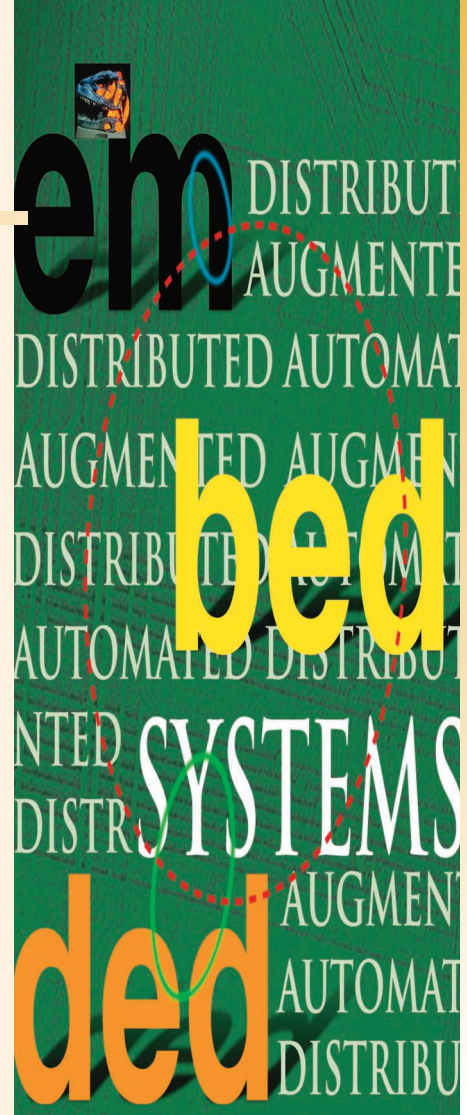


PICO: Automatically Designing Custom Computers

The PICO project automates the design of optimized, application-specific embedded computer systems to meet the demands of innovative smart products that require varying combinations of performance and cost.



Vinod Kathail
Shail Aditya
Robert Schreiber
B. Ramakrishna Rau
Darren C. Cronquist
Mukund Sivaraman
Hewlett-Packard
Laboratories

Embedded computers are everywhere—in video games, DVD players, TV sets, printers, scanners, cell phones, cars, and now even in smart robotic vacuum cleaners, lawnmowers, and virtual pets. Computers have displaced many analog circuits in photography, video, and telephony. The advent of system-level integration (SLI) foreshadows an era of yet more growth in the number and variety of innovative smart products and their embedded computers.

Such smart products demand varying combinations of performance, cost, and power. When a product mandates high performance, often the challenge is to lower cost to a level the market will accept. Whereas specialization increases performance and reduces cost, customization permits specialization when no adequately specialized, off-the-shelf design is available. Automation of embedded computer design would enable customization by lowering the barriers of design time, designer availability, and design cost, thereby unleashing the predicted explosion in smart products.

PICO ARCHITECTURE SYNTHESIS SYSTEM

The PICO (program in, chip out) project at HP Labs automates the design of optimized, application-specific computer systems. PICO uses an appli-

cation written in C to architect a set of high-quality system designs that trade cost for performance.

As Figure 1 shows, a PICO system design contains one EPIC/VLIW (explicitly parallel instruction computing/very long instruction word) processor¹ and an optional nonprogrammable accelerator (NPA) subsystem consisting of one or more NPAs, both connected to a two-level cache subsystem that, in turn, connects to the system bus. Each NPA is customized to execute a compute-intensive loop nest that would otherwise have been executed on the VLIW.

PICO generates the most cost-effective combinations of these subsystems to provide several high-quality system designs at varying points on the cost-performance tradeoff curve. PICO emits structural Verilog/VHDL for the hardware components, modifies application code to include software interfaces to the generated hardware, and retargets the compiler, assembler, and simulator to the custom VLIW processor.

Skeptics often assume that automated design must emulate human designers who can invent new solutions to problems. PICO's approach, however, is to automatically pick the most suitable designs from a large, well-engineered space of designs. In practice, it would be unrealistic for all designs to be

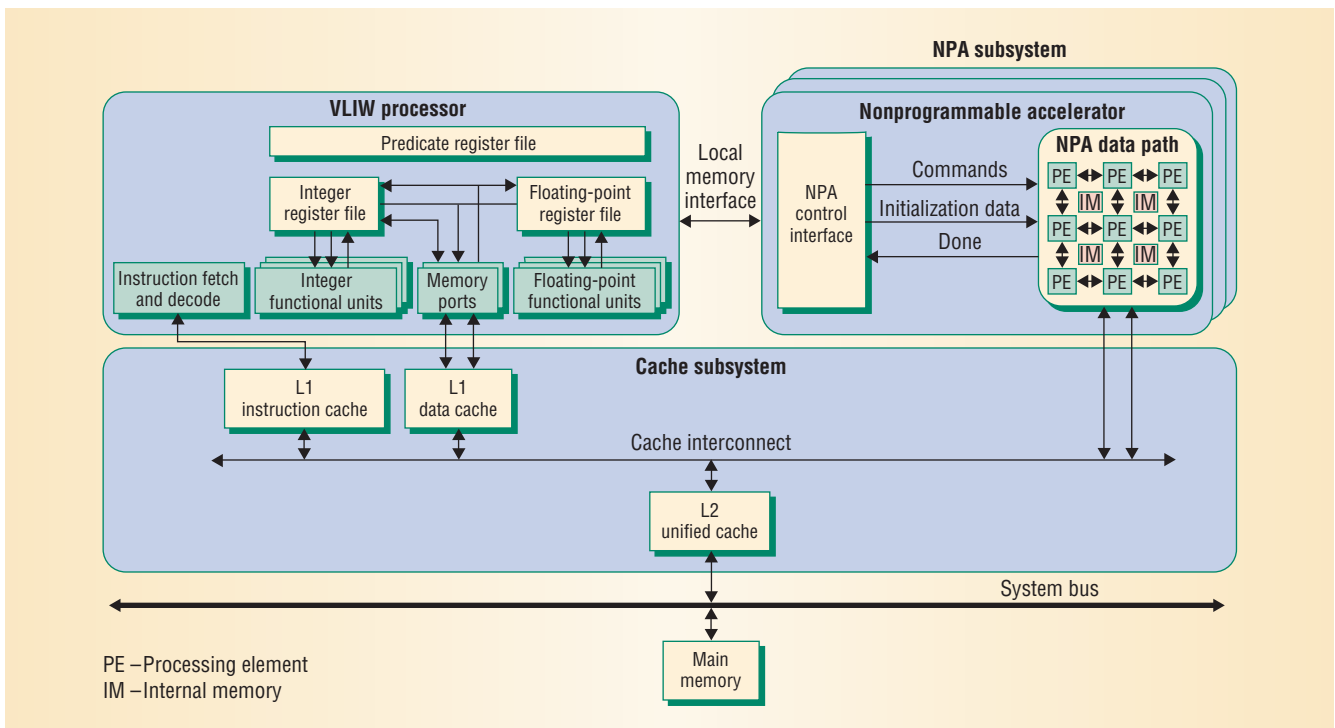


Figure 1. PICO's system-level architecture template. A system consists of three subsystems: a custom EPIC/VLIW processor; a custom, two-level cache hierarchy; and, optionally, an NPA subsystem that includes one or more custom NPAs.

preconstructed, so PICO's framework-based, hierarchical design methodology creates designs on demand during design space exploration.

FRAMEWORK-BASED AUTOMATION

Figure 2 shows PICO's framework-based design automation strategy. A framework consists of

- a parameterized architectural template that defines the space of designs to be considered,
- a spacewalker with a strategy for exploring the design space,
- a constructor that can construct every design in this space using components from a component library, and
- an evaluator that can measure the quality of any such design.

These elements together provide the basis for automatically identifying an approximate Pareto set—a set of designs, each of which is better than any other design in at least one measure of quality.

Template. The template defines parameters representing the design space and a set of rules and constraints that must be honored. Within the template, some aspects of the design, such as the presence of certain modules and how they connect, are predetermined. Architectural parameters specify other aspects of the design. For example, the number of memory ports in a processor might be a parameter in a processor design space. Once the parameters have been specified, a detailed construction of the design determines many of its other attributes. For example, the VLIW constructor determines a VLIW processor's instruction format.

When we have established an optimal or near-optimal algorithmic way of determining the design details, we view their definition as part of the construction task. When we have no clear way of determining a design's important aspects and must use a heuristic search to determine good values, we view these attributes as parameters. The number of parameters should be relatively small to be manageable. A specification is the set of values to which the parameters are bound and corresponds to a unique design.

Spacewalker. The set of all allowed parameter value combinations in the template defines the specification space. The spacewalker explores this space, looking for the Pareto-optimal designs. The template may fix the range of allowed values for a parameter, the user may set the range, or the spacewalker may determine the range through a preliminary examination of the application.

Using the quality metrics the evaluator provides, the spacewalker determines whether any previously examined design *eclipses* the new design—the new design is either equal or inferior to the previous design in all respects. If not, the new design is added to the Pareto set, discarding all previous designs in the set that the new design eclipses.

If a design space is too large to search exhaustively, we use a manifold strategy to search the design space more intelligently. The spacewalker uses heuristics to examine designs that are likely to be Pareto-optimal while avoiding the exploration of uninteresting designs. The goal is to find most of the Pareto-optimal designs while having examined only a small fraction of the specification space.

The spacewalker also maintains a repository of

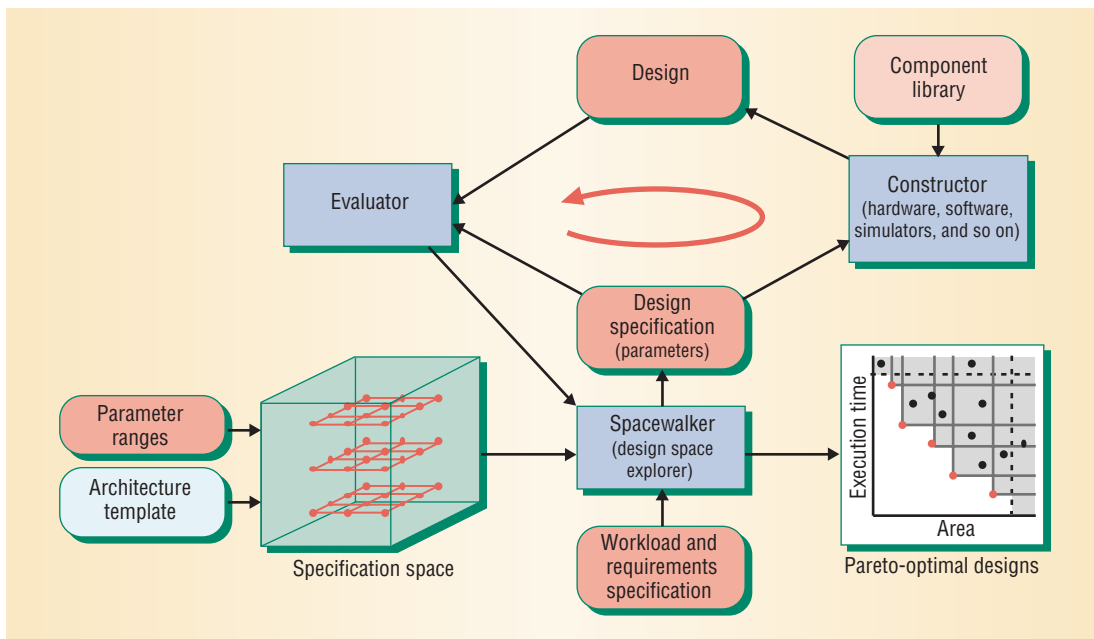


Figure 2. PICO's framework-based automation strategy. The PICO framework consists of a template, space-walker, constructor, evaluator, and component library.

previously explored designs to eliminate redundant examination of the same design. Analytical cost and performance modeling can help avoid construction and evaluation costs entirely. Finally, we use the basic divide-and-conquer paradigm, which in this context takes the form of hierarchical design and hierarchical design space exploration.

Constructor. Design construction derives a detailed design from the specification provided by the spacewalker. The spacewalker specification is, of necessity, an abstract description of the machine to be synthesized. The constructor fills in myriad details using the abstract description to create the best possible—and usually least costly—machine.

To construct the design, the constructor automatically assembles lower-level components chosen from a component library. Sometimes, these components are parameterized with respect to their structural properties such as bit width; once the parameters are specified, a generator automatically instantiates the specific component. The components themselves are designed, optimized, and characterized manually for properties such as area, power, and latency.

Evaluator. The framework also requires an evaluator that computes various metrics to assess the detailed design, the design specification, or both. In PICO, the cost of a design is measured in terms of gates or silicon area, and its performance is measured in terms of the number of cycles to execute the application via a combination of static estimation and simulation.

HIERARCHICAL DESIGN

To limit design complexity, designers decompose systems into subsystems that interact in a limited way. Identifying the best subsystem designs and then combining them to form complete systems

greatly reduces the system design space. A hierarchical design framework formalizes this intuitive approach.

A hierarchical system design framework decomposes the system into several subsystem frameworks. Each framework must include a template that defines the subsystem specification space, a spacewalker, a constructor, and an evaluator. Developers must define a system framework's specific decomposition strategy while designing the system framework, prior to automating its design. System composition, on the other hand, occurs during design space exploration.

System decomposition involves not only the system template's structural decomposition, but also decomposition of the constructor and the evaluator. As good software engineering practice demands, the subsystem constructors are typically already in place.

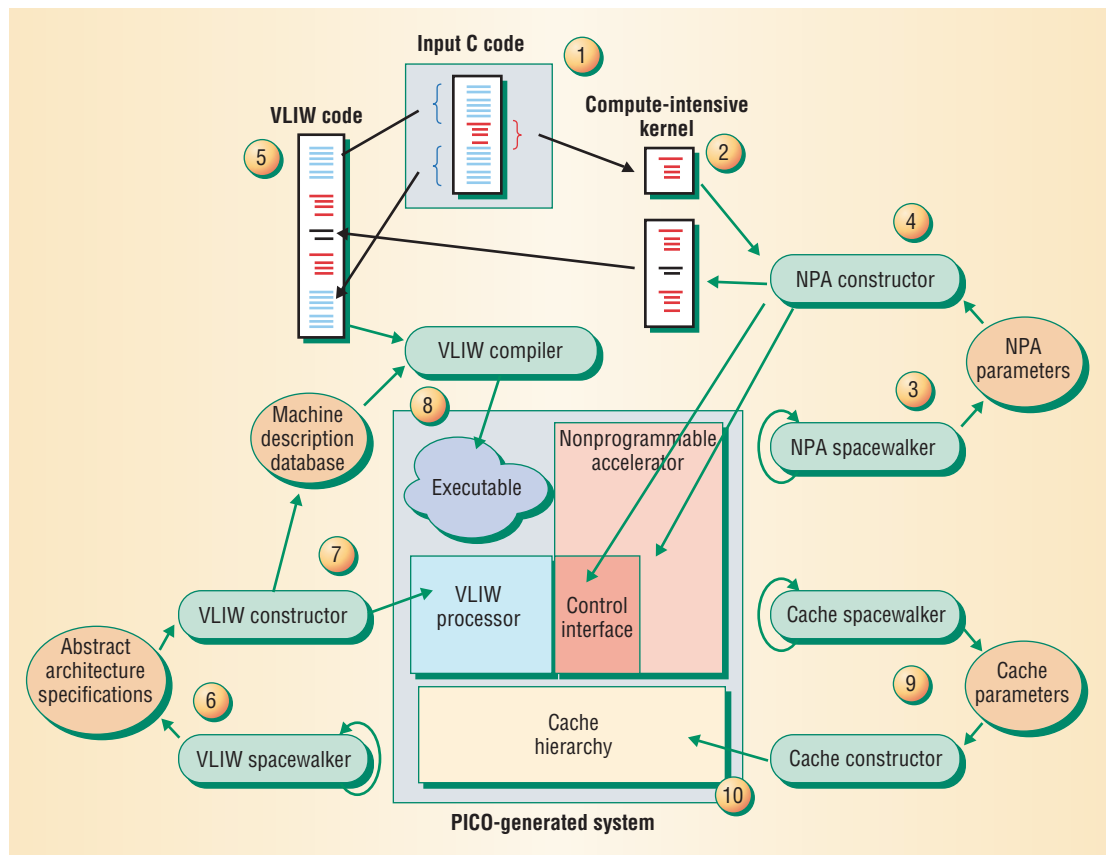
However, the challenge is decomposing the system evaluator. Subsystem evaluators can use different metrics than the system evaluator, but they must be good indicators of system-level performance. Specifically, the evaluators should reject poor subsystems without omitting subsystems that are system-level constituents of Pareto designs.

The key point in hierarchical design is that the design space at each level in the hierarchy is restricted to systems that can be built only from the Pareto-optimal subsystems. This restriction greatly reduces the size of the system design space that must be explored.

Spacewalking without decomposition

Consider, for example, a computer system template that consists of a processor and a data cache. Based on the specified parameters and ranges for these elements, suppose the system specification space holds 5,000 designs. An exhaustive spacewalk examines all 5,000 system specifications to find, for

Figure 3. PICO's hierarchical design flow. The NPA, VLIW, and cache subsystem frameworks use the input C application to produce Pareto-optimal subsystem designs that are composed to produce Pareto-optimal system designs.



example, 15 Pareto designs. These specifications comprise only 0.3 percent of the evaluated systems. Hierarchical design aims to reduce the number of designs examined without missing any Pareto points.

Spacewalking with decomposition

Consider a decomposition of the above system framework into a hierarchy consisting of a processor and a cache framework, each with its own set of parameters. Some system parameters become parameters of only one subsystem; others affect more than one subsystem. The exploration ranges of the subsystem parameters are derived from those of the system parameters. In addition, decomposition liberates some previously dependent system attributes, such as cache ports, and makes them independent subsystem parameters.

In this example, suppose there are 100 processor and 100 cache designs in their respective specification spaces, each containing 10 Pareto-optimal designs. Then, an exhaustive spacewalk examines only 300 designs—100 processor, 100 cache, and 100 system designs—to find the 15 Pareto designs, raising the exploration efficiency to 5 percent.

System composition and evaluation

In a well-formed system, various subsystems must obey certain interface constraints. In this example, one of the interface constraints is that processor load-and-store units—a processor subsystem parameter—must be equal to the cache-

ports parameter of the cache subsystem. Such interface constraints prevent subsystem designs from being interchangeable: A one-port data cache cannot be used in place of a two-port data cache, for example. Spacewalking for subsystems must account for this, or it will too aggressively remove subsystem designs that are necessary for Pareto-optimal system designs.

The system-level constructor must ensure that it builds and evaluates only valid combinations of subsystems. One possibility is to perform *validity filtering*, forming all possible combinations of Pareto subsystems and testing whether they satisfy the interface constraints to weed out invalid subsystem design combinations.²

When an interface constraint is a simple equality or inequality involving subsystem parameters, PICO avoids creating invalid systems altogether, using Pareto sets *indexed* by parameters to construct the valid systems directly.

PICO DESIGN FLOW

Figure 3 shows PICO's hierarchical design space exploration. After inputting a C application containing one or more compute-intensive loop nests or kernels (1), PICO first identifies and extracts each kernel (2). Then the PICO-NPA spacewalker repeatedly specifies an NPA, retaining only the best NPA designs (3). For each specification, the NPA constructor transforms the kernel to the requisite level of parallelism and main memory bandwidth

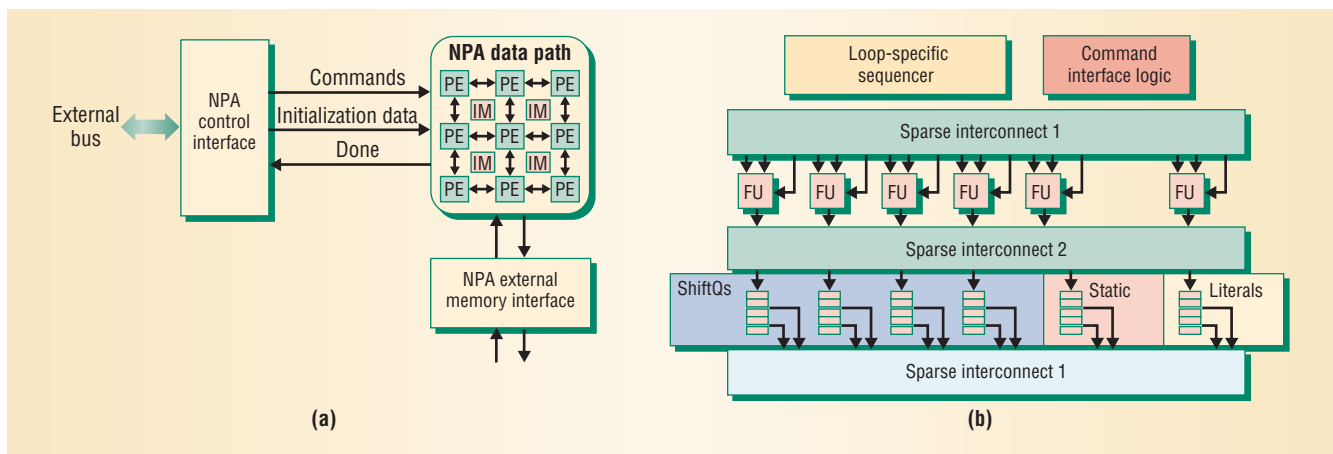


Figure 4. The NPA template. (a) An NPA consists of a one- or two-dimensional array of customized processing elements (PEs), with only synchronous parallel nearest-neighbor communication, and internal memories (IMs). (b) The given loop nest computation largely determines each PE's structure, which consists of an interconnected network of functional units and distributed register structures (ShiftQs).

and generates the register transfer level (RTL) design for the NPA along with the VLIW code that will repeatedly initialize and invoke the NPA (4).

At this point, using the combined VLIW code (5), the PICO-VLIW spacewalker repeatedly specifies a VLIW processor and retains only the best VLIW designs (6). For each specification, the VLIW constructor designs the VLIW processor's architecture and microarchitecture and emits an RTL design (7). In addition, the constructor generates a machine description of this processor for the Elcor VLIW compiler, which compiles the modified application to the VLIW processor (8). The cache spacewalker repeatedly specifies and evaluates cache subsystem configurations, retaining only the Pareto-optimal ones (9).

Finally, the system-level PICO spacewalker (not shown in the figure) combines compatible VLIW, cache, and NPA designs into Pareto-optimal system designs (10). During this process, and for each cost or performance level, PICO performs hardware-software partitioning and codesign by determining which kernels should be implemented as NPAs rather than as software on the VLIW.

NPA DESIGN

PICO-NPA accepts a loop nest in C, along with a range of performance requirements and available external memory bandwidth, and produces a Pareto set of NPAs customized for the given loop nest.³ For each Pareto design, PICO emits structural Verilog/VHDL that defines the NPA at the register transfer level, as well as requisite initialization and invocation code for the external host processor to execute.

Template

The NPA template consists of an array of customized processing elements (PEs) with only synchronous parallel nearest-neighbor communication, as Figure 4a shows. Further, the design can contain internal memories (IMs), an interface to external memory, and a memory-mapped host processor interface. As Figure 4b shows, each PE

has a loop-specific instruction sequencer and a data path consisting of

- functional units (FUs) with customized widths;
- distributed register structures with individual read and write access from and to any register, as opposed to addressable register files; and
- sparse interconnects customized to the loop nest.

The data path lacks centralized instruction storage, distributing the loop nest's operations across the FUs instead.

Constructor

First, a suite of loop-nest transformations and optimizations determines the placement of data in external memory, internal memory, or registers. Then the constructor tiles the iteration space to minimize the number of registers required while maximizing available external memory bandwidth utilization. The constructor schedules the loop iterations in time and maps them to PEs using the available parallelism to meet the given performance specification. Additional low-level optimizations follow, both standard (common subexpression elimination, dead code elimination, strength reduction) and novel (data-width inference and clustering).

Then, the constructor allocates a least-cost set of FUs capable of executing the operations in the optimized loop nest at the desired performance level given by its initiation interval (II) by solving an integer linear program. The constructor performs modulo scheduling to determine each operation's issue time and to bind each operation to an FU, so that each operation executes every II cycles without conflicting for resources.

The constructor uses various heuristics that cluster similar width operations and maximally share storage and interconnect, minimizing hardware costs prior to and during the scheduling and binding phase. ShiftQ⁴—a novel hardware structure consisting of registers and switches—buffers and transports operands between FUs. A dedicated ShiftQ

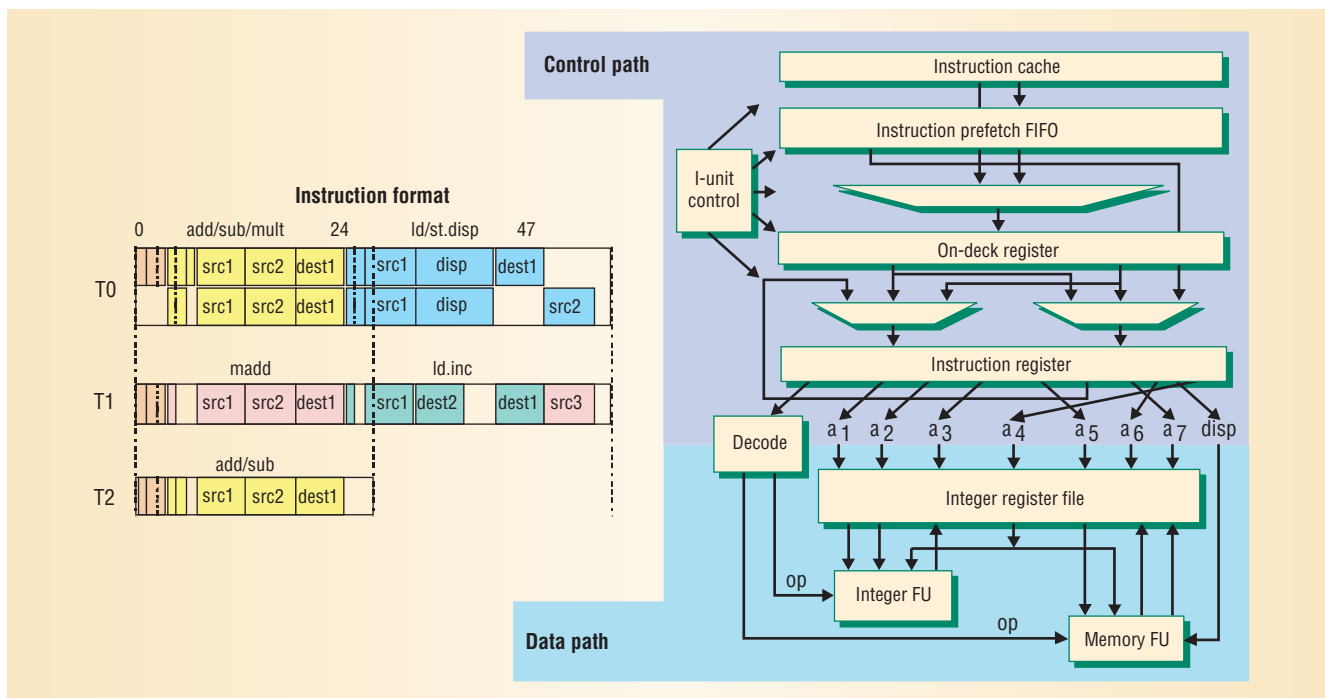


Figure 5. VLIW processor template. Although it fixes certain aspects of the design, the design allows flexibility elsewhere. For example, every VLIW processor contains an integer cluster consisting of an integer register file and a set of integer functional units connected to it. Optionally, the processor can contain a floating-point cluster with the same general structure.

for each FU minimizes the storage required for the results produced by the FU. The constructor creates connections from ShiftQ registers to FU inputs as needed, resulting in a sparse interconnect structure.

The constructor generates the instruction sequencer and creates multiple copies of the PE, interconnecting them in the geometry the spacewalker specifies. The constructor also generates internal memories, interfaces to the host processor and external memory, and provides arbitration and stalling circuitry, if needed. Finally, the constructor emits structural Verilog/VHDL for the NPA hardware and generates the C code that repeatedly invokes the NPA hardware after making appropriate initializations. This code is compiled onto the host processor along with the remainder of the application.

The NPA construction process permits simulation at several intermediate steps. By comparing these simulation results with those of the original source code, developers can detect errors in the input specification as well as errors that the PICO software introduces. In addition to the RTL artifact, the constructor also generates an RTL test-bench and memory simulation models for block-level RTL verification and a cycle-accurate C model to support system-level verification.

Evaluator

The NPA cost evaluation uses the parameterized formulas for area and gate count attached to each

component in the macrocell library to estimate the design's chip area and gate count. These formulas have been calibrated for a specific ASIC design-flow and process technology. The constructor designs the NPA to a steady-state performance supplied as a parameter. To compute the overall NPA performance, the evaluator makes adjustments for the time required to fill and drain the pipeline of PEs and for the delay anticipated due to memory-induced stalls.

VLIW PROCESSOR AND COMPILER DESIGN

PICO-VLIW uses a C application, with test data, to automatically produce a set of Pareto-optimal, custom VLIW processor designs. For each design, PICO-VLIW architects the processor—including the instruction format, the execution data paths, and the instruction unit—and emits structural Verilog/VHDL for it. In addition, it retargets the Elcor compiler for VLIW architectures to the newly designed processor.⁵

Template

The PICO-VLIW template shown in Figure 5 encompasses a broad class of EPIC/VLIW processors having advanced architectural and microarchitectural features.¹ Their operation repertoire is a customized subset of the HPL-PD operation set,⁶ optionally augmented by user-defined operations. The processors can issue multiple operations per cycle executing on multiple FUs. In addition to the global address space that the main memory and the

cache subsystem represent, a processor can use distinct load and store operations to access one or more local memories.

HPL-PD specifies four files for organizing the registers: integer, floating-point, predicate, and branch. FUs that do not need simultaneous access can share register file ports. To reduce code size, two or more FUs that cannot issue operations simultaneously can share the same bit positions in the instruction format. This format can consist of several instruction templates of different lengths to reduce the number of explicit no-op operations.

Spacewalker

The VLIW spacewalker emits a high-level architecture specification that specifies the types and sizes of register files, the operation repertoire as a collection of operation groups, and an abstract specification of the processor's instruction-level parallelism as mutual exclusions between operation groups. Operations in two different operation groups execute in parallel unless there is a mutual exclusion between the two groups. Furthermore, operations within an operation group cannot be issued in parallel. In the current implementation, the operation groups are instances of four operation types—the integer, floating-point, load-and-store, and branch operations—and there are no explicit mutual exclusions between operation groups.

The VLIW spacewalker customizes the operation repertoire to the application: Any operation in HPL-PD repertoire that the application does not use can be deleted. Alternatively, we can retain a core set of operations if required for greater generality.

The VLIW spacewalker's specification space can be quite large, and the evaluation of each design point can be time-consuming because it involves recompiling the full application program. Thus, exhaustive search is impractical, and the spacewalker adaptively limits its specification space search based on information gleaned from the points it has already examined.⁷ The key idea is to make incremental changes to the current design's parameters and to terminate the current search direction if all the current design's neighbors are non-Pareto designs.

Constructor

The VLIW constructor first generates the least expensive data path it can, consistent with the concurrency level the abstract architecture specification demands. The constructor uses the specified mutual exclusions to minimize the number of logical FUs required and to maximize register port sharing between them. The constructor may use a

single physical FU to implement the union of operations in mutually exclusive operation groups. Or, it may use multiple physical FUs that share register file ports.

The constructor then designs an instruction format for the processor using mutual exclusions and compiler feedback. The instruction format consists of a number of variable-length instruction templates that judiciously balance the total code size against the complexity of the instruction decode and distribution network.

The constructor uses mutual exclusions to design a basic instruction format that supports the requisite level of concurrency. It also generates an Elcor machine description, compiles and schedules the application, and gathers statistics on which sets of operations are frequently issued together. It uses these statistics to augment the basic instruction format with additional templates for operations that are frequently issued together in order to reduce the total code size.

The constructor next generates an instruction unit consisting of the instruction prefetch, alignment, and decode hardware customized for this instruction format using the schema specified in the VLIW template.

At this point, the detailed design is complete, and the constructor generates structural Verilog/VHDL. It also creates the final machine description needed to retarget the compiler, assembler, and simulator.

Evaluator

The VLIW evaluator estimates chip area and gate count in the same way as the NPA evaluator. To estimate the application's processor runtime, the evaluator multiplies each basic block's schedule length by its profiled execution frequency and sums the total over all basic blocks. This method produces an accurate performance estimate for a statically scheduled processor ignoring stall cycles due to cache misses. The cache subsystem evaluator is responsible for computing the additional stall cycles due to cache misses.

The evaluator also reports the generated object code's size to evaluate its incremental effect on the cost of main memory, which is charged to the VLIW cost. The code size also affects the evaluation of instruction cache (I-cache) and unified cache (U-cache) misses in the I-cache framework.

SYSTEM DESIGN

System framework decomposition is valid only if the various subsystem frameworks can be inde-

The spacewalker adaptively limits its specification space search based on information gleaned from points it has already examined.

Framework-based automation offers a powerful methodology for automating the design of complex processors and computer systems.

pendently constructed, evaluated and then combined to give a reasonable approximation to the overall system Pareto. PICO's system-level decomposition strategy, therefore, hinges upon two important factors.

First, subsystem compatibility is captured by using a set of additional parameters for each subsystem that participates in interface constraints during system composition. Given a family of parameterized subsystem Pareto sets, the system-level PICO space-walker combines only compatible VLIW, cache, and NPA designs into a system-level Pareto set. In the process, PICO determines which kernels should be implemented as NPAs rather than as software routines on the VLIW.

Second, PICO assumes that each subsystem can be independently evaluated for cost and performance using its parameters. This assumption is valid under certain design constraints that PICO enforces, and it has been verified both analytically and experimentally. For example, the evaluation of the memory hierarchy is broken into separate evaluations for the data cache (D-cache), I-cache, and U-cache. This decomposition is valid if the U-cache includes all data contained in the I-cache and D-cache.

PICO uses dilation, an empirical parameter that is the ratio of the compiled application code size on the given processor with respect to a fixed reference processor, to capture the effect of varying the instruction format of various VLIW processors over the I-cache and the U-cache instruction misses.⁸ Processors with various dilations are evaluated independently and then matched with cache subsystems with the same dilation without the need to evaluate each processor-cache pair separately.

Framework-based automation offers a powerful methodology for automating the design of complex, high-level structures such as processors and computer systems. It has been crucial to PICO's success in designing NPAs and VLIW processors. We strongly suspect that automated design is possible only with this sort of methodology.

The restrictions that a parametric template places on the design space remove enough variability to make automation possible; the existence of a template in turn makes designing automatic spacewalkers, constructors, and evaluators possible. The nature and number of parameters in the template determine the framework generality and flexibility. Frameworks can be quite varied. The template for one framework might contain RISC processors and DSPs, while

another might contain a vector processor.

Hierarchical design methodology, on the other hand, makes the problem tractable. Our experience with hierarchical design indicates that decomposing the evaluator presents a very challenging problem. Developers can easily decompose a template to identify the interface constraints and parameters, and to design spacewalkers and constructors for the resulting subsystems. Engineering acceptably accurate subsystem evaluators requires judicious decomposition. We believe that this is the most important research problem in hierarchical design. Until this problem is solved in an adequately general manner, developers of complex systems might need to restrict themselves to frameworks that easily lend themselves to hierarchical evaluation.

Engineering disciplines tend to go through fairly predictable phases: ad hoc, formal and rigorous, and automation. When the discipline is in its infancy and designers do not yet fully understand its potential problems and solutions, a rich diversity of poorly understood design techniques tends to flourish. As understanding grows, designers sacrifice the flexibility of wild and woolly design for more stylized and restrictive methodologies that have underpinnings in formalism and rigorous theory. Once the formalism and theory mature, the designers can automate the design process. This life cycle has played itself out in disciplines as diverse as PC board and chip layout and routing, machine language parsing, and logic synthesis.

We believe that the computer architecture discipline is ready to enter the automation phase. Although the gratification of inventing brave new architectures will always tempt us, for the most part the focus will shift to the automatic and speedy design of highly customized computer systems using well-understood architecture and compiler technologies. ■

Acknowledgments

We thank past members of our group: Scott Mahlke, Mike Schlansker, Santosh Abraham, Greg Snider, Sadun Anik, and Richard Johnson. Alain Dartre and Lothar Thiele made certain key theoretical contributions. The following research interns accelerated PICO software development: Frederic Vivian, Bruce Childers, Marnix Arnold, Matthew Jennings, Roderic Rabbah, Timothy Sherwood, Rajiv Ravindran, Guillaume Huard, Luca Ceresoli, Kevin Fan, and Benoit Meister. We thank Henk Corporaal for providing the code for the attractive GUI to his MOVE infrastructure, from which we

developed PICO's GUI, and Wen-mei Hwu and his IMPACT group for providing Elcor's machine-independent front end.

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, and Synthesis for Embedded Systems (CASES 2000)*, ACM Press, New York, 2000, pp. 71-79.
3. R. Schreiber et al., "PICO-NPA: High-Level Synthesis of Nonprogrammable Hardware Accelerators," *J. VLSI Signal Processing*, vol. 31, 2002, pp. 127-142.
4. S. Aditya and M.S. Schlansker, "ShiftQ: A Buffered Interconnect for Custom Loop Accelerators," *Proc. Int'l Conf. Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2001)*, ACM Press, New York, 2001; <http://hplabs.hp.com/research/itc/car/Templates/carpapers-hpl.html#2001>.
5. 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.
6. V. Kathail, M.S. Schlansker, and B.R. Rau, *HPL-PD Architecture Specification: Version 1.0*, tech. report HPL-93-80R1, Hewlett-Packard Laboratories, Palo Alto, Calif., 2000; <http://www.hpl.hp.com/techreports/93/HPL-93-80R1.html>.
7. G. Snider, *Spacewalker: Automated Design Space Exploration for Embedded Computer Systems*, tech. report HPL-2001-220, Hewlett-Packard Laboratories, Palo Alto, Calif., 2001; <http://www.hpl.hp.com/techreports/2001/HPL-2001-220.html>.
8. 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 99)*, IEEE CS Press, Los Alamitos, Calif., 1999, pp. 114-125.

Vinod Kathail is a principal research scientist and manager in the Compiler and Architecture Research Program at Hewlett-Packard Laboratories. His research interests include parallel computer architectures, compilers, programming languages, and automated design of custom computer systems. He received an ScD in electrical engineering and computer science from MIT. He is a member of the IEEE and the ACM. Contact him at vinod_kathail@hp.com.

Shail Aditya is a senior research scientist in the Compiler and Architecture Research Program at Hewlett-Packard Laboratories. His research interests include programming language design, multi-threaded and VLIW compilers and processor architectures, embedded system design, and design automation. He received a PhD in electrical engineering and computer science from MIT. He is a member of the IEEE Computer Society. Contact him at aditya@hpl.hp.com.

Robert Schreiber is a principal scientist in the Compiler and Architecture Research Program at Hewlett-Packard Laboratories. His research interests include scientific computing, sequential and parallel algorithms for matrix computation, embedded computer systems, and compiler optimization of parallel programs. He received a PhD in computer science from Yale University. He is a member of SIAM and the ACM. Contact him at schreiber@hpl.hp.com.

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

Darren C. Cronquist is a research scientist in the Compiler and Architecture Research Program at Hewlett-Packard Laboratories. His research interests include reconfigurable computing, high-level synthesis, compilers, and embedded systems. He received a PhD in computer science from the University of Washington. He is a member of the IEEE and the ACM. Contact him at Darren_cronquist@hp.com.

Mukund Sivaraman is a research scientist in the Compiler and Architecture Research Program at Hewlett-Packard Laboratories. His research interests include computer architecture, high-level synthesis, timing verification, delay-fault testing, and design automation of computer systems. He received a PhD in electrical and computer engineering from Carnegie Mellon University. He is a member of the IEEE and the ACM. Contact him at mukund@hpl.hp.com.