Embedded Computing: New Directions in Architecture and Automation

B. Ramakrishna Rau and Michael S. Schlansker

Compiler and Architecture Research

HPL-2000-115

September 2000

rau@hpl.hp.com, schlansk@exch.hpl.hp.com

embedded computing, special-purpose architectures, customization, custom architectures, off-the-shelf customizable systems, FPGA, automation, architecture synthesis, hardware-software co-design, processor-compiler co-design, frameworks, constructors, constructors, design space exploration, PICO, system synthesis, VLIW synthesis, non-programmable accelerator synthesis, cache hierarchy synthesis

With the advent of system level integration (SLI) and system-on-chip (SOC), the center of gravity of the computer industry is moving from personal computing into embedded computing. The resulting upheaval is only just beginning to be widely appreciated. The opportunities, needs and constraints of this next generation of computing are somewhat different from those to which we have got accustomed in general-purpose computing. In turn, we believe that this will lead to significantly different computer architectures, at both the system and the processor levels, and a rich diversity of off-the-shelf and custom designs. Furthermore, we predict that embedded computing will introduce a new theme into computer architecture: automation of computer architecture. In this report, we elaborate on these claims and provide, as an example, an overview of PICO, the architecture synthesis system that the authors and their colleagues have been developing over the past five years.

A version of this publication will appear in the Proceedings of the 7th International Conference on High-Performance Computing (HiPC2000), Bangalore, India, December 2000.

1 Introduction

Over the past few decades, driven by ever increasing levels of semiconductor integration, the center of gravity of the computer industry has steadily moved down from the mainframe price bracket to the personal computer price bracket. Now, with the advent of system level integration (SLI) and system-on-chip (SOC), the center of gravity is moving into embedded computing.

Embedded computers hide within products designed for everyday use as they assist our world in an invisible manner. We refer to such products as **smart products**. Embedded computers have been incorporated into a broad variety of smart products such as video games, digital cameras, personal stereos, televisions, cellular phones, and network routers. A digital camcorder, for instance, uses a high-performance embedded processor to record or playback a digital video stream of data. These embedded processors achieve supercomputer levels of performance on highly specific tasks needed for recording and playback. It is the availability of high-density VLSI integration that makes it practical to provide such product-defining performance at an affordable cost.

As VLSI density increases embedded processors continue to provide more compute power at even lower cost. This is stimulating the rapid introduction of a vast array of innovative smart products. Newly defined digital solutions, capable of inexpensively performing complex data manipulations provide revolutionary improvements to product functionality. Embedded processors are often used in products that previously relied on analog circuitry. When analog signals are digitally represented, digital processing performance increases can be used to provide higher speed, higher accuracy signal representation, more efficient storage, and more sophisticated processing uniquely available in the digital domain.

Any computer architecture must be designed to take advantage of the opportunities afforded by the latest technologies while taking into consideration the requirements of the market, product and application. We now examine the way in which these requirements are different in the embedded space relative to those for the general-purpose space.

1.1 Product Requirements

Demanding product requirements often constrain the design of embedded systems from many sides. A smart product may simultaneously need, higher performance, lower cost, lower power, and higher memory capacity. High throughput is especially important in dataintensive tasks such as imaging, video, signal processing, etc. Products often have demanding real time requirements that further exaggerate processing needs. In these cases, embedded processors are required to perform complex processing steps in a limited and precisely known amount of time. Compute-intensive smart products use special-purpose processing engines to deliver very high performance that cannot be achieved with a general-purpose processor and software. Such products are often enabled by **non-programmable accelerators** (**NPAs**) that accelerate performance critical functions.

Often, cost is more important than performance, with budgets that allow only a few cents for critical chips. Power is obviously important for battery-operated devices, but can be equally important in office environments where densely stacked equipment requires expensive cooling. In such settings, long term operating costs can easily exceed the purchase price of equipment. Lack of memory storage is a serious problem and compression techniques are used to conserve storage especially for high-volume video, image, and audio data. The storage of large computer programs is often too costly and processors are valued for their small code size. In some systems, programs are compressed to reduce their size and later decompressed for execution.

While the speed of general-purpose processors may exceed 1GHz, embedded processors often execute at modest clock speeds relying instead on parallelism to achieve needed performance. The use of parallel execution, rather than high clock speed, allows for designs with cheaper, lower power circuits. These low cost and power-efficient designs are also less dependent on precisely tuned low level circuitry. Additional power management features, such as gated or variable speed clocks, may also be used.

1.2 Market Requirements

Smart products require newly developed, and highly specialized, embedded systems in order to provide the novel and product-defining features necessary in a competitive marketplace. Often, the introduction of a new smart product depends upon the successful design and fabrication of a new chip that gives the product its distinguishing, high value features. Time-to-market determines the success or failure of products and businesses. While one **smart product vendor** (**SPV**) awaits a tardy chip design before he can ship product, a competing SPV's product takes advantage of the higher profit margins and visible press available during early product introduction. The SPVs who deliver product early capture the lion's share of the profit and visibility. Those whose products are late, miss the market window and must settle for what is left. The ability to design rapidly provides a distinct market advantage.

There is a proliferation in the number of smart products being introduced, and an escalating number of embedded system designs are needed to support them. The problem is exacerbated by the shorter life cycles experienced by smart products in the consumer space, since each generation of a rapidly improving smart product requires new embedded systems in order to improve functionality.

The need for complex new chip designs cannot currently be satisfied. Product innovation is limited by our ability to perform expensive and time-consuming design tasks. A design bottleneck results from the use of a limited pool of highly talented engineers who must design a greater variety of complex chips at ever increasing rates.

1.3 Application Characteristics

Embedded applications are characterized by small well-defined workloads. Embedded systems often are used within single function products. Such products have product-specific and portable forms with simple and intuitive user interfaces. Single function products are based on far simpler software than general-purpose computers. They do not have the system crashes and reboots that we commonly associate with general-purpose systems. Their administration is simple and computer-illiterate users can enjoy their use. Single-function products with complex, high-performance, embedded systems are increasingly popular as the cost of their electronic content decreases. Many users do not, and cannot, write programs for their products. To them, programs are invisible logic which is used to provide product-defining functions.

Often, key kernels within applications represent the vast majority of required compute cycles. Kernels consist of small amounts of code which must run at very high performance levels to enable a smart product's functionality. In video, image and digital signal processing, the applications' execution times are often dominated by simple loop nests with a high degree of parallelism. These applications are typically characterized by a greater degree of determinacy then general-purpose applications. Not only is the nature of the application fixed, but key parameters such as loop trip counts or array sizes are pre-determined by physical product parameters such as the size of an imaging sensor. For such demanding application-specific products, high performance and high efficiency is often obtained using deeply pipelined function units which have been crafted into custom architectures designed to solve kernel codes.

Products may also have real-time constraints where time-critical computing steps must be completed before well understood deadlines to prevent system failure. Real-time systems require highly predictable execution performance. This often requires the use of a real-time operating system which provides task scheduling mechanisms necessary to guarantee predictable performance. Complex dynamic techniques, such as the use of virtual memory, caches, or dynamic instruction scheduling may not be allowed. When these techniques make the accurate prediction of performance excessively difficult, real-time constraints cannot be verified.

1.4 Overview of this Report

The movement of the center of gravity of computing, from general-purpose personal computers to special-purpose embedded computers, will cause a major upheaval in the computer industry. Successful computer architecture has always resulted from a judicious melding of the opportunities afforded by the latest technologies with the requirements of the market, product and application. These are significantly different for embedded computing leading to substantially different computer architectures, at both the system and the processor levels, as well as a rich diversity of **off-the-shelf** (**OTS**) and custom designs. Furthermore, we believe that embedded computing will introduce a new theme into computer architecture: the automation of computer architecture.

In the rest of this report, we elaborate on these points. In Section 2, we look at embedded computer architectures, and explain the increased emphasis on special-purpose architectures over general-purpose ones. Section 3 considers the issue of customization and the circumstances that force a SPV to resort to it instead of using an OTS solution. In Section 4 we argue for the use of automation in architecting and designing embedded systems, and we articulate our philosophy for so doing. As an example of this philosophy, Section 5 provides an overview of PICO, the architecture synthesis system that the authors and their colleagues have been developing over the past five years.

2 Embedded Computer System Architecture

To achieve the requisite performance, high-performance embedded systems often take a hierarchical form. The system consists of a network of processors; each processor is devoted to its specialized computing task. Processors communicate as required by the application and network connections among processors support only the required communications. Each processor also provides parallelism and is constructed using networks of arithmetic units, memory units, registers, and control logic. Hierarchical systems jointly offer both the process-level parallelism achieved with multiple processors as well the instruction-level parallelism achieved by using multiple function units within each processor.

For efficient implementation, and to provide product features such as highest performance, lowest cost, and lowest power, embedded computer system designs are often irregular at both levels of the design hierarchy. The entire system, as well as each of the processors, represents a highly special-purpose architecture that shows a strong irregularity that closely mirrors application requirements and ideally supports kernel needs.

Whereas specialization can be used to provide the very highest performance at the very lowest cost, embedded systems employ architectures which span a spectrum. At one end, a special-purpose architecture provides very high performance with very good cost-performance, but little flexibility. At the other end of the spectrum, a general-purpose architecture provides much lower performance and poorer cost-performance, but with all of the flexibility associated with software programming. A third choice, the OTS customizable architecture, provides an increasingly important compromise between these two extremes.

2.1 Special-Purpose Architectures

Smart products often incorporate special-purpose embedded processing systems in order to provide high performance with low cost and power. For example, a printer needs to perform multiple processing steps on its input data e.g.: error correction, decompression, image enhancement, coordinate transformation, color correction, and final rendering. Enormous computation is needed to perform these steps. To achieve the needed performance, printers often use a pipeline consisting of multiple processors; data is streamed through this pipeline of concurrently executing special-purpose processors. This style of design produces relatively inexpensive and irregular system architectures that are specialized to an application's needs.

In performance- and cost-critical situations, these embedded computer systems are specialized to simplify circuitry. Enormous savings can be achieved by specializing high-bandwidth connections among processors. Connections of appropriate bandwidth are provided between processors, exactly as needed, to accommodate very specific communication needs. While high bandwidth data paths are provided among processors that must exchange a large volume of data, no data paths are provided among processors that never communicate.

Likewise, the design of each processor is also specialized to the specific needs of the task that it performs. If we look within the design of each processor, we again see that specialization greatly simplifies needed circuitry. Each processor's performance, memory, and arithmetic capability are all adjusted to exactly match its dedicated task needs. Arithmetic units and registers are connected with a network of data paths that is specific to task needs. Each arithmetic unit, data path, or register is optimized to exact width requirements dictated by the statically known arithmetic precision of the operations and operands that they support. This specialization process again eliminates substantial amounts of unnecessary circuitry.

Thc control circuitry for each special-purpose processor can also be specialized to exact task requirements. Control circuits often degenerate to simple state machines which are highly-efficient in executing simple dedicated tasks. RAM structures within each processor are distributed according to need. Special table look up RAMs may be connected directly to the arithmetic units which use their operands. Each RAM is minimal in size in both number and width of its words. Rather precise information about the application is used to squeeze out unnecessary arithmetic, communication, and storage circuitry. Chained sequences of arithmetic, logical, and data-transfer operations are statically optimized to squeeze out additional circuitry. These optimized circuits perform multiple operations using far less logic than would be required if cascaded units were designed to execute each operation separately.

2.2 General-Purpose Architectures

The widespread use of special-purpose architectures sharply increases the number of distinct architectures that must be designed. If satisfactory programmable general-purpose system architectures existed, they would eliminate the need to specialize architectures to specific applications. Such general-purpose and domain-specific systems would offer the hope that new smart products could be designed using OTS parts that are reusable across many new smart products. General-purpose systems are highly flexible and are, in fact, reusable across almost all low-performance applications. General-purpose systems are designed in a number of ways. A general-purpose RISC processor can be re-programmed for a large variety of tasks. Domain-specific systems are customized to specific application areas (e.g. digital signal processing) but not to a specific smart product or application. Domain-specific systems often incorporate a control processor to run an operating system and a digital signal processor (DSP) to accelerate signal processing kernel code. Too often, however, general-purpose and domain-specific systems are not are not able to meet demanding embedded computing needs. RISC, superscalar, VLIW, and DSP architectures do not efficiently scale beyond their respective architectural limits. They have limits in clock speed and ability to exploit parallelism that make them costly and impractical at the highest levels of performance.

Symmetric multiprocessors are commonly used to extend system performance through the addition of more processors. Because general-purpose multiprocessors are designed without application knowledge, they provide uniform communication among processors. Identical processors are connected in a symmetric manner. General-purpose interconnection networks and multiprocessor shared memories provide the required general and parallel access. However, such highly connected and symmetric hardware scales poorly as the number of processors is increased. The communication hardware is either over-designed, permitting high-volume data transfers that never occur, or under-designed and unable to handle those that do. For large numbers of processors, the interconnect requires large amounts of chip area and the long transmission delays adversely affect the cycle time.

General-purpose systems rely on general-purpose processors for their computing horsepower. Each general-purpose processor uses flexible, general-purpose, function units (FUs) that can execute any common operation that might be needed in an arbitrary program. The general-purpose FU, however, is very expensive when compared to specialized function units (within custom processors) that execute only a single operation type. General-purpose processors often require symmetric access between function units and registers or RAMs. They use expensive and slow multi-ported register files and multi-ported RAMs to allow general (and parallel) communication and storage. Multi-ported register files and multi-ported RAMs do not scale and processors retain their efficiency at only modest levels of parallelism. Since the general-purpose processor is expected to be capable of executing a broad range of applications, it ends up being both over-designed and, often, under-designed for the specific aplication it is called upon to execute in the embedded system.

The control for a general-purpose processor is based on instructions that require wide access, complex shifting, and long distance transfer. The unpacking of the complex instruction formats needed to reduce code size can be very complex. This problem is especially difficult for wide-issue superscalar or VLIW machines. Their instruction units are well designed for supporting arbitrary and large programs on machines of modest issue width. But, for simple and highly parallel tasks, they are too expensive when compared to state-machine based controllers found in special-purpose systems.

2.3 Off-the-shelf, Customizable Systems

In many settings, general-purpose and domain-specific OTS processor chips cannot deliver adequate computing power. Product designers seek other OTS chip architectures to meet high-performance needs. Field Programmable Gate Arrays (FPGAs) [1] provide one alternative approach. FPGAs use programmable logic cells interconnected with a network of wires and programmable switches. Rather than relying on normal sequential programming, FPGAs are programmed using hardware design techniques. FPGAs have traditionally been used to implement simple control logic and "glue logic"-the left over logic needed to glue key components together. An FPGA allows such logic to be collected within a single chip to reduce system cost. The density of FPGAs has grown to the point where complex data paths are now possible on a single FPGA. The architecture of FPGAs continues to improve as features are added to support wider data paths, wider arithmetic, and substantial amounts of on-chip local memory. With these improvements, FPGAs are increasingly used to implement special-purpose, high-performance processors.

The data path and control of an embedded architecture can be specified as a circuit or as network of hardware functions. The circuit can execute operations from a repertoire of memory and logic operations that are supported directly, or through libraries, in OTS FPGA hardware. The embedded system circuit can be carefully specialized to application needs. After completing logic design, the circuit is then mapped onto an FPGA. The simplified logic diagram is mapped onto logic cells and placed and routed within OTS FPGA hardware.

While not as easy as software programming, embedded system design using FPGAs eliminates expensive, risky, and slow chip design efforts and substantially decreases product risk and time-to-market. Because they are programmable, the cost of fixing bugs in FPGA-based systems is small. A fix can be quickly tried and shipped. Firmware downloads can be used to fix FPGA related problems. FPGAs offer an increasingly important programming paradigm for delivering high-performance processing and rapid time to market.

Due to inherent hardware costs for supporting programmable logic, FPGAs cannot hope to be as efficient as customized hardware. Though FPGAs may be an order of magnitude less efficient than customized hardware, FPGA-based designs of high-performance processors can be far more efficient than designs that rely solely on general-purpose processors for computing horsepower.

FPGA vendors now provide OTS chips that contain a general-purpose processor, FPGA, and RAM hardware in a single SOC. These FPGA-based architectures permit the design of complex special-purpose processing systems. The general-purpose processor provides flexible low-performance computing while the FPGA, along with its associated configurable RAM blocks, allows high-performance special-purpose processors to be implemented as programmable logic. FPGA libraries now also provide processor cores that are programmed as FPGA logic. These processors can be modified or enhanced for specific application needs. Other functionality, such as support for peripheral interfaces and programmable I/Os, further increase FPGA utility.

3 System Customization

By customization we mean the process of taking an OTS system and modifying it to meet one's requirements. In its extreme form, it entails designing the system from scratch. At one level, customization is quite commonplace. For instance when one buys a personal computer, one typically configures the amount of memory, disk, and the set of devices connected to the peripheral bus. But one never modifies the processor or the system architecture (e.g., its bus structure). That task is viewed as the domain of the semiconductor or computer manufacturer from whom we expect to buy an OTS system. Our discussion of customization is focused on this part of the overall embedded system, what we will refer to as the **central computing complex** (**CCC**), which is the set of processors, memories and interconnect involved in executing the embedded application.

Customization of the CCC increases the design cost of smart products and often delays product introduction. Whereas it can greatly simplify the system, allowing high performance at low cost, it requires a complicated design process. A customized system involves complex tasks, including architecture, logic, circuit, and physical design. Designs must be verified and masks fabricated before chip production can begin. The SPV is well advised to use an OTS CCC when possible; from the viewpoints of time-to-market, engineering effort and project risk, this is clearly the preferable approach.

And yet, SPVs routinely design custom CCCs, processors and accelerators. What are the circumstances that force them to do so? Why is it that what the SPV needs is not available OTS, and that using something OTS would fall far short of his needs?

3.1 Why Customize?

Our view is that three conditions, in conjunction, create the situation that forces a SPV to have to customize his CCC. Firstly, the smart product must have challenging requirements which can only be met by specializing the system or processor architecture, as described in Section 2. Else, the SPV could just use an OTS general-purpose design. Secondly, for the given application, the performance, perhaps even the usability, of designs within the space of meaningful, special-purpose designs must be very disparate. For this particular application, all the OTS designs must fall short to such an extent that it is worth the SPV's while to pay the costs of customization: longer time-to-market, greater engineering effort and increased project risk. Lastly, the space of worthwhile special-purpose designs must be so large, that it is not possible, or not economical, for someone to make them all available, on an OTS basis.

This last criterion raises a further question. If it was worth the SPV's while to create a custom design, why was it not so for a manufacturer of OTS designs to have designed and offered the same thing? There are at least three reasons that serve as explanations. The most frequent reason is that the application, or a portion of it, represents the product-defining functionality that provides competitive differentiation to the smart product, i.e., it incorporates algorithms that are proprietary. If the performance requirements on these algorithms are sufficiently demanding, the CCC architecture must be specialized to reflect these proprietary algorithms; it must be customized. Secondly, the unit volume represented by a given smart product may be too low to make it worthwhile for a supplier to provide an OTS system. A final possibility, is that the diversity of special-purpose solutions demanded by SPVs is just too large for every one of them to be provided as OTS solutions, even though neither of the other two reasons is applicable. The SPV must fend for himself.

3.2 Customization Strategies

Customization incurs two types of design costs: architectural design cost and physical design cost. The former includes the design costs associated with architecting the custom system and any custom processors that it may contain, performing hardware-software partitioning, logic synthesis, design verification and, subsequently, system integration.

Physical design cost includes the design costs associated with the floorplanning, placement and routing, as well as the cost of creating a mask set.

While customization can be used to provide the very highest performance at the very lowest cost, SPVs employ a variety of strategies which span a spectrum. At one end, a custom architecture provides very high performance, but at very high architectural and physical design costs. At the other end of the spectrum, an OTS architecture provides much lower performance with the low design costs associated with software programming. A third choice, the OTS customizable system, provides an increasingly important compromise: the ability to use OTS parts, but requiring FPGA programming that is more complex and similar to physical design. Although the architectural cost and a part of the physical design cost must be borne, the mask set cost is avoided.

Minimizing architectural design cost. The key to minimizing architectural design cost is to reuse pre-existing designs to the extent possible. One strategy is to fix the system-level architecture, but to customize at the subsystem level. For instance, the system architecture may consist of a some specific interconnect topology, containing one or more specific OTS microprocessors plus one or more unspecified accelerators. The accelerators are defined by the application and must, therefore, be custom designs. However, only the architectural design cost of the 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 of the FUs, for instance, may be customized [2].

Minimizing physical design cost. An embedded system may either fit on a single chip, or be spread over multiple chips. For every part of a chip that has been customized, one must necessarily bear the cost of placement and routing. One can avoid this cost for the rest of the chip by using what is known as **hard IP**, i.e., subsystems that have been taken through physical design. Although the floorplanning, placement and routing for the chip as a whole must still be performed, the hard IP blocks are treated as atomic components, greatly reducing the complexity of this step. However for every chip that has been customized, even to a small extent, the entire cost of creating the mask sets for the chip must be borne.

At lower levels of VLSI integration, a system used to consist of multiple chips, of which only a few might have needed to be custom. Furthermore, the cost of creating a mask set was relatively low. The advent of SOC greatly reduces the cost of a complex embedded system. However, it comes with a disadvantage; *any* customization of the system, however tiny, requires a new mask set. Worse yet, the cost of creating a mask set is now in the hundreds of thousands of dollars, and rising.

Avoiding mask set costs. This is a powerful incentive to avoid VLSI design completely, motivating the notion of OTS, customizable SOCs or "reconfigurable hardware". The basic idea, as before, is to fix certain aspects of the system's and processors' architectures, and to allow the rest to be custom. The difference is that instead of implementing the custom accelerators and FUs using standard cells, they are implemented by mapping them, after performing logic synthesis, on to FPGAs. Accordingly, the OTS SOC contains the fixed portions of the system architecture, implemented as standard cells, but provides FPGAs instead of the custom processors, accelerators and FUs. By programming the FPGAs appropriately, the OTS SOC can be customized to implement a number of different custom system architectures. The entire architectural design cost for the custom subsystems is incurred, as are the costs of programming the FPGAs (similar in many ways to placement and routing), but the VLSI design costs are eliminated.

This is an extremely attractive approach when the desired custom system fits the system-level architecture of the OTS, customizable SOC. If not, the SPV must design a custom SOC. For high volume products, where application needs are well understood, and where high design cost and design time can be tolerated, customized SOCs provide higher performance at a lower cost than do FPGAs. However, its programmability allows an OTS FPGA to serve a far greater variety of immediate product needs and allows complex products to get to market more quickly and to evolve with changing application requirements. Further benefits of this approach are quick prototyping and field programmability in the event that the nature of the customization needs to be changed.

Note that our discussion of OTS customizable system here and in Section 2.3 are two views of the same thing. There, our view of it was as an alternative style of architecture. Here, our view of it is the more traditional one, as a way of implementing a hardware design.

4 Automation of Computer Architecture

Embedded computing, with its distinct set of requirements and constraints, is generating the need for large numbers of custom embedded systems. Whether they are implemented as custom SOCs or by using OTS, customizable SOCs, the architectural costs must be incurred. We believe that the need for mass customization, and its associated architectural costs, have brought into existence a new theme in computer architecture-the automation of computer architecture. We call this architecture synthesis in order to distinguish it from other forms of high-level synthesis such as behavioral synthesis [3, 4] and, to distinguish it from low-level synthesis such as logic synthesis [5].

4.1 When is Automation Important?

One could argue that for the foreseeable future, an automatically architected computer system can be expected to be less well-designed than a manually architected and tuned design. This statement is not as obviously true as it might sound; the ability of an automated system to evaluate thousands of disparate designs could quite conceivably yield a superior design. But if we accept this statement as true, the question that arises is under what circumstances it is desirable to use automation. We believe that there are at least three sets of circumstances that argue for automation.

The first one is when the desired volume of custom designs, stimulated by an explosion in the number of smart products, exceeds the available design manpower. The demand might be due to a large number of either application-specific or domain-specific designs. It might also be due to shortened product life cycles, either because the relevant standards are evolving, or because of competitive pressures in a consumer business. Automation addresses the problem by sharply increasing the aggregate design bandwidth.

Automation is also useful when time-to-market or time-to-prototype is crucial. Often, the product definition could not be anticipated far enough ahead of time to use manual design methodologies, perhaps because a relevant standard had not yet converged, or perhaps the functionality of a new product was not yet clearly understood. In such cases, the speed of automated techniques is of great value.

A third motivation for automation occurs when the expected volume of the custom design is too small to permit the product to be economical using a manual design process. Automation reduces the design cost (which must be amortized over the small volume), and can make such a product viable.

4.2 A Philosophy of Automation

A typical reaction to the notion of automating computer system design is that it is a completely unrealistic endeavor. Typically, the assumption underlying this reaction is that the automatic design system would emulate the human design methodology. That would, indeed, be a very hard problem in artificial intelligence, since human designers tend to invent new solutions to problems they encounter during design. We do not believe that one should try to build an architecture synthesis system that does this. Instead, our approach picks the most suitable design out of a large, possibly unbounded, denumerable design space. This space has to be large and diverse enough to ensure that there is a sufficient repertoire of good designs so that a best design, selected from the space, will closely match application needs.

Framework and Parameters. Since it is impractical to explicitly enumerate every feasible design, the space of designs is defined by a set of rules and constraints that must be honored by each design in the space. We call this a **framework**. Within a framework, some aspects of the design, such as the presence of certain modules and the manner in which they are connected are predetermined. Other aspects of the design are left unspecified. Of these, some can be derived once the rest have been specified. We refer to the latter as **parameters**. The **specification** of a design consists of binding the values of the parameters. From these, the derived aspects of the design are computed. Together, and in the context of the framework, they constitute a completely specified design. We define **construction** to be the process of deriving the detailed, completely specified design once the parameters are given.

Our philosophy for deciding what is a parameter, and what is not, is determined operationally. When we believe that we have an algorithmic way of determining certain design details in an optimal or near-optimal manner, we view their definition as part of implementation. When we have no clear way of determining important attributes of a design, and we use heuristic search to determine well-chosen values, we view them as parameters. After all design parameters are bound, a design is completely specified and the design can then be constructed. In the specification of a given design, it is often the case that not every combination of parameters is valid. For a system design to be valid, certain parameters of one subsystem must match the corresponding parameters of another subsystem. These are expressed as validity constraints involving the parameters that must match [6].

Components. Designs are constructed by assembling lower-level components, picked from a component library. Sometimes, these components are parameterized with respect to certain of their attributes; once the parameters are specified, a component constructor can be used to instantiate the corresponding component. The components, or their constructors, are designed and optimized manually. The components must fit into the framework and, must collectively provide all of the building blocks needed to construct any design.

In addition to its detailed design, each component must have associated information needed during the construction of the system design. Information needed to properly interface components into a broader design context includes a description of a component's functional capability, a description of a component's input and output wiring needs, and a description of a component's externally visible timing and resource requirements. Components are often described as a network of lower-level components forming a design hierarchy.

A Paradigm for Automation. Typically, one has multiple evaluation metrics in mind (such as cost, performance and power) when picking a good design. Thus, finding an optimal design involves a multi-objective optimization task. A design is said to be a **Pareto-optimal design**, or a Pareto design for short, if there is no other design that is at least as good as it with respect to every evaluation metric, and better than it with respect to at least one evaluation metric. The set of all Pareto designs is the **Pareto set**, or the Pareto for short.

We automatically find a Pareto set using three interacting modules. The **spacewalker** explores the space of possible designs, looking for the Pareto-optimal ones. The space of possible designs is specified to the spacewalker by the user, who provides a range of values for each parameter. The **design space** is the Cartesian product of the sets of values for the various parameters. It defines the space of designs that the spacewalker must explore. At each step in the search, the spacewalker specifies a design by binding parameters. A **constructor** can take a design, as specified by the spacewalker, and construct a hardware realization of the chosen design. The effects of a component binding are evaluated using an **evaluator** that determines the suitability of the spacewalker's choice. Evaluation is most accurately performed by first executing the constructor for a design, with appropriately bound parameters, to produce a detailed design. Then the evaluators use the detailed design to compute the evaluation metrics. When the cost of constructing candidate detailed designs is excessive, approximate evaluation metrics can sometimes be quickly estimated directly from design parameters. The evaluation process uses multiple tools including compilers, simulators, and gate count estimators.

At each step in the search, the spacewalker invokes the constructor and evaluators to determine whether, in the context of the designs evaluated thus far, the latest design is Pareto-optimal. If the design space is small, the spacewalker may use an exhaustive search. Otherwise, at each step, it uses the evaluation metrics, possibly other statistics relating to the design, and appropriate heuristics to guide it in taking the next step in its search. The goal in this case is to find all, or most, of the Pareto-optimal designs while having examined a very small fraction of the design space. Spacewalking is hierarchical when an optimized system is designed using a spacewalking search and components of the system are treated as sub-systems that are in turn optimized using lower-level spacewalking searches.

A framework restricts the generated design to a subset of all possible designs that a human might have created. However, it is precisely from this that the power of a framework arises. It is difficult to conceive of how one could create constructors and evaluators capable of constructing and evaluating any possible design. Yet, without constructors and evaluators, automation would be impossible. The limits placed by a framework, on the types of designs that have to be evaluated and constructed, are crucial to making automation possible. The challenge, when designing a framework, is to choose one that is large and diverse enough to contain good designs, while at the same time retaining the ability to evaluate and construct every design in that framework.

5 The PICO Architecture Synthesis System

In order to illustrate our automation philosophy, we briefly describe PICO (Program In, Chip Out), our research prototype of an architecture synthesis system for automatically designing custom, embedded computer systems. It employs our paradigm for automation, in a hierarchical fashion, four times over. PICO takes an application written in C, automatically architects a set of Pareto-optimal system designs, and emits the structural VHDL for them. Currently, optimality is defined by two evaluation metrics: cost (gate count or chip area) and performance (execution time). PICO explores trade-offs between the ways in which silicon area can be utilized in such a system, presenting to the user a set of Pareto-optimal system designs. In the process, PICO does hardware-software co-design-partitioning the given application between hardware (one or more custom NPAs) and software (on a custom EPIC/VLIW processor[1] [7]). PICO also retargets a compiler to each custom VLIW processor; we call this processor-compiler co-design.

5.1 System Synthesis

At the system level, PICO's task is to identify the Pareto-optimal set of custom, application-specific, embedded system designs for a given application. Each system that PICO designs consists of a custom VLIW processor and a custom, two-level cache hierarchy. The cache

[1] EPIC (Explicitly Parallel Instruction Computing) is a generalization of VLIW. For convenience, in the rest of this report we use the term VLIW to include EPIC as well.

hierarchy consists of a first-level data cache (Dcache), a first-level instruction cache (Icache), and a unified second-level cache (Ucache). In addition, the system may contain one or more custom NPAs that work directly out of the second-level cache. PICO exploits the hierarchical structure of this design space. Within PICO's framework, a system design consists of a VLIW processor, one or more NPAs, and a cache hierarchy. Accordingly, PICO decomposes the system design space into smaller design spaces, one for each of these major subsystems. The components that PICO uses to create a system-level design are the custom, application-specific VLIW processors, NPAs and cache hierarchies that are yielded by the spacewalkers and constructors for the various subsystems, as discussed below. The system-level parameters are the union of the parameters for the VLIW processor, the NPAs, and the cache hierarchy.

The system design space typically contains millions of designs, each requiring an hour or more of profiling, compilation, synthesis and simulation time. An exhaustive search of the design space is infeasible. Instead, PICO exploits the hierarchical structure of the design space. The basic intuition is that Pareto-optimal systems are composed out of Pareto-optimal component subsystems [6]. Accordingly, the system-level spacewalker invokes the subsystem-level spacewalkers to get the Pareto-optimal sets of subsystem designs. The set of all combinations of Pareto-optimal subsystems is far smaller that the original design space. In the simplest case, the system-level spacewalker would consider all these combinations, evaluate them, and discard all that are not Pareto-optimal system designs. But due to validity constraints, not all combinations are valid. Therefore, the system-level spacewalker requires that each subsystem-level spacewalker return not just a single Pareto set, but rather a set of parameterized Pareto sets. When composing subsystems, the system-level spacewalker enforces validity constraints by only combining designs from compatible, parameterized Pareto sets. For such compatible Pareto sets, it considers all combinations of subsystems, evaluates them, and discards all that are not Pareto-optimal system designs.

For each loop nest in the application, that is a candidate to be implemented in hardware, the spacewalker examines both options. If there are N such loop nests, the spacewalker explores 2N system architectures, from those that have no NPAs at all, to those in which every loop nest has been implemented as a NPA. For each system architecture, it comes up with the Pareto set as described above. It then forms the union of these Pareto sets and finds the Pareto-optimal designs within this union. This final Pareto set contains all designs of interest for the application. Typically, at low performance levels, the Pareto-optimal designs contain no NPAs. Conversely, at sufficiently high levels of performance, all of the loop nests may be implemented as NPAs. In this manner, the system-level spacewalker makes different hardware-software partitioning decisions for Pareto optimal systems with varying cost and performance.

The system-level constructor utilizes the constructors for the VLIW, NPA and cache hierarchy subsystems to construct the subsystem designs. It then glues these subsystem designs together by synthesizing the appropriate hardware and software interfaces between the VLIW processor and the NPAs. Likewise, the system-level evaluators make use of the subsystem-level evaluators. System designs are evaluated by adding the costs and the execution times, respectively, of the component subsystems.

5.2 VLIW Synthesis

PICO-VLIW is the PICO subsystem that designs custom, application-specific VLIW processors and generates a parameterized set of Pareto sets [8]. In addition, it retargets Elcor (PICO's VLIW compiler) to each new processor so that it can compile the C application to that processor. The processors currently included within PICO-VLIW's framework encompass a broad class of VLIW processors with a number of sophisticated architectural and micro-architectural features [9, 10]. A complete specification of a VLIW processor within this framework involves hundreds of detailed decisions. If all of these were parameters, it would result in an extremely unwieldy design space exploration task. Our choice of the interface between the spacewalker and the VLIW constructor involves a delicate balance between giving the spacewalker adequate control over the architecture, without bogging it down by requiring it to specify all details. Our compromise is that the parameters that the spacewalker must specify are limited to the sizes and types of register files, the operation repertoire, and the requisite level of ILP concurrency. Thereafter, it is the job of the VLIW constructor to make the remaining detailed design decisions.

The number of parameters needed to specify a VLIW design is still relatively large. Consequently, even if the range for each parameter is small, the size of the design space can be extremely large. Furthermore, the evaluation of the performance of a VLIW design is time-consuming, since it involves compiling a potentially large application. The spacewalker, therefore, explores the design space using sophisticated search strategies and heuristics to prune the design space based on previously evaluated processor designs.

For each set of parameters generated by the spacewalker, the VLIW constructor designs the architecture and micro-architecture of the specified VLIW processor, including the execution datapaths, the instruction format and the instruction unit, and emits structural

VHDL. It also automatically extracts that part of the **machine-description database** (**mdes**) [9] that drives Elcor during scheduling and register allocation. The VLIW constructor uses RTL components from PICO's macrocell library (such as adders, multiplexers and register files) to synthesize the VLIW processor. In addition to the gate-level design, each component has associated with it information regarding its area, gate count, and degree of pipelining. Functional unit macrocells also are annotated with the set of opcodes that they can execute.

The cost evaluator estimates the chip area and gate count for the design using parameterized formulae for area and gate count that are attached to each component in the macrocell library. These formulae are calibrated against actual designs. The performance evaluator estimates the execution time of the given application on the newly designed processor using Elcor and PICO's retargetable assembler. Both are automatically retargeted by supplying them with the mdes for the target processor. The schedule created by Elcor, along with the profiled frequency of execution of each basic block, suffices to estimate the execution time. The object code generated by the assembler serves two objectives during design space exploration. One is to evaluate the code size and its impact upon the cost of main memory. The other is to permit an estimate of its effect upon the Icache and Ucache miss rates and the resulting impact on execution time.

5.3 NPA Synthesis

PICO-N is the PICO subsystem for designing NPAs customized to a given loop nest and for obtaining a parameterized set of Paretos for such NPAs [11]. A design in the NPA framework consists of a synchronous, one- or two-dimensional array of customized processing elements (PEs) along with their local memories and interfaces to global memory, a controller, and a control and data interface to a host processor. Each PE is a datapath with a distributed register file structure. Each register file is a FIFO with random read access. Interconnections between the FIFOs and FUs exist only as needed by the computation, resulting in a sparse and irregular interconnect structure with connections to only some of the FIFOs' elements. PICO-N also synthesizes the code required to make use of the NPA.

The design parameters are the number of PEs, the initiation interval (II) between starting successive iterations on any single PE, and the amount of memory bandwidth to the second-level cache. Since the number of parameters is small, the spacewalker performs an exhaustive search through the design space defined by the ranges of the three parameters. Because the precise geometry of the array of PEs is not specified as a parameter, the spacewalker steps through all one- and two-dimensional array geometries which have the specified number of PEs. This serves as an additional parameter for the constructor. The results of this search are used to create a set of Paretos, each one parameterized by the number of memory ports that the NPA has to the second level cache.

The NPA constructor starts off with a loop nest expressed as a sequential computation working out of global memory. It first tiles the loop nest, creating a new set of sequential outer loops, that will run on the VLIW processor, along with an inner loop nest with fewer iterations (the tile), that will be executed in parallel by the NPA. This transformation allows the constructor to not exceed the available global memory bandwidth, by performing register promotion in the inner loop nest, while minimizing the cost of the additional registers that this entails. Smaller tiles result in lower hardware cost but higher memory traffic. The constructor uses constrained combinatorial optimization to minimize the tile volume without exceeding the memory bandwidth allocated to the NPA. Next, the constructor transforms the inner loop nest into multiple, identical, synchronously parallel computations, one per PE. It does so by assigning a PE and a start time to each iteration in the tile while honoring data dependences between iterations. Furthermore, from the perspective of each processor, this iteration schedule is required to start an iteration every II cycles. This allows the constructor to express the computation on each PE as a single loop that performs all of the iterations assigned to that PE.

This loop is used to synthesize a single PE. Using user-specified pragmas regarding the requisite bit widths of variables, the constructor infers the minimum width requirements for all variables, temporaries, and operations. This allows the width of every register, FU and datapath to be minimized. A minimum-cost set of FUs are allocated and the operations of the loop body are assigned to these functional units and scheduled in time. Elcor is used to perform software-pipelining of the loop at the specified II using a variety of heuristics to minimize hardware cost. At this point the hardware for one PE is materialized, using the RTL components in the macrocell library. Register files, in the form of FIFOs with random read access are allocated to hold temporary values. Interconnections between the FIFOs and FUs are created only as needed, resulting in a sparse and irregular interconnect structure with connections to only certain of the FIFOs' elements.

An NPA consists of multiple instances of the PE, configured as an array. As many copies of the PE are created as specified by the design parameter, and interconnected in the specified geometry. The controller and the global memory interface are generated. The NPA, including its registers and array-level local memories, is accessed via a local memory interface by the VLIW processor. It may be initialized and examined using this interface. Finally, structural VHDL for the NPA is emitted.

The NPA constructor also performs some software synthesis. It takes the loop nest after tiling, with its additional outer loops, and removes the inner loop nest that has now been implemented as hardware. In its place, it generates the code that will invoke the NPA after making the appropriate initializations via the NPA's local memory interface. This new loop nest is inserted back into the application in place of the loop nest that was presented to PICO-N. Instead of executing the loop nest on the VLIW processor, the application will now trigger the computation on the NPA.

The cost evaluator estimates the chip area and gate count for the NPA, as described earlier for the VLIW processor. The performance evaluator for the NPA (which executes a predictable loop nest) is estimated using a formula instead of via simulation.

5.4 Cache Hierarchy Synthesis

The third major PICO subsystem automatically generates a parameterized set of Pareto sets for cache hierarchies that have been customized to the given application [12]. A design within the cache hierarchy framework consists of a first-level Dcache, a first-level Icache and a second-level Ucache. Just as at the system level, PICO decomposes the cache hierarchy design space into smaller design spaces for the Dcache, Icache and Ucache, respectively. Each of the three caches-Icache, Dcache and Ucache-is parameterized by the number of sets, the degree of associativity, the line size, and the number of ports. A valid cache hierarchy design must have parameters that are compatible as specified by the validity constraints, e.g. the porting on the Ucache must at least be equal to the Dcache porting (if data fetched from the Ucache is permitted to bypass the Dcache). A final parameter, dilation, is an attribute of the code size for each VLIW processor. This parameter determines not the design of the cache hierarchy, but rather the performance of the Icache and Ucache.

The spacewalker takes advantage of the fact that the cache hierarchy design space is decomposed into three smaller design spaces corresponding to the Icache, Dcache and Ucache, respectively. For each of these design spaces, a parameterized set of Pareto sets is formed. Each design space's set contains member Pareto sets which are formed separately for each setting of the values of the parameters that participate in validity constraints The recomposition step uses these Paretos to form parameterized Pareto sets for the overall cache hierarchy, while enforcing validity constraints by only considering those combinations of Icache, Dcache and Ucache Pareto sets that are compatible.

The costs of the Icaches, Dcaches and Ucaches are evaluated using parameterized formulae. Trace-driven simulation is used to evaluate the miss rates, using the Cheetah cache simulator [13] which exploits inclusion properties between caches to simulate, in a single pass through the address trace, a range of cache designs with a common line size. This trace-driven cache simulation is done once for a reference VLIW processor. To a first order of approximation, the Dcache miss rate is assumed to be unaffected by the details of the VLIW processor. But this is not the case for the Icache and Ucache miss rates. Analytic techniques that use the dilation parameter, combined with interpolation of the reference processor's miss rate, are used to estimate the performance of these caches.

6 Conclusions

System-on-chip levels of VLSI integration are causing the center of gravity of the computer industry to move into embedded computing. The driving application for embedded computing will be a rich diversity of innovative smart products which depend, for their functionality, on the availability of extremely high-performance, low-cost embedded computer systems.

Successful computer architecture results from achieving a careful balance between the opportunities afforded by the latest technologies, on the one hand, and the requirements of the market, product and application, on the other. The opportunities, needs and constraints of embedded computing are quite distinct from those of general-purpose computing. This creates a new playing field and will lead to substantially different computer architectures, at both the system and the processor levels. Embedded architectures will be far more special-purpose, heterogeneous and irregular in their structure. There never was such a thing as the "one right architecture", even for general-purpose computing. This is even truer for embedded computing, which will trigger a renaissance in system and processor architecture. Custom and customizable architectures will assume a new importance.

Furthermore, we believe that the very large number of custom architectures required, due to the expected explosion in the number of smart products, will introduce a new theme into computer architecture: the automation of computer architecture. Our experience with PICO is that this is a perfectly practical and effective endeavor. The resulting designs are quite competitive with manual designs, but are obtained one or two orders of magnitude faster.

7 Acknowledgements

The ideas and opinions expressed in this report have been greatly influenced by our colleagues, past and present, in the Compiler and Architecture Research group at HP Labs: Rob Schreiber, Shail Aditya, Vinod Kathail, Scott Mahlke, Santosh Abraham, Darren Cronquist, Mukund Sivaraman, Greg Snider, Sadun Anik and Richard Johnson. They, along with the authors, are responsible for the development of the PICO system.

References

1. A. K. Sharma. Programmable Logic Handbook: PLDs, CPLDs, and FPGAs. (McGraw Hill Companies, 1998).

2. R. E. Gonzalez. Xtensa: a configurable and extensible processor. IEEE Micro 20, 2 (2000), 60-70.

3. D. W. Knapp. Behavorial Synthesis: Digital System Design Using The Synopsys Behavorial Compiler. (Prentice Hall PTR, Upper Saddle River, New Jersey, 1996).

4. J. P. Elliot. Understanding Behavorial Synthesis: A Practical Guide to High-Level Design. (Kluwer Academic, 1999).

5. W. F. Lee. VHDL: Coding and Logic Synthesis with Synopsys. (Academic Press, 2000).

6. S. Abraham, B. R. Rau and R. Schreiber. Fast Design Space Exploration Through Validity and Quality Filtering of Subsystem Designs. HPL Technical Report HPL-2000-98. Hewlett-Packard Laboratories, August 2000.

7. M. S. Schlansker and B. R. Rau. EPIC: Explicitly Parallel Instruction Computing. Computer 33, 2 (2000), 37-45.

8. S. Aditya, B. R. Rau and V. Kathail. Automatic architectural synthesis of VLIW and EPIC processors. Proc. International Symposium on System Synthesis, ISSS'99 (San Jose, California, November 1999), 107-113.

9. B. R. Rau, V. Kathail and S. Aditya. Machine-description driven compilers for EPIC and VLIW processors. Design Automation for Embedded Systems 4, 2/3 (1999), 71-118.

10. V. Kathail, M. Schlansker and B. R. Rau. HPL-PD Architecture Specification: Version 1.1. HPL Technical Report HPL-93-80 (R.1). Hewlett-Packard Laboratories, February 2000.

11. R. Schreiber, S. Aditya, B. R. Rau, V. Kathail, S. Mahlke, S. Abraham and G. Snider. High-level synthesis of nonprogrammable hardware accelerators. Proc. International Conference on Application-Specific Systems, Architectures, and Processors (ASAP 2000) (Boston, Massachussetts, July 2000), 113-124.

12. S. G. Abraham and S. A. Mahlke. Automatic and efficient evaluation of memory hierarchies for embedded systems. Proc. 32nd Annual International Symposium on Microarchitecture (MICRO '99) (November 1999), 114-125.

13. R. A. Sugumar and S. G. Abraham. Efficient Simulation of Caches under Optimal Replacement with Applications to Miss Characterization. Proc. ACM SIGMETRICS (1993), 24-35.