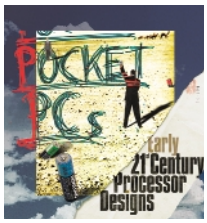
Embedded Computer Architecture and Automation



The distinct requirements of embedded computing, coupled with emerging technologies, will stimulate system and processor specialization, customization, and computer architecture automation.

B. Ramakrishna Rau
Michael S. Schlansker
Hewlett-Packard Laboratories

With the advent of system level integration (SLI)—the next level of integration beyond VLSI—and system-on-chip (SOC) capabilities, the computer industry's focus is shifting from personal to embedded computing. The opportunities, needs, and constraints of this emerging trend will lead to significantly different computer architectures at both the system and processor levels as well as a rich diversity of off-the-shelf (OTS) and custom designs. Embedded computing will also stimulate automation of computer architecture, which we illustrate using an architecture synthesis system called PICO—program in, chip out—that we developed with our colleagues during the past five years.

EMBEDDED COMPUTING

Driven by the accelerated pace of semiconductor integration during the past three decades, the computer industry has steadily moved from mainframes and minicomputers to workstations and PCs. In accordance with a corollary of Moore's law, computing power becomes half as expensive every 18 to 24 months. Over a decade, this reduces the cost by a factor of 30 to 100, making computing affordable to an exponentially larger number of users and dramatically changing the key applications of this computing power.

Manufacturers have for several years incorporated embedded computers in so-called smart products such as video games, DVD players, televisions, printers, scanners, cellular phones, and robotic vacuum cleaners. Using embedded computers in devices that previously relied on analog circuitry—such as digital cameras, digital camcorders, digital personal recorders, Internet radios, and Internet telephones—provides revolutionary performance and functionality that merely improving analog designs could not achieve. The increasing availability of SLI heralds a vast array of even more innovative smart products.

Any computer architecture must balance the latest technological opportunities with product, market, and application requirements that together determine three important features of embedded computing architecture: specialization, customization, and automation. Specialization increases the performance and reduces the manufacturing cost of embedded computer systems. Customization permits specialization when no adequately specialized OTS product is available. Automation reduces the design costs incurred by customization.

Product requirements

Smart products demand various combinations of high performance, low cost, and low power. When budgets allow only a few cents for critical chips, cost can be more important than performance. In other cases, the smart product's functionality mandates high performance. For example, data-intensive tasks such as image, video, and signal processing require throughput that significantly exceeds that of high-end workstations.

The challenge now is to lower cost to a level that the market will accept. In certain instances, providing more computing power can reduce cost. Many imaging and video products require large amounts of DRAM to hold their data sets. Using compression techniques such as MPEG, JPEG, and MP3 for large video, image, and audio data sets, respectively, reduces the amount of DRAM required but introduces a sig-

0018-9162/01/$10.00 © 2001 IEEE

Specialization minimizes the fabrication cost of an embedded computer system or processor.

nificant amount of additional computation. The reduction in memory cost more than pays for the extra processing power.

Power is of great concern in many smart products. It is obviously important in mobile smart products in which battery life is a primary factor, but it can also be important in office and computer-room products in which electrical operating and cooling costs are increasingly significant.

Market requirements

The successful design and fabrication of an embedded computer gives a new smart product its distinguishing high-value features and competitive advantage. Time to market has a disproportionate impact upon a product's life-cycle profits and can determine its success or failure. Because the consumer market typically emphasizes short product life cycles, designing rapidly is critical.

A second important market requirement is the need to support a sharp increase in the number and complexity of embedded system designs. The current scarcity of talented designers threatens to cause a design bottleneck as the need for smart product designs increases. The lack of sufficient design talent may result in an inability to design otherwise profitable products.

Application characteristics

As the cost of their electronic content decreases, single-function products with complex, high-performance embedded systems are becoming increasingly popular. Such embedded applications have relatively small and well-defined workloads. Key kernels within applications often represent the vast majority of required compute cycles. For example, in video, image, and digital signal processing, small loop nests with a high degree of parallelism often dominate the applications' execution times. These applications typically have more determinacy than general-purpose applications. Not only is the nature of the application fixed, but physical product parameters such as imaging sensor size predetermine loop trip counts, array sizes, and other key parameters.

SPECIALIZATION

Developers design general-purpose embedded systems as OTS parts for reuse in numerous smart products. Because their specific applications are unknown, these designs must incorporate both generality and completeness. Specialization involves departures from both characteristics with respect to structure and functionality. Extreme specialization results in an application-specific system or a processor designed for a single application. A domain-specific system or processor, in contrast, has been specialized for an application domain—for example, HDTV or set-top boxes—but not for a specific application.

At one end of the spectrum, application-specific architectures provide very high performance and high cost performance, but reduced flexibility. At the other end, general-purpose architectures provide much lower performance and cost performance, but with the flexibility and simplicity associated with software programming. For a desired performance level on a specified workload, specialization minimizes the logic complexity and die size—and thus the fabrication cost—of an embedded computer system or processor.

Limits of general-purpose systems

The widespread use of special-purpose architectures greatly increases the need for new designs. General-purpose systems typically consist of a conventional bus-based architecture with one or more reduced instruction set computer (RISC), complex instruction set computer (CISC), or very long instruction word (VLIW) processors.

While a single processor is feasible at lower performance levels, meeting the demanding requirements of high-performance embedded systems requires multiple processors. Because developers must design general-purpose multiprocessors without knowledge of the specific application, they must provide uniform communication among symmetrically connected identical processors. Such architectures scale poorly; a large-scale multiprocessor interconnection network requires a lot of area and long transmission delays that adversely affect cost and cycle time.

System specialization

The need therefore arises for tailoring every aspect of a high-performance system to meet specific demands.

- The system topology must be sparse and irregular, reflecting application requirements.
- Processors must be heterogeneous and specialized to their assigned tasks.
- Multiple low-cost, decentralized, low-bandwidth RAMs must replace a single, expensive, centralized high-bandwidth memory system.

Especially for compute-intensive applications, designers often can use *synchronous parallelism* to specialize and simplify synchronization among multiple processors. This lockstep parallelism relies on knowledge of the relative progress of interacting parallel processes. A compiler can accurately track progress at compile time, and a hardware designer or CAD tool can do so during hardware design. Incorporating correct synchronization into the hard-

ware or software design eliminates the need for explicit runtime synchronization among synchronously operating processors.

Even when complete synchronicity is impossible, developers can specialize the hardware to exploit knowledge of the communication pattern. For example, if a data set's production and consumption exhibit a consistent order and rate, a simple, small FIFO buffer capable of stalling the producer or consumer can replace a more general RAM structure with software-implemented synchronization.

General-purpose processor limits

High-performance general-purpose processors provide symmetric access between function units (FUs)—which can execute commonly needed operations—and storage. This flexibility, however, comes at the cost of expensive and slow, multiported registers and RAMs. Also, control for general-purpose processors is instruction based and, for multiple-issue processors, requires wide instruction-memory access and complex, expensive shifting and alignment networks. RISC, CISC, DSP (digital signal processor), and even VLIW architectures consequently do not scale efficiently beyond relatively narrow architectural limits.

Processor specialization

Specialization maximizes cost performance by minimizing the number of opcodes, the number and width of data paths, the number and width of registers, and the size and width of caches. Data path topology specialization becomes crucial at high-performance levels of tens or hundreds of operations per cycle.

Data path sparseness and asymmetry. Embedded processors must incorporate instruction-level parallelism (ILP) as multiple FUs in each processor. Because a complete and symmetric interconnection between FUs and register files would be prohibitively expensive, the processor's interconnect topology must be sparse and irregular to reflect application requirements. Data paths specifically designed for an application connect arithmetic units and registers. Specialized arithmetic, logical, and data-transfer FUs are used, often chained into sequences of FUs to squeeze out additional circuitry.

Control strategies. Processor specialization can also reduce control cost. Instruction-based control permits control signals to be specified on a cycle-by-cycle basis. If certain control signals change relatively infrequently, specifying them from a residual control register with contents set either by an instruction or during hardware initialization can conserve code size. Fixing certain control signals during hardware optimization can eliminate part of the data path. Instruction-based control is possible at various levels of generality and flexibility, including a full-fledged instruction unit operating out of an instruction cache or main memory, a microprogrammed controller using a microprogram RAM or ROM, or a hardwired controller implemented as a finite state machine.

In principle, these data path and control strategies can exist in any combination. Using a symmetric and homogenous data path with a full-fledged instruction unit satisfies the need for ease of programming and a quality compiler. However, the entire set of combinations is attractive when designing nonprogrammable accelerators—hardware that can perform just one function or one of a fixed set of functions.

Customization enables better cost performance than an OTS design but necessitates a complicated design process.

CUSTOMIZATION

Customization refers to a level of specialization beyond OTS availability that a smart-product vendor undertakes to meet its product's specific needs. As far as time to market, engineering effort, and project risk are concerned, an OTS design is preferable whenever possible. Nevertheless, smart-product vendors routinely design custom systems, processors, and accelerators when three conditions jointly occur:

- The smart product has challenging requirements that can only be met by specializing the system or processor architecture.
- The benefits—in product cost, performance, power, or usability—of specialization beyond those of existing OTS designs are so great that they justify the development cost in dollars, time to market, and risk.
- It has not been economically attractive for a semiconductor vendor to previously develop an OTS design adequately specialized to the smart product's needs. This often occurs because the product has proprietary algorithms or a low anticipated unit volume or because the diversity of special-purpose solutions that smart-product vendors collectively demand inhibits individual OTS solutions.

Customization enables even better cost performance but necessitates a complicated design process, thereby increasing the product's nonrecurring expenses (NRE). Smart-product vendors must amortize this increased NRE over the product volume, which adds to product cost.

Customization strategies

Customization incurs three types of NRE design costs. Architectural design costs include designing the custom system and any custom processors, hardware-software partitioning, logic synthesis, design verification, and postfabrication system integration. Physical

Automation reduces design costs, thereby making low-volume customized products viable.

design costs include floor planning, placement, and routing. Designers must also contend with the substantial costs associated with creating the one or more mask sets needed to complete a design.

Reducing architectural design costs. One way to reduce architectural design costs is to reuse preexisting soft IP or virtual components for the constituent subsystems. When appropriate soft IP is unavailable, incurring the architectural design cost for that subsystem is unavoidable. For example, the system architecture may consist of a specific interconnect topology containing one or more OTS microprocessors and one or more accelerators.

Since defined by the application, the accelerators must be custom designs. Only the architectural design cost of these accelerators is incurred. This strategy can be applied one level down to the processors. Most of a processor's architecture can be kept fixed, but certain FUs, for example, can be customized.[1]

Reducing physical design costs. Every customized part of an SOC imposes placement and routing costs. However, using hard IP—subsystems taken through physical design—can minimize this cost. Although performing the floor planning, placement, and routing for the entire SOC is necessary, treating the hard IP blocks as atomic components greatly reduces these steps' complexity.

Avoiding mask set costs. An SOC can greatly reduce the expense of a complex embedded system, but customizing even part of a chip requires creating a new mask set for the entire SOC, which costs hundreds of thousands of dollars. This provides a powerful incentive for completely avoiding SLI design in favor of OTS, customizable SOCs containing reconfigurable hardware. Rather than using standard cells to implement the custom accelerators and FUs, the strategy calls for performing logic synthesis and mapping them onto field-programmable gate arrays.[2]

FPGAs, along with their associated configurable RAM blocks, allow custom, special-purpose processors or FUs to be implemented as programmable logic. FPGA libraries now also provide general-purpose processor cores programmed as FPGA logic that can be modified or enhanced for specific application needs. Other functionality, such as support for peripheral interfaces and programmable I/Os, further increases FPGA utility.

The OTS, customizable SOC contains the fixed portions of the system architecture—including general-purpose processors, DSP processors, certain accelerators, memory, and peripheral interfaces—implemented using full custom or standard cell logic, but provides FPGAs to implement the customized subsystems. This strategy eliminates the expensive, risky, and slow chip-design process.

By programming the FPGAs appropriately, designers can customize the OTS SOC to implement many different system architectures. This approach incurs the costs of the custom subsystems' architectural design as well as programming the FPGAs. The inherent hardware costs for supporting programmable logic make FPGAs an order of magnitude worse than equivalent custom, standard cell designs in both silicon area and cycle time. Nevertheless, FPGA-based designs for high-performance accelerators are more cost-effective than designs relying solely on general-purpose processors.

If the desired custom system does not fit the OTS, customizable SOC's system-level architecture, the smart-product vendor must design a custom SOC. For high-volume products with well-understood application needs that can tolerate high design cost and time, custom SOCs provide higher performance at a lower cost than OTS SOCs.

AUTOMATION

The architectural design costs of implementing either custom SOCs or OTS, customizable SOCs often dominate NRE costs. Along with the need for mass customization, this has led to the automation of computer architecture. At least three circumstances argue for automation:

- *Not enough designers exist to meet the explosive demand for unique embedded designs.* Automation addresses this problem—which shrinking product life cycles exacerbate—by increasing design productivity.
- *Time to market or time to prototype is crucial.* If developers can't anticipate product requirements early enough to permit traditional manual design—perhaps because no relevant standard exists or because they do not yet understand product functionality—the speed of automated design is extremely valuable.
- *The expected volume of custom-designed products is too small for manual design to be economical.* Automation reduces design costs, thereby making low-volume customized products viable.

We call this automation *architecture synthesis* to distinguish it from behavioral synthesis[3,4] and logic synthesis.[5]

Automation philosophy

Skeptics commonly argue that automating computer system design is unrealistic because it would require emulating human designers, who typically invent new solutions to the design problems they

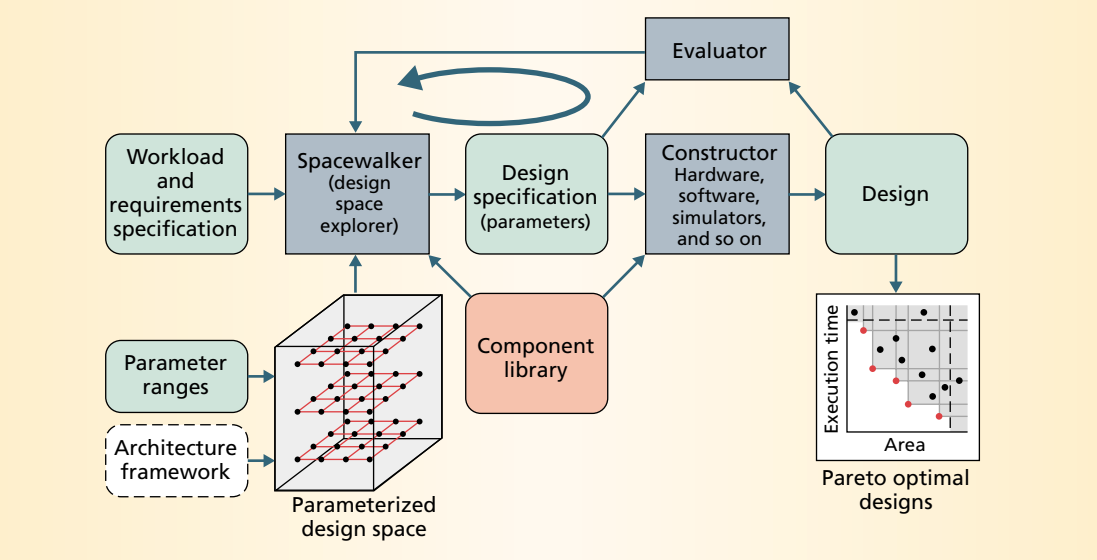


Figure 1. Program in, chip out (PICO) automation philosophy. PICO operates within a parameterized design framework. To define a parameterized design space, the user specifies a set of values that each parameter can assume. The spacewalker explores this space to find good designs for the specified workload and requirements by repeatedly binding the values of the parameters. A constructor can build any design within the framework from a corresponding component library.

encounter. Rather than attempting to emulate a human's ability to invent, the approach we developed within the PICO project, described in the "PICO Architecture Synthesis System" sidebar, selects the design that most closely matches the application's needs out of a large, denumerable design space. This space must be large enough and diverse enough to ensure a sufficient repertoire of good designs so that a best design, selected from this space, will closely match application needs.

Framework and parameters. Because explicitly enumerating every feasible design is impractical, a framework of rules and constraints defines the design space. The framework predetermines some aspects of the design, such as the presence of certain modules and how they are connected, while leaving others unspecified. Of these, some aspects can be derived algorithmically once the rest have been specified. The latter, termed parameters, provide a terse and abstract way of uniquely specifying a design. A design specification consists of a binding of values to parameters. Construction is the process of algorithmically deriving the remaining design details from the specified parameters.

Components. Designs are assemblies of lower-level components selected from a library. In the case of parameterized components, the constructor uses the specified parameters to instantiate a detailed component design. The components fit into the framework and collectively provide all the building blocks needed to construct any design within the framework. Each component also has associated information necessary to interface components into a system design context. Information includes descriptions of a component's functionality, inputs and outputs, and externally visible timing and resource needs. Components themselves can be a network of lower-level components forming a design hierarchy.

An automation paradigm. The best design optimizes multiple evaluation metrics such as cost, performance, and power. A design is *Pareto optimal* if no other design is at least as good with respect to every evaluation metric and better than it with respect to at least one evaluation metric. The Pareto set includes all the Pareto-optimal designs.

As Figure 1 shows, PICO uses three interacting modules to automatically find a Pareto set.

- A *spacewalker* searches for Pareto-optimal designs in the design space that is the Cartesian product of the sets of values for the various parameters. At each step in the search, the spacewalker specifies a design by binding the parameters.
- A *constructor* constructs a hardware realization of the design specification as an assembly of components from the library.
- An *evaluator* determines the suitability of the spacewalker's design choice.

Accurate evaluation involves first executing the constructor to produce a detailed design, which the eval-

PICO Architecture Synthesis System

Our research prototype of an architecture synthesis system, PICO (program in, chip out), automatically designs custom embedded computer systems. For an application written in C, it automatically architects a set of Pareto-optimal system designs and emits the structural VHDL for the hardware components as well as the compiled software code. Two metrics—cost as measured by gate count or chip area and performance as measured by execution time—currently define optimality.

At the system level, PICO identifies the Pareto-optimal set of custom system designs for a given application. Within PICO's framework, shown in Figure A, a system design consists of a custom EPIC (explicitly parallel instruction computing) or VLIW (very long instruction word) processor,[1] a custom cache hierarchy and, optionally, one or more custom nonprogrammable accelerators (NPAs). PICO exploits this hierarchical structure by decomposing the system design space into smaller individual spaces, one for each subsystem. The system-level spacewalker invokes subsystem-level spacewalkers to get the Pareto-optimal sets of subsystem designs and then composes Pareto-optimal systems.[2] Figure B illustrates the PICO design flow.

In the process, PICO does hardware-software codesign, partitioning the application between custom NPAs and software on the custom VLIW processor. For each loop nest in the application that is a candidate for hardware implementation, the spacewalker executes the loop nest either on the VLIW processor or on a custom NPA. Because the spacewalker retains only the Pareto-optimal choices, the system designs that the system-level spacewalker selects represent the most effective hardware-software partition at a given level of cost or performance.

PICO-VLIW, the PICO subsystem that designs custom, application-specific VLIW processors,[3] also retargets Elcor, PICO's VLIW compiler, to each new processor so that it can compile the C application to that processor.[4] The processors currently included within PICO-VLIW's framework encompass a broad class of VLIW processors with a number of sophisticated architectural and microarchitectural features.[1,5] Typically, the design space includes numerous VLIW designs. Furthermore, evaluating the performance of each VLIW design is time-consuming because it involves compiling a potentially large application. The spacewalker therefore uses heuristic search strategies to prune the design space based on previously evaluated processor designs. Examples of similar work are the MOVE project[6] at the Technical University of Delft and the custom-fit processor work[7] at Hewlett-Packard Laboratories.

The PICO-N subsystem designs NPAs for a given loop nest.[8] The NPA framework design consists of a synchronous, one- or two-dimensional array of customized processing elements (PEs) with local memories and interfaces to global memory, a controller for the PEs, and a host processor interface. Each PE forms a data path with a distributed register file structure implemented using FIFOs with random-read access.

Interconnections between the FIFOs and FUs exist only where needed, resulting in a sparse and irregular interconnect structure. PICO-N transforms a sequential C repre-

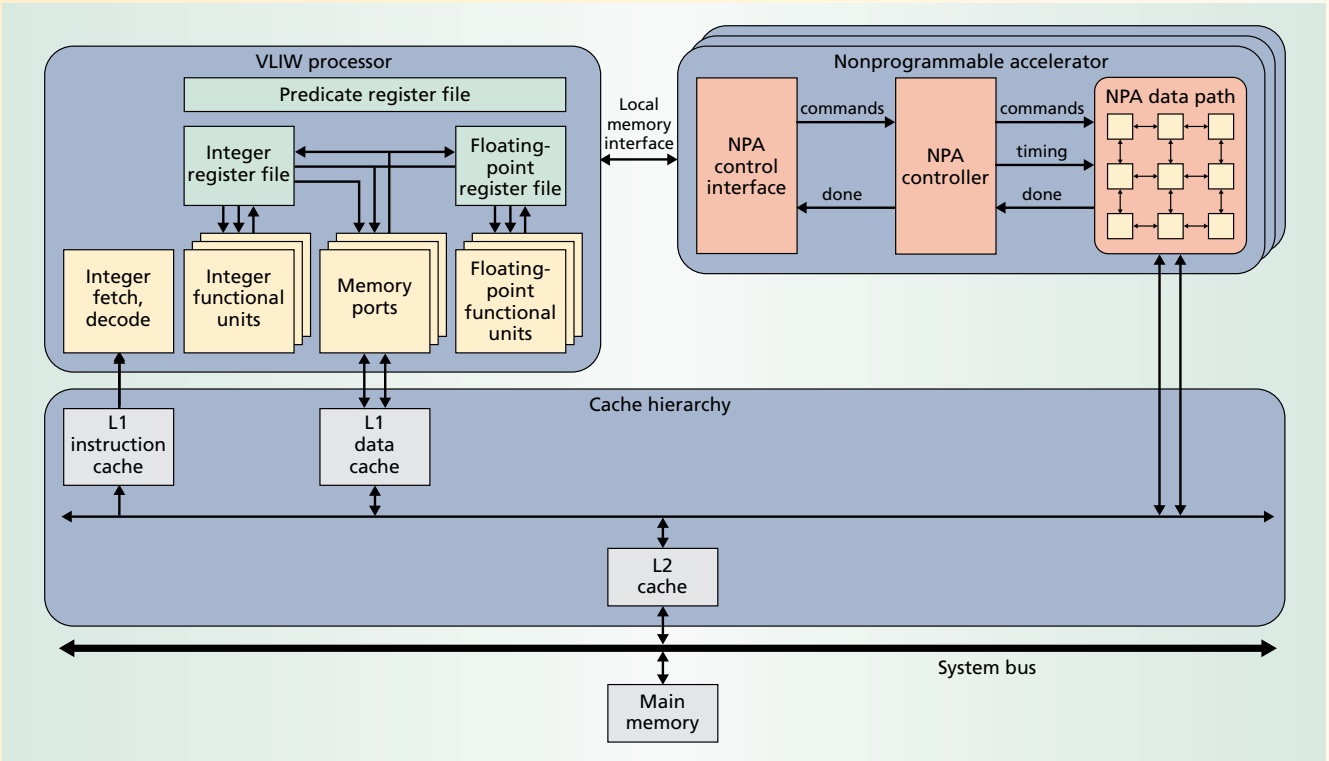


Figure A. Program in, chip out (PICO) framework. A VLIW processor is connected to a cache hierarchy consisting of an instruction cache, a data cache, and a unified second-level cache that, in turn, connects to the system bus. The control interface of each nonprogrammable accelerator (NPA) is mapped into a local memory address space; the NPA's data interface connects to the second-level cache. The processor contains an integer cluster and optional floating-point cluster, each consisting of a register file and set of functional units (FUs), and memory ports connected to one or both register files. Each NPA contains an array of identical processing elements—a network of FUs, register FIFOs, and interconnected memories—that operate in a synchronously parallel fashion.

sentation of the loop nest into multiple sequential invocations of multiple identical, synchronous parallel computations working out of a distributed address space. It synthesizes a PE for each of these parallel computations as well as the VLIW interface code required to use the NPA.

The third major PICO subsystem automatically generates the Pareto sets for cache hierarchies customized to the given application.[9] A design within the cache hierarchy framework consists of a first-level data cache, first-level instruction cache, and second-level unified cache. As at the system level, PICO decomposes the cache hierarchy design space into smaller spaces for the data cache, instruction cache, and unified cache, respectively, and employs a further level of hierarchical spacewalking.

References

1. M.S. Schlansker and B.R. Rau, "EPIC: Explicitly Parallel Instruction Computing," *Computer*, Feb. 2000, pp. 37-45.
2. S.G. Abraham and B.R. Rau, "Efficient Design Space Exploration in PICO," *Proc. Int'l Conf. Compilers, Architecture Synthesis for Embedded Systems* (CASES 2000), ACM Press, New York, 2000, pp. 71-79.
3. S. Aditya, B.R. Rau, and V. Kathail, "Automatic Architectural Synthesis of VLIW and EPIC Processors," *Proc. Int'l Symp. System Synthesis* (ISSS 99), IEEE CS Press, Los Alamitos, Calif., 1999, pp. 107-113.
4. B.R. Rau, V. Kathail, and S. Aditya, "Machine-Description Driven Compilers for EPIC and VLIW Processors," *Design Automation for Embedded Systems*, vol. 4, no. 2/3, 1999, pp. 71-118.
5. V. Kathail, M. Schlansker, and B.R. Rau, *HPL-PD Architecture Specification: Version 1.1*, tech. report HPL-93-80 (R.1), Hewlett-Packard Laboratories, Palo Alto, Calif., Feb. 2000.
6. H. Corporaal, *Microprocessor Architectures: From VLIW to TTA*, John Wiley & Sons, Chichester, England, 1997.
7. J.A. Fisher, P. Faraboschi, and G. Desoli, "Custom-fit Processors: Letting Applications Define Architectures," *Proc. 29th Ann. Int'l Symp. Microarchitecture* (MICRO-29 1996), IEEE CS Press, Los Alamitos, Calif., 1996, pp. 324-335.
8. R. Schreiber et al., "High-level Synthesis of Nonprogrammable Hardware Accelerators," *Proc. Int'l Conf. Application-Specific Systems, Architectures, and Processors* (ASAP 2000), IEEE CS Press, Los Alamitos, Calif., 2000, pp. 113-124.
9. S.G. Abraham and S.A. Mahlke, "Automatic and Efficient Evaluation of Memory Hierarchies for Embedded Systems," *Proc. 32nd Ann. Int'l Symp. Microarchitecture* (MICRO-32 2000), IEEE CS Press, Los Alamitos, Calif., 1999, pp. 114-125.

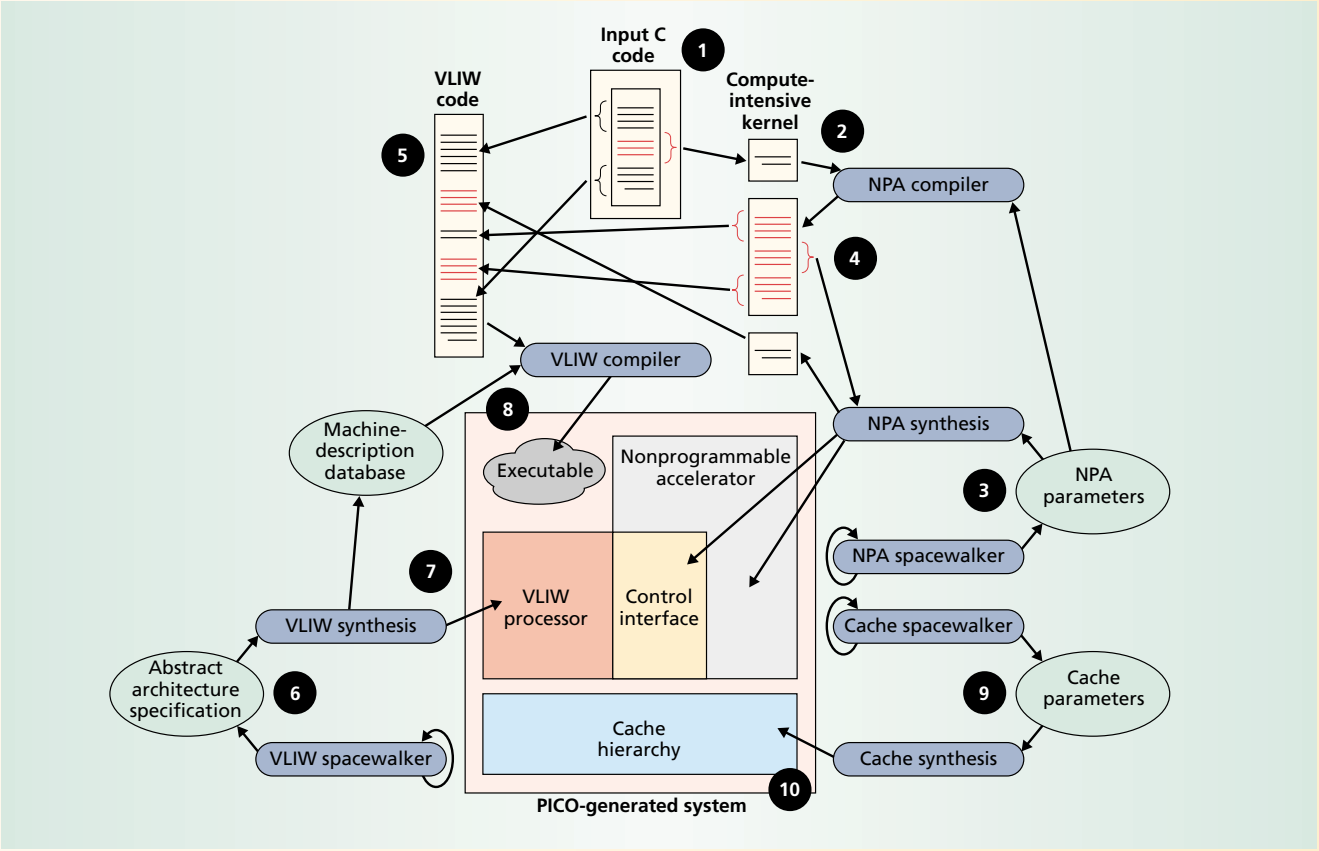


Figure B. PICO design flow. (1) The input C application contains compute-intensive loop nests or kernels, (2) each of which PICO identifies and extracts. (3) The PICO-N spacewalker specifies an NPA. (4) PICO-N transforms the kernel to have the requisite level of parallelism, introduces local memories to minimize main memory bandwidth, and generates the register-transfer level (RTL) for an NPA along with the needed code. (5) For each kernel to be implemented as an NPA, the synthesized code replaces the original kernel code in the application. (6) The PICO-VLIW spacewalker specifies a VLIW processor. (7) PICO-VLIW designs the VLIW processor's architecture and microarchitecture, emits an RTL design, and generates a machine-description database that describes this processor to Elcor, the VLIW compiler. (8) Elcor is automatically retargeted to the newly designed VLIW processor and compiles the modified application. (9) The cache spacewalker specifies and evaluates cache hierarchy configurations. (10) The system-level PICO spacewalker composes compatible, Pareto-optimal VLIW, cache, and NPA designs into valid system designs and determines which kernels to implement as NPAs rather than as software on the VLIW.

The functionality of innovative smart products relies on the availability of extremely high performance, low-cost embedded computer systems.

uator then uses to compute the evaluation metrics. When the cost of constructing candidate detailed designs proves excessive, the evaluator can instead estimate the evaluation metrics approximately, without construction, directly from the design parameters. Evaluators use multiple tools including compilers, simulators, and gate-count estimators.

At each step in the search, the spacewalker invokes constructors and evaluators to determine whether the latest design is Pareto optimal. If the design space is small, the spacewalker can use exhaustive search. Otherwise, it uses the evaluation metrics, other design statistics, and heuristics to guide its search. In this case, the spacewalker seeks to find many Pareto-optimal designs while examining just a small fraction of the design space. Spacewalking is hierarchical if a spacewalking search designs an optimized system and also uses lower-level spacewalking searches to optimize the system components as subsystems.

A framework restricts designs to a subset of all possible designs, but this accounts for the framework's power; it is not possible to create constructors and evaluators capable of treating all possible designs. The challenge is to choose a framework large and diverse enough to contain a sufficient repertoire of good designs. Designs within the framework need not be optimal—to be useful, automation need only produce designs that are competitive with manually produced designs.

Technologies for architecture synthesis

Automatic architecture synthesis relies on both compiler and computer-aided design technologies. Compilers analyze and transform software, providing a framework for processing programs with complex branching and sequencing, while CAD technology analyzes, optimizes, and transforms hardware.

Compilers. Nonprogrammable accelerator synthesis uses compiler technology to process an input loop nest specified as a software program. The compiler analyzes the loop nest to obtain critical program optimization information. Data flow, control flow, and memory dependence analyses provide information that allows the compiler to expose available parallelism and to optimize and transform the loop nest.

Compilers use high-level transformations such as function inlining, loop interchange, and loop unrolling to increase program parallelism. Traditional scalar optimizations—including common subexpression elimination, dead code elimination, strength reduction, and loop-invariant code removal—improve input code quality after the application of transformations that necessitate further optimizations.

Compilers also let programs written with unlimited virtual resources execute with limited physical resources. For example, the compiler transforms code that references an unlimited number of virtual registers into code that references as few physical registers as possible, facilitating the generation of inexpensive hardware by the hardware synthesis step.

Software programs reference a single shared memory that holds all the data structures. Compilers can analyze programs to recognize noninteracting data structures that separate local memories can hold. Using such memories requires changing the address computation code as well as the code that initializes local memories or retrieves final values from them. After the compiler identifies potential local memory use, hardware synthesis determines how many physical memories to use. When advantageous, multiple data structures can exist within a single local memory.

CAD. The architecture synthesis phase that generates hardware performs a function similar to high-level synthesis, drawing upon both back-end compiler and behavioral synthesis technologies. Back-end compilers schedule code and allocate registers in the context of arbitrary program structure. However, the technology has traditionally focused on developing the best code for a given processor. Because the hardware in this case is unspecified, jointly performing these allocation decisions minimizes the required FU and interconnect hardware, which is precisely the object of behavioral synthesis.

Although behavioral synthesis packages have adopted techniques originally developed for ILP compilers, such as software-pipeline scheduling, the class of computations they can deal with and the code optimizations they can perform are relatively limited. This architecture synthesis phase is a fertile research area in which new techniques will enhance the applicability and quality of automatically produced embedded architectures.

Architecture synthesis is complementary to, and relies upon, the lower-level CAD steps of logic synthesis, circuit synthesis, and physical design. After architecture synthesis produces a register-transfer-level (RTL) design, commercially available CAD packages translate the design to the logic and circuit levels, further optimize it at each of these levels, and transform it into a form suitable for creating the mask sets. These lower-level steps are largely independent of architecture synthesis. However, architectural decisions can interact with important lower-level design properties such as the ability to meet timing specifications. Whether and how architecture synthesis should create RTL design to improve the success of these lower-level CAD steps remains a topic for further investigation.

The functionality of innovative smart products relies on the availability of extremely high performance, low-cost embedded computer systems. Designing successful computer architectures requires

carefully balancing the opportunities the latest technologies offer with market, product, and application requirements. We believe that the distinctive requirements of embedded computing will lead to a renaissance in system and processor architecture, which will be far more special-purpose, heterogeneous, and irregular. The need for custom and customizable architectures will also lead to increased automation that our experience with PICO indicates is both practical and effective. The resulting designs are competitive with their manual counterparts but are obtained one to two orders of magnitude faster. ✸

Acknowledgments

Our present and past colleagues in the Compiler and Architecture Research group at Hewlett-Packard Laboratories who worked with us in developing the PICO system have greatly influenced the ideas and opinions expressed in this article: Rob Schreiber, Shail Aditya, Vinod Kathail, Scott Mahlke, Darren Cronquist, Mukund Sivaraman, Santosh Abraham, Greg Snider, Sadun Anik, and Richard Johnson.

References

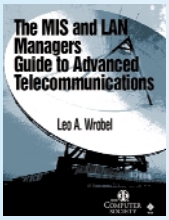
1. R.E. Gonzalez, "Xtensa: A Configurable and Extensible Processor," *IEEE Micro*, Mar./Apr. 2000, pp. 60-70.
2. A.K. Sharma, *Programmable Logic Handbook: PLDs, CPLDs, and FPGAs*, McGraw-Hill, New York, 1998.
3. D.W. Knapp, *Behavioral Synthesis: Digital System Design Using the Synopsys Behavioral Compiler*, Prentice Hall, Upper Saddle River, N.J., 1996.
4. J.P. Elliot, *Understanding Behavioral Synthesis: A Practical Guide to High-Level Design*, Kluwer Academic, Boston, 1999.
5. W.F. Lee, *VHDL: Coding and Logic Synthesis with Synopsys*, Academic Press, San Diego, Calif., 2000.

B. Ramakrishna Rau *is an HPL fellow and manager of the Compiler and Architecture Research group at Hewlett-Packard Laboratories. His research interests include computer architecture, compilers, operating systems, and the automated design of computer systems. Rau received a PhD from Stanford University. He is a Fellow of the IEEE and a member of the ACM. Contact him at rau@hpl.hp.com.*

Michael S. Schlansker *is a principal scientist at Hewlett-Packard Laboratories. His research interests include computer architecture, compilers, and high-level synthesis of computer systems. Schlansker received a PhD from the University of Michigan. He is a member of the IEEE and the ACM. Contact him at schlansk@hpl.hp.com.*

How will it all connect?

Find out in



The MIS and LAN Managers Guide to Advanced Telecommunications

$50 for Computer Society members

Regular price $99. Offer expires 14 August

Now available from the Computer Society Press

IEEE COMPUTER SOCIETY