on larger ratios of internal processor operations to processor I/O operations and hardware activity. In this case, the execution time can be reduced by more than three orders of magnitude compared to complete RTL simulation, up to the full host performance for the more abstract second approach. Designs with application-specific processors (ASIPs) and core processor extensions with special-function coprocessors are harder to deal with, due to the custom hardware's much higher activity. In this case, custom abstract cosimulation models must be developed as part of the design process. When models for these highly active parts become available, the cosimulation process can continue as usual.

A second problem is the early detection of design faults. The design team creates an abstract system model—a cospecification simulated or formally verified for certain system properties. Simulation requires an executable cospecification, also called a virtual prototype. Numerous system design tools from different application domains support cospecification simulation, including STATEMATE, MatrixX, MATLAB, COSSAP, or Bones and SPW.

Rapid prototyping with hardware-in-the-loop emulates the physical behavior of a system and can replace virtual prototyping. This proves necessary when simulation time dominates design time.

Executable specifications face several problems. Depending on the application domain and specification semantics, they are based on different models of computation. Some support modeling and simulations of event-driven reactive systems, while others target dataflow systems. A combination using both domains (for example, telecommunications) implies simulation overhead. The inclusion of reused components and functions that must match the input specification's level of abstraction remains a problem. Finally, virtual prototypes do not cover most of the nonfunctional constraints and objectives, such as power consumption or safety.

Synthesis

Industrial tools for system synthesis are not as developed as modeling and analysis tools. On the software side, we can use real-time operating systems (RTOSs) for load distribution and scheduling. Codesign, however, requires a closer look at processes and communication. The second problem is code generation. Specialized architectures, such as digital signal processors or microcontrollers, dominate the embedded-system market due to their superior cost efficiency—especially compared to modern general-purpose processors.[3] Special processor compilation, however, is in many cases far less efficient than manual code generation. Consequently, a considerable amount of assembly coding in embedded systems is still observed.[4]

Even if compilation improved, there is still the problem of generating efficient compilable code (such as C) from abstract models. This problem can be solved for small reactive systems with finite-state machine behavior and with simple operations on data. However, as soon as more complex data operations are required, the design space grows tremendously and the design, when executed manually, includes target-system-specific transformations. Good examples include memory optimization requiring target-architecture-dependent loop transformations, optimized word length selection, and process restructuring for fine-grain load distribution. For example, C code developed for the TI TMS320C6x is not optimized for running on Philips TriMedia or MPACT processors. The problem is worse here than with parallel compilers because of architecture specialization.[5] Porting functions between hardware and software implementation becomes particularly cumbersome.[6]

Fortunately, given a certain state of circuit technology, the choice of hardware or software implementation is predetermined for many system parts. However, this border moves with technological progress and new constraints. Power minimization and increased flexibility requirements drive this development.

To be competitive, an automated code generator must cover a large design space using transformation rules, and this far exceeds current techniques. Nevertheless, there are many commercial C or VHDL code generators—such as STATEMATE, MATLAB, or MatrixX—suitable for prototyping and acceptable for the final design. They are suitable in cases of simply structured target architectures and low-to-moderate cost efficiency and performance requirements. Other tools restrict the processor types (usually to standard microprocessors) and the language scope (for example, the Cmicro code generator for the SDL tool suite of Telelogic). Guaranteed timing behavior of the generated code is another problem.

One can circumvent the code generation problem with libraries of predefined and parameterized code modules adapted to an application. This of course requires a matching of input to target system modules based on a large module library. User-defined and library parts are then combined with a suitable schedule. This corresponds to a partially automated design for a specific system domain. The COSSAP, SPW, and Bones tools fall under this category, as well as the Mentor Graphics DSP Station.

On the hardware side, we see a growing set of high-level synthesis tools: the Behavioral Compiler of Synopsys, Monet of Menor Graphics, and RapidPath of DASYS. While this is a big step forward for cosynthesis, we still need to look at problems with memory optimization, parallel heterogeneous hardware architectures, programmable hardware synthesis and optimization, and communication optimization,

to name just a few. Similar to software synthesis, the design space is still narrow when compared with a manual design. Interface synthesis has been neglected for a long time in commercial tools, and only recently have the first commercial systems appeared. The CoWare system (as an example of an interface synthesis generating both hardware and software parts of a protocol) and the Synopsys Protocol Compiler (as an example of a hardware interface synthesis tool) represent this group.

In summary, there are tools that reach a remarkable degree of automation for specific applications yet do not exploit the design space to obtain an optimized solution. Other tools create competitive designs, but only for very specific problems using hardware or software component libraries, leaving the rest of the design to be created and integrated manually. Besides the integration effort, development of such specific libraries ties up design capacity.

The current synthesis tool landscape leaves the impression of a patchwork of partial solutions that must be mixed and matched by the designer. Portability is rudimentary at best. The situation is certainly better than not having synthesis tools, but parts can hardly be expected to grow together over time into a homogeneous system. Interestingly, the current discussion on reusable intellectual property circuit functions focuses on circuit technology, library, and interface compatibility issues. These logistical problems sometimes seem to block the more fundamental design issues discussed earlier.

Exploration and optimization

An integrated and coherent codesign system should capture the complete design specification, support design space exploration with optimization based on this specification, and, if possible, cosynthesize a selected design point. Research has provided numerous contributions toward this goal, but issues of completeness and design process integration still arise.[7] Even push-button cosynthesis approaches (which have been demonstrated for special applications and architectures) can only solve part of the design problem.

Complete design capture

Some of the languages previously mentioned for executable cospecifications in commercial systems and many others (including VHDL and C)[5,8-10] can serve as input to the codesign process. In general, systems are described as a set of communicating and concurrent entities activated at a given time or upon arrival of events or data. The name "process" denotes such an entity. Upon activation, the process executes a function, thereby changing or generating output data and reading or consuming input data. Depending on the language, these functions can range from very simple, logical operations to very large, high-level programs.
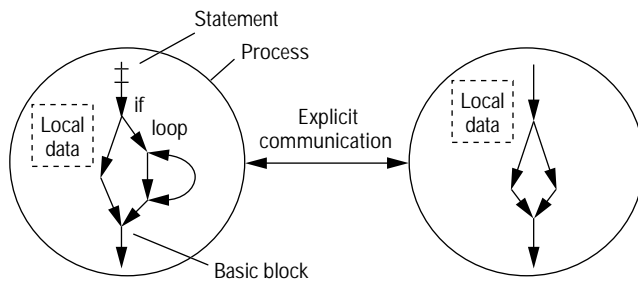
Figure 2. Communication in process partitioning.

To be complete, the input description must contain all design constraints, such as timing requirements. It can also contain information such as preselected components or cost functions to guide the design space exploration process.

At this point, the design might still not be fully captured. Think of a telecommunications system used in a larger network or a video coder optimized for maximum average performance. In these cases, we need additional information about process execution and input/output data (to analyze the internal execution paths of the processes). Current cosynthesis systems expect either profiling data or execution traces. These data are also part of the design description.[11]

Global optimization

The high-level and general views of cospecification as an input to the cosynthesis process sufficiently identify some of the main problems in design space exploration and optimization. First, the activation rules of the processes are modified for implementation. A process, ready for execution upon arrival of an external event, might eventually be executed by interrupt or periodically, polling for the event. The introduction of buffers delays process execution, thus widening the design space; it also supports pipelining. There is a rich variety of solutions originating in real-time software development.[12] Design constraints, mainly the required timing, limit feasible solutions. As a consequence, the input and target models of computation can be quite different, affecting the individual components as well as their interaction. This requires global system optimization.

Global optimization has become more complicated due to the combination of (static) data flow and reactivity in a single application. Take an MPEG2 encoder generating a fixed bit rate signal to be transmitted over a communication channel with a very small receiver buffer, as used in digital video broadcasting.[13] In this application, adaptive video coding that controls the buffer level is useful.[13] We can still describe the system as a dataflow network, but data flow is controlled with a tight feedback loop that limits pipelining, communication, or internal buffering in the encoder.

Secondly, the input process size (granularity) is appro-

priate for system function description but not necessarily for design space exploration and hardware-software implementation. Also, the designer might want to reuse system parts with processes that were optimized to another design and thus must be retargeted.

Process transformation

Before discussing the optimal process size for design space exploration and optimization, let's review some of the process transformation problems and techniques. With process transformation, we can partition and merge processes, which in turn requires communication transformation. Explicit or implicit communications occur in the abstract system specification used as input (the cospecification). As an example, concurrent finite-state machine variables, by definition, are globally accessible and arbitrarily referenced. Referencing variables implies communication, which is inserted when the concurrent finite-state machines are distributed over several components.

Communication transformation. Figure 2 shows the problem of manual communication transformation. Except for simple finite-state machine processes, a process generally contains one or several threads, each consisting of basic blocks with a sequence of statements. They all work on a set of local data. If communication between processes is restricted to explicitly specified process communication (like send or receive statements), then no process can access another process' data except through communication statements. A cosynthesis system can easily determine the required communication actions if the processes are assigned to different components—the communication statements directly map to physical communication actions. At a finer partitioning granularity, dataflow analysis is required to find the required communication.[13-15] Resolving variable array indices becomes important in breaking up processes, since array accesses often occur in loops with optimization potential.[15] Another approach avoids breaking up processes and asks the user to write several versions of a communicating processes system, but this leads to verbose process descriptions and places the burden of process transformation on the designer.

If the individual process contains concurrent elements, then inserting communication statements is generally insufficient. State charts, for example, assume synchronous operation and broadcasting. If state charts are partitioned and communication is inserted, synchrony must be checked and adapted.[16]

Process merging. This high-level transformation problem is mostly postponed to back-end cosynthesis tasks, namely scheduling, high-level synthesis, and software code generation. The reduction in communication overhead

when merging processes is estimated as a preprocessing step to cosynthesis, typically just counting variables to be communicated. This can only be done for a subset of all combinations, such as nearest neighbors in the control or data flow. Also, communication overhead grows with finer partitioning. The overhead declines later by using shared memory and by communicating pointers to blocks of variables rather than communicating all the variables directly;[17] however, the high-level transformation problem remains. To enable maximum memory optimization, partitioning of the local variable set and the larger array variables must complement fine-grain partitioning. This aspect still requires intensive investigation, since it is highly relevant in data signal processor (DSP) loop optimization.

Figure 3 summarizes the granularity dilemma. Approaches at the finest level of granularity (used for ASIPs) do not exploit the full design space.

The bulk of cosynthesis approaches for reactive systems with short process execution times use the input processes without changes,[9,18,19] thus saving the partitioning step. Cosynthesis approaches for systems with higher computation requirements provide most of the contributions to process partitioning. They work mostly on the basic block level;[15,20] other systems select the function level as a compromise, since a function call still exposes the communicated parameters, saving the analysis step.

Because most system designs are not extreme, it is useful to adapt the level of granularity to the application[21] and to the target architecture. So far, little work has been done in granularity adaptation.

High-level transformations are also important to retarget a process (for example, from a RISC to a DSP with a complex memory system). At the current state of research (and probably for some time to come), this transformation step needs designer interaction.

Architecture definition

Many embedded systems consist of a complex, heterogeneous set of standard processors, ASIPs, coprocessors, memories, and peripheral components. The designer typically preselects the architecture to reduce the design space.[10,12] It can also be hard-coded as a cosynthesis template—examples include processor-coprocessors,[9,15,20] VLIW-ASIPs,[22] or single buses with shared memory.[23] There are a few exceptions[24,25] where the search space is controlled by using clustering (for example, according to deadlines or by string encoding in a genetic algorithm).

Design space exploration

At this point, we have defined the architecture, and the input system of processes has been analyzed and adapted. The allocation of processes to components and (complex) vari-

Granularity	Analysis	Optimization potential	Communication overhead	Design effort
Process	No (explicit)			
Function/ global data	Global data flow			
Basic block/ local data set	Global and local data flow			
Statement/ variables	Global and local data flow	↓	↓	↓

Figure 3. Granularity effects.

ables to memory, the mapping of abstract process communication to physical communication, and the scheduling of processes sharing the same resource remain as problems.

Hardware-software partitioning. The architecture determines the cosynthesis approach. The main difference appears between ASIPs (requiring a fine-grain, statement-level approach) and processor-coprocessor systems (working on at least a basic-block level).

ASIPs hold a large market share in markets where the extra design effort is justified. Given an application, the problem of ASIP design is to derive an appropriate processor architecture that can implement application-specific software. For that purpose, one must adapt compilers, libraries, operating system functions (if any), and the simulation and debugging environment. Specialization often leads to irregular register sets and interconnects, which makes compilation and compiler adaptation hard but not infeasible. Most of the work considers hardware design and software generation as separate tasks, just as in quantitative general-purpose processor design.[3]

Approaches combining both tasks are mostly based on standard compilers and try to group three address code statements into complex instructions, thereby adapting the data path[22] and exploring only a part of the overall ASIP design space.[5]

Another ASIP design automation approach with a fixed single-processor template (soft core) uses the instruction word length as a main user-defined parameter to optimize program memory size.[26] In contrast, design space exploration for more complex embedded ASIPs with numerous free parameters, such as multiprocessor DSP architectures, is based on user-driven quantitative exploration.[27]

Standard processors lead to a different set of cosynthesis problems. Here, the software development environments are given, and the application splits into a part implemented on the processor and a part implemented in application-specific hardware (coprocessors). From a tool perspective,
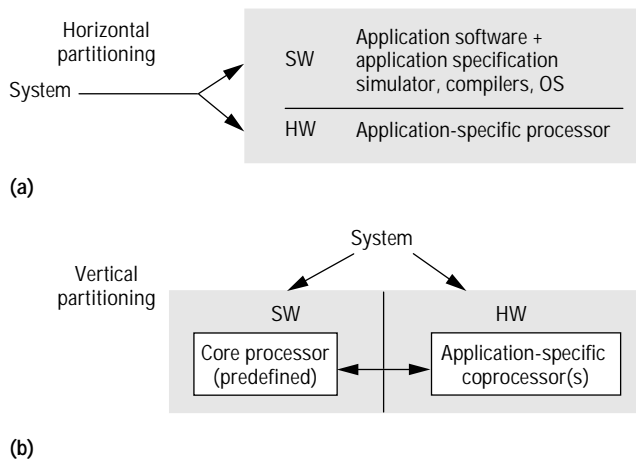
Horizontal partitioning

System

SW — Application software + application specification simulator, compilers, OS

HW — Application-specific processor

(a)

System

Vertical partitioning

SW — Core processor (predefined)

HW — Application-specific coprocessor(s)

(b)

Figure 4. Partitioning types. Application-specific processors (a) and processor and coprocessor (b).

this corresponds to a clear separation in application functions: some are synthesized by hardware synthesis tools and others are handled by software development tools. This was the original meaning of the term hardware-software partitioning. In the case of ASIPs, hardware and software syntheses are applied to the same application functions in sequence, but hardware and software are eventually disjoined. To highlight the differences, we use the terms vertical hardware-software partitioning and horizontal hardware-software partitioning, as shown in Figure 4.

The role of hardware-software partitioning is the optimized distribution of system functions to software and hardware components. There is a general similarity to the scheduling problem in real-time operating systems (RTOSs).[28] Time constraints must be regarded, context switching is required, and we need process communication and synchronization.

However, there are major differences. First, the design space is much larger, since the hardware architecture is not finalized and includes components with vastly different properties that software drivers encapsulate and substitute in RTOSs. In hardware-software partitioning, time constraints can range to less than a microsecond—considerably below typical time constraints in an RTOS, giving high importance to communication and context switch overhead.

Communication synthesis. This step must map communication in the input description to physical communication in the target architecture. Target system hardware and software components can communicate via shared memory, or they can exchange messages. There are many different media: point-to-point, switched channels, buses, or larger networks. The channels can be buffered or non-buffered. There are many protocols, including packet trans-

fer or split transaction. Cost and bandwidth vary widely. In other words, communication design space resembles the component design space in size.

Given the input and output communication models, we can identify three major tasks: communication channel selection, communication channel allocation, and communication channel scheduling.

Currently, no tool can cover the whole variety of communication mechanisms.[25] Communication channel selection is mostly manual or predefined, with few exceptions. Communication channel allocation is mostly treated as a consequence of process allocation. Tools with static non-preemptive process scheduling regularly perform communication scheduling. In these cases, static communication scheduling complements process scheduling to obtain an overall fixed schedule.

Hardware-software scheduling. Scheduling enables hardware and software resource sharing. On the process level, there are several scheduling policies derived from RTOSs,[28] for example, static table-driven, priority-based preemptive scheduling, and various other dynamic plan-based policies that have not yet been applied to codesign.

Priority-based preemptive scheduling is the classic approach of commercial RTOSs. Process priorities are determined a priori (static priority) or at runtime (dynamic priority). They are used in reactive as well as dataflow systems with sufficiently coarse process granularity. Dynamic priority assignment requires a runtime scheduler process that assigns priorities according to process deadlines. This increases component utilization, in particular for reactive systems, but makes timing verification harder.

In static (table-driven) scheduling, the order of process execution is determined at design time. It has been used for periodic process scheduling, where a static schedule exists for the least common multiple (LCM) of all process periods.[29] The process sequence can be stored in a schedule table, but the processes can also be merged into a sequence to use the compiler (or the synthesis tool) to minimize context switching overhead,[30] usually at the cost of a larger program. This is the domain of small processes, where context switching times are significant compared with the process execution times. Static scheduling can also combine with preemptive scheduling. Processes communicating with static data flow triggered by the same event can be clustered and scheduled statically, while the process clusters are scheduled preemptively. This allows for local dataflow optimization, including pipelining and buffering.

In recent years, static scheduling has also been used in event-driven reactive systems. A first approach is to adapt the sequence of executions in a static schedule to the input events and data.[17] A second approach is to collect all parts of a process

activated by the same event in one static thread of operations,[31] which can then be statically scheduled into a single process. Both scheduling approaches can be combined and used as a basis for process merging in event-driven systems.

Complex embedded architectures require distributed scheduling policies for hardware and software parts such as scheduling, which is optimized for several communicating hardware and software components. Communication, especially in context with processing, has drawn little attention in RTOS research or has been treated pessimistically.[12] This treatment is not acceptable for highly integrated embedded systems, where communication and buffering have a major impact on performance, power consumption, and cost. Global approaches to distributed scheduling of communicating processes have been proposed for preemptive[12] and static scheduling.[29]

Instead of a uniform scheduling policy, components or subsystems may use different policies, especially when combining different but compatible system types; but the policies must be compatible. An example is the TOSCA system.[9] It uses static priority scheduling for software implementation of concurrent finite-state machines, while the hardware side does not share resources, thus avoiding hardware scheduling. In the POLIS system,[8,18] software scheduling is even less constrained. A more complicated approach[32] proposes static priority (rate monotonic)[28] software scheduling combined with interrupt-activated, shared-hardware coprocessors in a robot control system. Notably, out of these global policies, only the static uniform approach supports global buffering between components, which is explained by the complex behavior of preemptive scheduling.

Exploiting process semantics, such as mutually exclusive process execution and conditional communication,[32,33] can improve scheduling efficiency. In static scheduling, this knowledge can optimize utilization,[33] while in preemptive scheduling, it can help to verify timing constraints and to optimize communication.

A major problem of static scheduling is data-dependent process execution found not only in software execution, but also in hardware with conditional control flow. Since non-preemptive scheduling must assume worst-case behavior, data-dependent timing leads to lower average system performance. One approach is to resort to dynamic scheduling with local buffers, even in purely static dataflow applications.[34]

The variety of process models of computation and scheduling policies (and their possible combinations) is a challenge to design space exploration and cosynthesis. It requires design representations that allow the mixing and matching of different models of computation for scheduling. Software reuse and object-oriented design imply that critical system parts that the designer knows in detail will combine with less familiar legacy code.

Memory optimization. Memory is becoming a dominant cost factor in integrated systems and often bottlenecks system performance. Allocation of data to memory can, in principle, be combined with hardware-software partitioning and process scheduling.[35] Memory optimization, however, drastically increases the number of design parameters beyond the assignment of data variables to memories and accesses to memory ports. Multidimensional data arrays can be rearranged in memory by index transformations to improve memory use or simplify array index generation.[36] Loop transformations can minimize memory accesses and size.

The optimal access pattern is technology dependent. SDRAM, for example, requires efficient burst access to rows. This influences memory allocation and transformations, especially in cases of memory hierarchies. Past research has examined program cache optimization exploiting the program structure and profiling results to minimize cache misses at the cost of extra main memory space.[37] Other work considers optimization for architectures with memories of different types, such as a combination of scratch-pad SRAM and DRAM.[38] Dynamic memory allocation is another problem that we have only recently addressed in hardware-software codesign.[39]

Estimation. All optimization steps discussed must acknowledge final implementation data, such as the execution time of a process when executed on a specific processor or a coprocessor, and the required sizes of program and data memory. At the time of cosynthesis, these data are not yet available, except for reused system parts. All other hardware and software data must be estimated.

One way to obtain such data is to implement each single process with the target synthesis tool or compiler, and then run them on the (simulated) target platform or use formal analysis.[11] Since efficient synthesis tools and compilers are not available for all target architectures, this approach does not cover the whole design space. In practice, tool license problems could arise, since a company might not want to acquire tools for a large range of possible processors just for the sake of design space exploration.

A second approach is to estimate the results using a simplified but generic version of the synthesis tools, such as a list scheduler or a path-based scheduler, in case the data path architecture isn't completely familiar.[17]

Notably, neither of the two approaches is fully accurate, since both, in general, cannot precisely predict the savings in cost and timing when several processes share the same hardware component,[15] which is the standard case in design. Analyzing all potentially useful combinations of processes and resources, however, is prohibitively time consuming except for critical system parts.

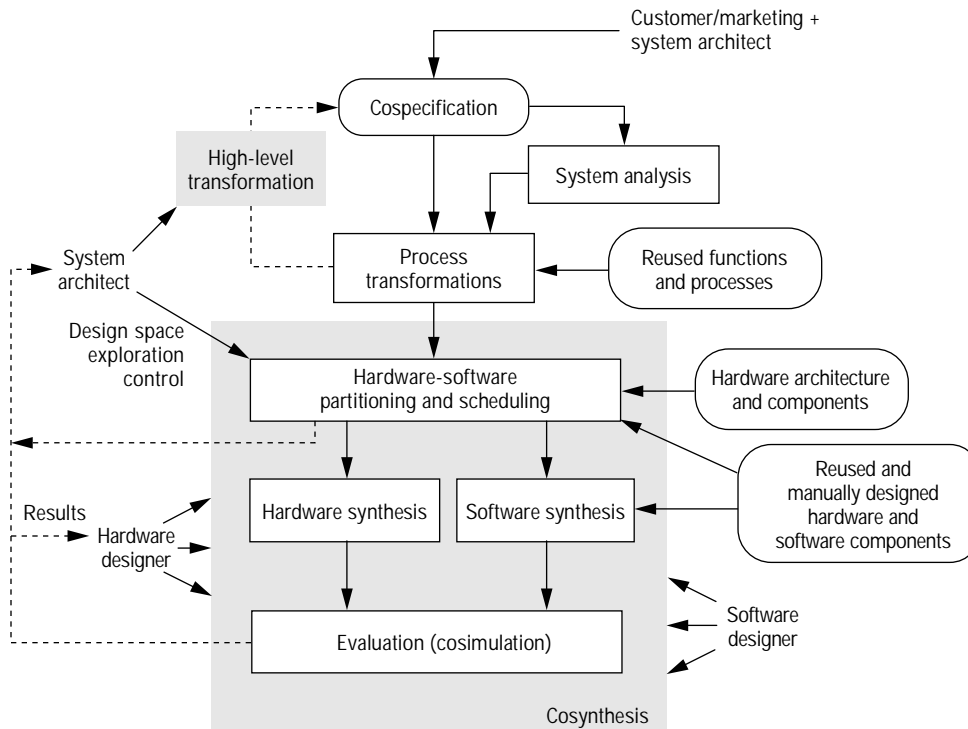Despite these limitations, the current estimation techniques are already much closer to real implementation than

Figure 5. Design space exploration process.

simple back-of-the-envelope estimations.

Exploration and cosynthesis process

Now, we can put the pieces together to propose an integrated computer-aided design space exploration and cosynthesis approach. Figure 5 shows that the system architecture development process of Figure 1 has been detailed into the process transformation, hardware-software partitioning, and scheduling steps. System analysis provides system data derived from the executable cospecification (for example, profiling data). Process transformation prepares the specified system for the exploration process by matching the different parts of the cospecification that may be described in different languages and models of computation.

The system architect can apply high-level transformations to this description to better match the process to the intended memory model. Computer support for high-level transformations will improve over time, but there will always be a difficult-to-automate, creative part. For this reason, we should leave the architecture selection to the system architect and the designers, who can then explore different alternatives, possibly iterating over the high-level transformation process.

In the next phase, the system architect or the design team can control the design space exploration process by changing the selected components (processor types, memories, co-

processors, peripheral units), communication channels, the scheduling policy, or whatever parameters the cosynthesis system supports. Next, the cosynthesis tools partition and schedule the system and provide feedback on the resulting system data. The design space exploration process should be completed by hardware and software synthesis to validate the estimations.

A fully automated hardware and software synthesis process (including interface synthesis) requires flexible and powerful high-level synthesis tools supporting component reuse as well as compilers with excellent code generators. Since both are not readily available for all architectures, manual design support of these backend steps will still be necessary to obtain optimized results in a large design space. This sounds like bad news for an automated design process, but design space exploration does not depend on a complete synthesis of all parts. Instead, it can exploit data for those parts of a design in which efficient compilers and synthesis tools or reused hardware and software components are available, and use estimations for the remaining parts. Cosimulation can then evaluate the system timing, where abstract models can be used for the nonimplemented parts.

This approach has many advantages:

- The design space exploration process is split into two loops with increasing turnaround time and accuracy. The first loop ends with partitioning and scheduling; the second one includes part of the hardware and software synthesis. Estimation precision can be increased by manual interaction. Using both loops, precision can gradually be improved in the course of the design process, increasing result reliability and avoiding unnecessary precision requirements in the early design phases. The short iteration time of the first loop allows iteration over the cospecification parameters. Benner and Ernst[40] have demonstrated how the impact of a specified maximum bus throughput on the optimum architecture of a design could be investigated within a few hours.

- Reuse of components is supported at different levels of abstraction.
- The intermediate results of process transformation and hardware-software partitioning and scheduling can be used for the final design, and therefore, there is little overhead.
- The design process can quickly respond to changes in the specification.
- The design process can profit from an increasing set of intellectual property libraries as well as from progress in hardware synthesis and compiler technology, without a change in the overall methodology.

The many interfaces in Figure 5 give an idea of the work required to integrate the research results into an easy-to-use and extendible industrial design environment. I think it's evident that this effort would be well spent.

COMPUTER-AIDED HARDWARE-SOFTWARE CODESIGN has made considerable progress in the past few years. The greatest demand is currently in system analysis, including cosimulation, coverification, and (executable) cospecification, which is obvious when considering the current design process. Cosynthesis and computer-aided design space exploration are only beginning to reach the industrial design practice. Using a possible design space exploration scenario, we have identified the major problems of this emerging EDA field and have reviewed the results of ongoing research. Highly automated and optimizing cosynthesis approaches have been demonstrated, but for a limited class of architectures and applications. We have, however, seen how the results can contribute to an advanced interactive design space exploration process for a much wider range of architectures. Reuse at different levels of the design process and design migration between different implementation technologies can occur without the need for a complete automation of design steps. ◈D&T

References

1. P. Dreike and J. McCoy, "Cosimulating Hardware and Software in Embedded Systems," *Proc. Embedded Systems Programming Europe*, IEEE Computer Soc. Press, Los Alamitos, Calif., Sep. 1997, pp. 12-27.
2. D. Jaggar et al., "ARM Architecture and Systems," *IEEE Micro*, Vol. 17, No. 4, Jul.-Aug. 1997, pp. 9-11.
3. J.L. Hennessy and D.A. Patterson, *Computer Architecture—A Quantitative Approach*, 2nd ed., Morgan-Kaufmann, San Mateo, Calif., 1996.
4. P. Paulin et al., "Trends in Embedded Systems Technology: An Industrial Perspective," *Hardware-Software Codesign*, G. De Micheli and M. Sami, eds., Kluwer Academic, Boston, Mass., 1996, pp. 311-337.
5. W. Wolf et al., *Hardware-Software Codesign: Principles and Practices*, Kluwer Academic, 1997.
6. D. Herrmann et al., "High-Speed Video Board as a Case Study for Hardware-Software Codesign," *Proc. Int'l Conf. Computer Design*, IEEE CS Press, 1996, pp. 185-190.
7. J. van den Hurk and J. Jess, *System-Level Hardware-Software Codesign*, Kluwer Academic, 1998.
8. M. Chiodo et al., "A Case Study in Computer-Aided Codesign of Embedded Controllers," *J. Design Automation Embedded Systems*, Vol. 1, No. 1, 1996, pp. 51-67.
9. A. Balboni, W. Fornaciari, and D. Sciuto, "Cosynthesis and Cosimulation of Control-Dominated Embedded Systems," *J. Design Automation Embedded Systems*, Vol. 1, No. 3, Jun. 1996, pp. 257-288.
10. D.D. Gajski et al., *Specification and Design of Embedded Systems,* Kluwer Academic, 1994.
11. W. Ye and R. Ernst, "Embedded Program Timing Analysis Based on Path Clustering and Architecture Classification," *Proc. Int'l Conf. Computer-Aided Design,* IEEE CS Press, 1997, pp. 598-604.
12. T.-Y. Yen and W. Wolf, *Hardware-Software Cosynthesis of Distributed Embedded Systems,* Kluwer Academic, 1996.
13. M. Zhou, *Optimization of MPEG-2 Video Encoding*, doctoral dissertation, TU Braunschweig/HHI Berlin, 1997.
14. S. Agrawal and R. Gupta, "Dataflow-Assisted Behavioral Partitioning for Embedded Systems," *Proc. Design Automation Conf.,* ACM, N.Y., 1997, pp. 709-712.
15. R. Ernst, J. Henkel, and T. Benner, "Hardware-Software Cosynthesis for Microcontrollers," *IEEE Design & Test*, Vol. 10, No. 4, Dec. 1993, pp. 64-75.
16. P. Chou and G. Borriello, "An Analysis-Based Approach to Composition of Distributed Embedded Systems," *Proc. Int'l Work. Hardware-Software Codesign,* IEEE CS Press, 1998.
17. R. Ernst et al., "The COSYMA Environment for Hardware-Software Cosynthesis of Small Embedded Systems," *J. Microprocessors and Microsystems*, Vol. 20, No. 3, May 1996, pp. 159-166.
18. M. Chiodo et al., "Hardware-Software Codesign of Embedded Systems," *IEEE Micro*, Vol. 14, No. 4, Jul.-Aug. 1994, pp. 26-36.
19. T.D. Ismail, M. Abid, and A. Jerraya, "COSMOS: A Codesign Approach for Communicating Systems," *Proc. Int'l Work. Hardware-Software Codesign*, IEEE CS Press, 1994, pp. 17-24.
20. R.K. Gupta and G. De Micheli, "A Cosynthesis Approach to Embedded System Design Automation," *J. Design Automation Embedded Systems*, Vol. 1, No. 1, Jan. 1996, pp.

69-120.

21. J. Henkel and R. Ernst, "A Hardware-Software Partitioner Using a Dynamically Determined Granularity," *Proc. Design Automation Conf.,* ACM, 1997, pp. 691-696.

22. J. Wilberg and R. Camposano, "VLIW Processor Codesign for Video Processing," *J. Design Automation Embedded Systems*, Vol. 1, No. 1, 1996, pp. 79-119.

23. J. Axelson, *Analyis and Synthesis of Heterogeneous Real-Time Systems,* doctoral thesis, Dept. of Computers and Info. Studies, Linköping Studies in Sci. and Tech., 1997.

24. B.P. Dave and N. Jha, "CASPER: Concurrent Hardware-Software Cosynthesis of Hard Real-Time Aperiodic and Periodic Specifications of System Architectures," *Proc. DATE*, IEEE CS Press, 1998, pp. 118-124.

25. J. Teich, T. Blickle, and L. Thiele, "An Evolutionary Approach to System-Level Synthesis," *Proc. Int'l Work. Hardware-Software Codesign*, IEEE CS Press, 1997, pp. 167-172.

26. B. Shakleford et al., "Memory-CPU Size Optimization for Embedded System Design," *Proc. DATE*, IEEE CS Press, 1997, pp. 246-251.

27. P. Lieverse et al., "A Clustering Approach to Explore Grain Sizes in the Definition of Weakly Programmable Processing Elements," *Proc. Work. Signal Processing Systems*, IEEE CS Press, 1997, pp. 107-120.

28. K. Ramamritham and J. A. Stankovic, "Scheduling Algorithms and Operating Systems Support for Real-Time Systems," *Proc. IEEE*, Piscataway, N.J., Jan. 1994, pp. 55-67.

29. E.A. Lee and D.G. Messerschmitt, "Synchronous Data Flow," *Proc. IEEE*, Sep. 1987, pp. 1235-1245.

30. A. Österling, T. Benner, and R. Ernst, "Code Generation and Context Switching for Static Scheduling of Mixed Control and Data Oriented HW/SW Systems," *Proc. Asia Pacific Conf. Hardware Description Languages*, Tsing Hua Univ., 1997, pp. 131-135.

31. J. Li and R.K. Gupta, "HDL Optimization Using Timed Decision Tables," *Proc. Design Automation Conf.*, IEEE CS Press, 1996, pp. 51-54.

32. V.J. Mooney and G. DeMicheli, "Real-Time Analysis and Priority Scheduler Generation for Hardware-Software Systems with a Synthesized Run-Time System," *Proc. Int'l Work. Hardware-Software Codesign*, IEEE CS Press, 1997, pp. 605-612.

33. P. Eles et al., "Scheduling of Conditional Process Graphs for the Synthesis of Embedded Systems," *Proc. DATE*, IEEE CS Press, 1998, pp. 132-138.

34. J.A. Leijten et al., "Stream Communication between Real-Time Tasks in a High-Performance Multiprocessor," *Proc. DATE*, IEEE CS Press, 1998, pp. 125-131.

35. E. De Greef, F. Catthoor, and H. De Man, "Memory Organization for Video Algorithms in Programmable Signal Processors," *Proc. Int'l Conf. Computer Design*, IEEE CS Press, 1995, pp. 552-557.

36. H. Schmit and D. Thomas, "Address Generation for Memories Containing Multiple Arrays," *Proc. Int'l Conf. Computer-Aided Design*, IEEE CS Press, 1995, pp. 510-514.

37. H. Tomiyama and H. Yasuura, "Optimal Code Placement of Embedded Software for Instruction Caches," *Proc. ED&TC*, IEEE CS Press, 1996, pp. 96-101.

38. P.R. Panda, N.D. Dutt, and A. Nicolau, "Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications," *Proc. ED&TC*, IEEE CS Press, 1997, pp. 7-11.

39. G. de Jong et al., "Background Memory Management for Dynamic Data Structure Intensive Processing Systems," *Proc. Int'l Conf. Computer-Aided Design*, IEEE CS Press, 1995, pp. 515-520.

40. T. Benner and R. Ernst, "An Approach to Mixed System Cosynthesis," *Proc. Int'l Work. Hardware-Software Codesign*, IEEE CS Press, 1997, pp. 9-14.

Rolf Ernst is a professor at the Institute of Computer Engineering at the Technical University of Braunschweig, Germany. His current interests are in hardware-software codesign, high-level synthesis, rapid prototyping, and embedded-system architecture.

Address questions or comments to Rolf Ernst, Institut fuer Datenverarbeitungsanlagen, Technische Universitaet Braunschweig, Hans-Sommer-Str. 66, D-38106 Braunschweig, Germany; ernst@ida.ing.tu-bs.de.