Automatic architectural synthesis of VLIW and EPIC processors

Shail Aditya B. Ramakrishna Rau Vinod Kathail
{aditya,rau,kathail}@hpl.hp.com

Hewlett-Packard Laboratories
1501 Page Mill Road, MS 3L-5, Palo Alto, CA 94304

Abstract

This paper describes a mechanism for automatic design and synthesis of very long instruction word (VLIW), and its generalization, explicitly parallel instruction computing (EPIC) processor architectures starting from an abstract specification of their desired functionality. The process of architecture design makes concrete decisions regarding the number and types of functional units, number of read/write ports on register files, the datapath interconnect, the instruction format, its decoding hardware, and the instruction unit datapath. The processor design is then automatically synthesized into a detailed RTL-level structural model in VHDL along with an estimate of its area. The system also generates the corresponding detailed machine description and instruction format description that can be used to retarget a compiler and an assembler respectively. All this is part of an overall design system, called Program-In-Chip-Out (PICO), which has the ability to perform automatic exploration of the architectural design space while customizing the architecture to a given application and making intelligent, quantitative, cost-performance tradeoffs.

1. Introduction

VLIW (Very Long Instruction Word) processors have started establishing themselves as the processor of choice in high performance embedded computer systems, especially in situations where an efficient compiler for a high level language is available. Although a fair amount of work has been done on providing the capability to automatically design the architecture of a sequential, application-specific instruction-set processor (ASIP) – primarily a matter of designing the opcode repertoire – there has been relatively little work in the area of automatic architecture synthesis of VLIW processors or, for that matter, processors of any kind that provide significant levels of instruction-level parallelism (ILP). The work which has been done tends to focus largely upon the synthesis of a VLIW processor's datapath [5, 6, 8]. The automatic design of a non-trivial instruction format, and the synthesis of the corresponding instruction fetch and decode micro-architecture have not been addressed for VLIW processors. And yet, it is these issues that consume the major portion of a human designer's efforts during the architecture and micro-architecture phases of a VLIW design project.

In this paper, we present a fully automated system for designing the architecture and micro-architecture of VLIW processors and their generalization, EPIC (Explicitly Parallel Instruction Computing) processors[1]. We refer to this process as *architecture synthesis* to distinguish it from behavioral or logic synthesis which are at a lower level. In addition to the well understood features of the VLIW style of architecture, the space of processors that we are interested in exploring includes features such as predication, control and data speculation, rotating registers, and explicit source and destination specifiers for load and store operations at various levels of the memory hierarchy [9]. Processors with these features have the ability to exploit high degrees of compiler-specified ILP both in numerically-intensive applications as well as in applications that are intensive in branches and pointer-based memory references.

The architecture synthesis system that we describe in this paper is part of PICO (Program-In-Chip-Out), a broader system synthesis and design exploration tool which performs hardware-software co-synthesis. In addition to the custom VLIW processor, PICO may design one or more non-programmable, systolic-array co-processors (ASICs) and a two-level cache hierarchy to support these processors. It partitions the given application between hardware (the systolic arrays) and software, compiles the software to the custom VLIW, and synthesizes the interface between the processors. We refer to PICO's VLIW design capability as PICO-VLIW which is the subject of this paper.

The major contribution of the work reported here is not necessarily in the specific heuristics used but in establishing

[1] For the sake of brevity, we use the term VLIW to include EPIC as well in the rest of this paper.

1080-1820/99 $10.00 © 1999 IEEE

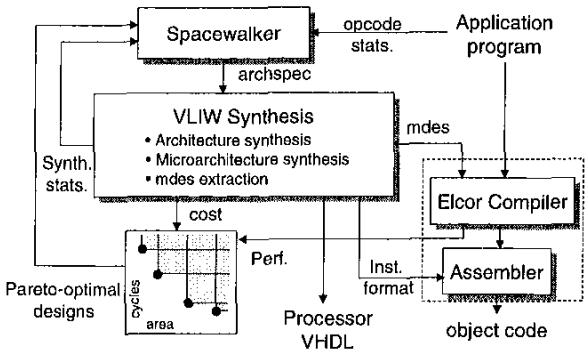

Figure 1. The PICO-VLIW design system.

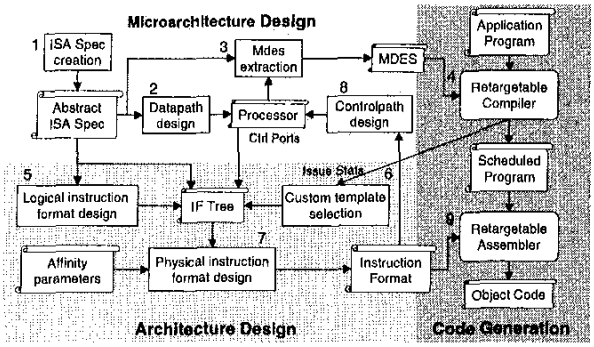

Figure 2. Design flow in PICO-VLIW.

a framework which formalizes and makes algorithmic what has thus far been an ad hoc, manual process.

2. Overview of the PICO-VLIW System

In PICO-VLIW, we decompose the process of automatically designing an application-specific VLIW processor into three closely inter-related sub-systems as shown in Figure 1. The first sub-system is our design space explorer, the **Spacewalker**, whose responsibility is to search for the Pareto-optimal architectures, *i.e.*, those architectures whose implementations are either cheaper or faster (or both) than any other architecture. In order to do this efficiently, the Spacewalker uses sophisticated search strategies and heuristics that are, however, beyond the scope of this paper.

The second sub-system is the **VLIW architecture synthesis** sub-system whose responsibility is to take the abstract architecture specification generated by the Spacewalker and to create the best possible concrete architecture and micro-architecture, as well as a machine-description database used to retarget the compiler. The system outputs a RTL-level, structural VHDL description of the processor and estimates the chip area consumed by it.

The third sub-system consists of **Elcor**, our retargetable compiler for VLIW processors whose operation repertoire is a subset of the HPL-PD repertoire [9], and a retargetable assembler. Both are automatically retargeted by supplying the machine-description database. Elcor's responsibility is to generate the best possible code for the application on the processor designed by the VLIW architecture synthesis subsystem, and to evaluate its performance by counting the number of cycles taken to execute the program. The area and execution time estimates are then used by the Spacewalker to guide the next step of its search.

PICO-VLIW design flow. The design flow within PICO-VLIW may be divided into three major activities: architecture design, micro-architecture design and code generation as shown in Figure 2. The figure also shows the various design steps numbered in design flow sequence, and the dependence relationships among them.

In manual VLIW design as well as related work on VLIW synthesis [6, 7], the starting point is the concrete ISA which consists of a specification of the register file structure and an instruction format. We take a different approach, since we view the concrete ISA as an overly-constrained input specification. Instead, we start with an abstract architecture specification (step 1), which specifies the desired levels of concurrency and the opportunities for resource sharing, but which leaves the detailed decisions as to how best to share register ports and instruction bits to the datapath and the concrete ISA design steps, respectively. This allows PICO-VLIW to go about the design in an unconventional order: first, to design a datapath that is consistent with the requirements of the abstract architecture specification (step 2); next, to design a concrete ISA in the light of the control ports of the datapath (steps 5-7), and to then design the controlpath (step 8), *i.e.*, the instruction prefetch, alignment and decode hardware. By designing the concrete ISA after the datapath, we are able to achieve better trade-offs between code size and the complexity of the controlpath.

In the following sections, we focus our attention on the VLIW synthesis sub-system of PICO-VLIW (steps 1-3, 5-8). Further details of each step are provided in [1, 2].

3. Abstract architecture specification

Architecting a VLIW processor is considerably more complex than a sequential one. In addition to picking an operation repertoire, one must specify the extent and nature of the processor's ILP. A VLIW processor, when designed by an expert architect, exhibits certain features which we want PICO-VLIW to emulate. For example, the processor may use heterogeneous functional units – although one might include the ability to issue two adds every cycle, which requires two integer units, only one unit may be capable of shifting and the other unit able to do multiplication. The register file ports may be shared – a multiply-add opera-

tion, which requires three register read ports, may be accommodated by "borrowing" one of the ports of another functional unit which cannot, now, be used in parallel with the multiply-accumulate. Likewise, instruction bits may be shared – a load or store operation, which requires a long displacement field, might use the instruction bits that would otherwise have been used to specify an operation on some other functional unit. In order for PICO-VLIW to yield well-architected processors, the Spacewalker needs to be able to specify such architectures to the VLIW synthesis sub-system.

Our choice of the interface between the Spacewalker and the VLIW synthesis sub-system is called the *abstract architecture specification* (*archspec* for short) which provides a delicate balance between giving the Spacewalker adequate control over the architecture, without burdening it with the need to specify a detailed instruction format. Through the archspec, the Spacewalker specifies (Figure 2, step 1) the register files of the target machine, its operation repertoire and the requisite level of ILP in terms of concurrent operation groups, and the opportunities for sharing register ports and instruction bits in terms of exclusion groups. We will describe these components shortly. Thereafter, the Spacewalker relies upon the concrete ISA design, the datapath design and the controlpath design steps to use these opportunities while honoring the requisite level of concurrency.

As an example, a simple 2-issue machine is given below:

Register Files

Name	Width	Registers/Literals	Virtual File
gpr	32	r0,...,r31	I
pr	1	p0,...,p15	P
lit	10	[-512,511]	L

Operation Groups

Name	Operations	Operation Format
addsub	ADD,SUB	pr ? gpr, gpr : gpr
mult	MPY	pr ? gpr, gpr : gpr
multadd	MPYADD	pr ? gpr, gpr, gpr : gpr
loadinc	LI	pr ? gpr : gpr, gpr
loaddisp	LM	pr ? gpr, lit : gpr
storedisp	SM	pr ? gpr, gpr, lit :

Exclusion Groups

Name	Op Groups
EG0	addsub mult multadd
EG1	loadinc loaddisp storedisp
EG2	addsub mult loadinc
EG3	multadd loaddisp storedisp

Each *Register File* specified in the archspec identifies its width in bits, the registers it contains, and a virtual file specifier that specifies the types of data it can hold. An immediate literal field within the instruction format of an operation is also considered to be a (pseudo) register file consisting of

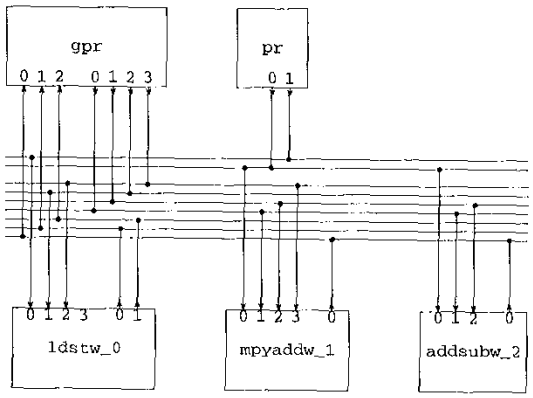

Figure 3. A datapath example.

a number of "literal registers" that have fixed values. The example shows that the above machine has a 32-bit general purpose register file "gpr", a 1-bit predicate register file "pr" and a 10-bit literal (pseudo) register file "lit".

The various instances of HPL-PD opcodes for a given machine are grouped into *Operation Groups* (*opgroups* for short). This example specifies six operation groups, implementing the operations add/subtract, multiply, multiply-add, load with post-increment, and load/store with displacement respectively. Each operation group also specifies one or more *Operation Formats* shared by all the opcodes within the group. These specify the desired input/output operand connectivity to the register files of the machine. For predicated operations, a separate predicate input is specified.

In addition to the desired opcode repertoire, the archspec also abstractly specifies the amount of parallelism to be supported in the target machine. By definition, all opcode instances within an operation group are mutually exclusive while, by default, those across operation groups are allowed to execute in parallel. The parallelism of the machine may be further constrained by placing two or more operation groups into *Exclusion Groups* as shown above. All operation groups within an exclusion group are deemed to be mutually exclusive, a fact that can be exploited by the datapath design step to share hardware resources such as functional units, register file ports and buses. In the above example, the exclusion groups "EG0" and "EG1" serve to represent the notion of one arithmetic and one memory functional units each, while "EG2" and "EG3" allow further sharing of register file ports and instruction format bits as shown later.

4. Datapath design

The datapath designed for the machine specified in Section 3 is shown pictorially in Figure 3 which also illustrates our general design scheme. The datapath consists of one or more functional units selected on the basis of the desired

operation functionality connected to the specified register files via multiple buses. Each bus corresponds to a register file read or write port. There are several important design decisions to be made at this step (Figure 2, step 2) that are outlined below. The details are provided in [1].

Functional unit allocation. The first step in datapath synthesis is to select a set of functional unit macrocells from the database that can together implement all the operations specified in the archspec such that the specified ILP constraints are met and the total cost (area or gates) is minimized. Our strategy is to formulate it as a clique finding problem on the graph of exclusion relationships among the operation groups and then determining a set of minimum cost functional units that cover all cliques.

Register file port allocation. VLIW processors typically need multi-ported register files in order to cater to the needs of multiple, concurrently-executing functional units. Multi-ported register files are very expensive (in terms of area), and therefore a novel aspect of our system is that we automatically determine the minimum number of such read and write ports by taking into account the exclusion constraints among the operation groups in the archspec.

For each type of port (read/write) to a register file, we formulate a separate resource allocation problem. In each formulation, the desired port connectivity is determined by consulting the operation formats of the various operation groups assigned to each functional unit. A conflict graph among the requesting functional unit ports is constructed based on their concurrent use (*e.g.*, two operands of a binary operation), and mutual exclusions specified in the archspec. Each problem is then solved using a variation [1] of the graph coloring approach by Chaitin [3]. In our example (refer Figure 3), the two arithmetic macrocells share all of their register file ports. More interestingly, input 3 of the "mpyaddw_1" and input 2 of the "ldstw_0" macrocells also share a register file port because an exclusion was specified between their operations groups.

Register file and interconnect generation. In the final step of datapath synthesis, we instantiate each register file with the appropriate number of read and write ports and generate the interconnect between the register file ports and the various functional unit ports as specified by the port allocation.

5. Mdes extraction

Once the datapath has been designed, our system automatically extracts a compiler-oriented view of it (Figure 2, step 3) in the form of a *machine description* (*mdes* for short) [10]. This is then used to re-target our VLIW compiler, Elcor, to the target machine (Figure 2, step 4). Our system extracts a non-structural and operation-centric mdes from the archspec and the datapath by combining the individual mdes contributions of the various functional units and augmenting them with the resource and latency constraints of the surrounding hardware. The details are provided in [1].

6. Concrete ISA design

The concrete ISA consists of a specification of the register file structure and the instruction format. The former is taken directly from the archspec, whereas the latter is generated automatically by the PICO-VLIW system. A novel feature of our approach is the distinction we make between the logical and the physical instruction formats, which is discussed below. Further details appear in [2].

Logical instruction format design. The *logical instruction format* is equivalent to what one traditionally thinks of as an instruction format: a set of instruction templates each consisting of one or more operation groups that can be issued simultaneously. Each operation group has one or more operation formats, each of which is a set of *instruction fields* encoding the opcode and the various operands. Also, each operation group may appear in multiple templates, yielding multiple *instances* of each field.

The objective of the logical instruction format design step (Figure 2, step 5) is to avoid code wastage due to no-ops specified in the program that may result from the use of a simplistic instruction format. This code wastage comes in two forms. First, a single instruction template, which contains an operation slot dedicated to each functional unit macrocell (akin to horizontal microcode) is quite wasteful since the archspec and the datapath may not allow all of those operations to be issued simultaneously. We address this by designing multiple instruction templates, each of which is only capable of specifying a set of operations that can, in fact, be issued simultaneously. Our design strategy is to treat each clique of concurrent operation groups specified in the archspec as a separate instruction template.

Second, a given instruction, as scheduled by the compiler, may not have enough ILP to use all the available slots of its template. The unused operation slots must specify a no-op, again leading to code wastage. We address this by providing additional, narrower templates which correspond to statistically frequent combinations of operations in the scheduled code. Identifying these custom templates (Figure 2 step 6) entails invoking the scheduler, which in turn requires that the datapath has already been designed.

As an example, the exclusions specified in the archspec of Section 3 stipulates two instruction templates as shown in Figure 4(a). Each template consists of a consume-to-end-of-packet (EOP) bit which is used for aligning branch targets [1, 2], a template selector field, and one or more concurrent slots encoding the various operation groups.

Physical instruction format design. The *physical instruction format* allows the fields within each template to be po-

sitioned in any convenient order, but an order that is fixed for that template. Furthermore, an individual field is also permitted to consist of a discontiguous set of bit positions. One of the objectives of the physical instruction format design step (Figure 2, step 7) is to exploit these additional degrees of freedom with a view towards minimizing the width of each instruction template. This is done by assigning the same or overlapping bit positions to fields that cannot appear simultaneously in the same instruction while ensuring that fields which can be present simultaneously, are assigned disjoint bit positions.

A second, somewhat conflicting, objective at this step is to minimize the complexity of the decode and distribution network that lies between the instruction register and the datapath control ports. This is done by minimizing, for each control port, the number of distinct bit positions, across all of the templates, at which the instruction fields controlling the given port are to be found. Once again, this requires that the datapath has already been designed.

The physical instruction format is designed using the instruction format tree (*IF-tree* for short) data-structure which is a hierarchical representation of the grammar of an instruction for the target architecture. The leaves of the tree are the logical instruction fields for which physical bit positions need to be allocated. The IF-tree is used to compute a conflict matrix among the instruction fields where two fields are said to *conflict* if they can be present simultaneously in the same logical template, and therefore must be assigned disjoint bit positions. The allocation algorithm we use is a variant [2] of Chaitin's graph coloring algorithm [3] where instruction bits are resources and each requesting instruction field may request multiple bits. Heuristics are used to reduce the overall template width and the decode complexity by packing the instruction fields to the left (leftmost allocation), assigning contiguous bit positions to multi-bit fields (contiguous allocation), and aligning instruction fields corresponding to the same control port to the same bit position (affinity allocation). Finally, the bits of each template are rounded up to the next multiple of a fixed quantum size (Q_I) in order to simplify its alignment in the memory and the instruction register as discussed in Section 7.

The physical format for the template T1 of our example machine is shown in Figure 4(b). This template consists of two concurrent operation groups "multadd" and "loadinc" whose various instruction fields have been assigned bit positions as shown in the figure. Note that the SRC3 field of the "multadd" operation group is positioned in the midst of the bits corresponding to the "loadinc" operation group because it has affinity with the SRC2 field of the "storedisp" operation group in template T0. This was due to the fact that these two fields drive the same register file address port, which in turn was a result of specifying a exclusion "EG3" between the two operation groups in the archspec.

Template	EOP	TSel	OpGroups	OpGroups
T0	0	1	addsub mult	loaddisp storedisp
T1	0	1	multadd	loadinc

(a)

multadd: pr ? gpr, gpr, gpr : gpr

Template	PRED1	SRC1	SRC2	SRC3	DEST1
T1	5···8	9···13	14···18	35···39	19···23

loadinc: pr ? gpr : gpr , gpr

Template	PRED1	SRC1	DEST1	DEST2
T1	26···29	30···34	45···49	40···44

storedisp: pr ? gpr, gpr, lit :

Template	PRED1	SRC1	SRC2	SRC3
T0	26···29	30···34	35···39	40···49

(b)

Figure 4. Example instruction templates.

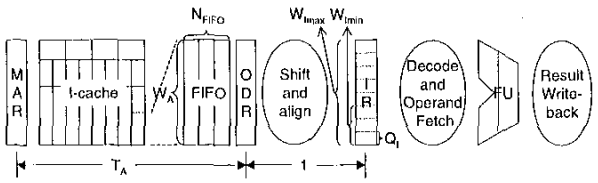

Figure 5. The instruction fetch pipeline.

7. Controlpath design

We partition the problem of controlpath design into two major components: the instruction sequencer and the instruction fetch pipeline. The sequencer design is dependent upon the presence/absence of features such as exception and interrupt handling, error recovery, branch prediction etc. but is largely independent of the instruction-set architecture of the machine. Therefore, we assume that the appropriate set of sequencer macrocells is available in our macrocell database. Below, we address the design of the instruction fetch pipeline (Figure 2, step 8) consisting of the following components (refer Figure 5).

Instruction Cache. For purpose of the instruction pipeline design, the cache is characterized by its access time (T_A) and the size of an instruction packet (W_A) which is the unit of instruction fetching from the processor side.

Instruction prefetch buffer. An instruction packet is fetched from the instruction cache and brought into a FIFO queue. The prefetch policy is to keep the inventory of useful packets at a constant size $\lceil T_A \times W_{Imax}/W_A \rceil$, where the inventory of useful packets is defined as the sum of the number of packets in the prefetch buffer and the number of outstanding cache requests. Intuitively, this is the number of packets required to completely mask the cache latency T_A even when the maximum sized (W_{Imax}) instructions are being issued. This policy requires one to initiate a cache fetch whenever the actual inventory falls below this size limit.

The inventory size is also an upper bound on the size of

the prefetch buffer, *i.e.*,

$$N_{FIFO} \leq \lceil T_A \times W_{Imax}/W_A \rceil$$

A tighter upper bound is possible if more history is kept around dynamically regarding the exact number and the timing of the outstanding cache fetches as discussed in [1].

Instruction alignment network. In order to accommodate variable length instructions, an instruction alignment network aligns the left boundary of the next instruction to the first bit position of the instruction register (IR) at each cycle. This network consists of a series of multiplexors controlled by the states of the IR, the On-Deck register (ODR), and the head of the FIFO queue. Not all shift increments are necessary since the instructions sizes are guaranteed to be multiples of a quantum size (Q_I).

Instruction unit control tables. The alignment network, the prefetch buffer, and the instruction fetch from the cache are controlled by logic whose specification as a control table is generated automatically according to the prefetch policy described above. This logic is responsible for the following tasks at each cycle:

- keeping track of the width of the instruction as well as the unused bits in the instruction register and at the head of the prefetch buffer,
- issuing instruction cache fetches, prefetch buffer writes and instruction register fills at the appropriate times, and
- generating the appropriate shift signal for the alignment network to align the next instruction.

Instruction decode tables. At each cycle, the left aligned instruction in the instruction register is decoded to yield the appropriate control signals for the various datapath control ports. A control table specification for this decode logic is generated automatically by walking the IF-tree. This may then be implemented either as random logic or as a PLA using standard logic synthesis tools.

8. Experimental Results

PICO-VLIW has been operational as a research prototype since late 1997. It allows us to explore hundreds or thousands of architectural alternatives in designing ASIPs, something that is very hard or impossible to do without an automated system. At this point, we have exercised it with several applications ranging from loop-intensive algorithms for signal and image processing to less structured ones such as compress and ghostscript. As an example, we present some of the results from the design space exploration for an application whose time-consuming part consists of a finite impulse response (FIR) filter. The following table lists the parameter ranges that define the design space to be explored; the number in parentheses are the step sizes. This design space contains 17640 different machines.

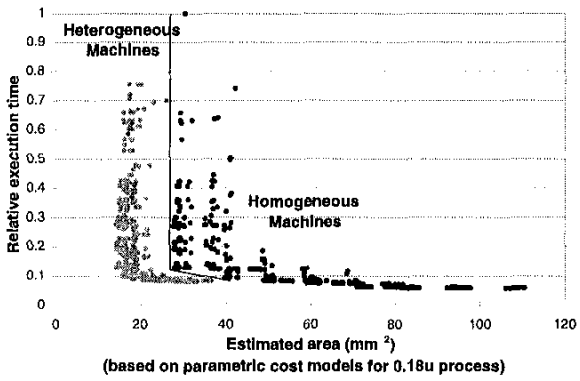

Figure 6. Sample PICO-VLIW machines.

Parameter	Range	Parameter	Range
predication	yes, no	speculation	yes, no
integer FUs	1-3	float FUs	1-5
memory FUs	1-2	branch FUs	1
integer regs.	16-64(8)	float regs.	16-64(4)
predicate regs.	256	branch regs.	8-16(4)

It is very hard to characterize the "quality" of an architecture objectively other than its cost and performance for a given application. In that spirit, Figure 6 shows the cost/performance characteristics of a number of machines in the design space that the Spacewalker actually considered to find 68 pareto-optimal designs. The machines displayed in Figure 6 fall in two categories. Machines represented by black circles are *homogeneous* machines which have general-purpose functional units and in which all functional units of a specific type (e.g., integer) are identical. Machines represented by grey circles are *heterogeneous* machines in which functional units have been specialized to the needs of the application and in which all units of a specific type are not necessarily identical. The results confirm the intuition that for a given performance, good heterogeneous designs are cheaper than good homogeneous designs, in this case by up to 50%.

9. Related work

The related work focuses on either the datapath design using a Spacewalker or the processor design from a concrete instruction set architecture (ISA). The MOVE project at Delft University falls in the first category. The emphasis is on the design of processor datapaths for *Transport Triggered Architectures* [5]. The datapath template used by the Spacewalker consists of a set of functional units, a set of register files and a set of buses connecting the functional units and the register files. The Spacewalker works with a structural representation of the datapath, adding and deleting register files, functional units, buses and interconnection

points to come up with a set of pareto-optimal datapaths. The philosophy for designing the control is simple, similar to horizontal microprogramming, *i.e.*, each control point is controlled by a separate field in the instruction word. Thus, the work doesn't address the design of sophisticated instruction formats optimized for code size and the corresponding instruction fetch and decode logic within the processor.

The work by Fisher *et al.* at HP Labs [6] is similar in nature and focuses on the design of processor datapath for a clustered VLIW architecture, similar to the Multiflow Trace architecture [4]. The datapath template used in the design process is highly stylized; for example, it doesn't permit register port sharing and assumes that each functional unit has dedicated ports to register files. A major component of their work is directed towards understanding how a processor designed for an application or a group of applications performs on other applications in the same domain, *e.g.*, image processing.

The approach presented by Hadjiyiannis *et al.* [8] uses Instruction Set Description Language (ISDL) [7] to specify a concrete ISA, which includes not only the desired operations but also the detailed instruction format and the constraints on instruction issue. The specification is then used to design the processor hardware in the form of synthesizable Verilog and to retarget various tools, such as a code-generator, assembler and simulator, needed to evaluate the performance. ISDL is a very general language capable of specifying many different types of architectures. Since an ISDL specification is at the level of a concrete ISA, the designer (either a person or a Spacewalker) has to do most of the work (*e.g.*, instruction format design) that our system does automatically. In our opinion, this makes it less suitable as a tool for comprehensive design space exploration and more suitable for a design process that requires only small incremental changes to an existing specification.

10. Conclusions

PICO-VLIW is a synthesis system for automatically designing the architecture and micro-architecture of VLIW and EPIC processors. It designs sophisticated processors with non-trivial requirements and constraints upon their ILP, shared register ports, variable-length multi-template instruction formats that minimize code size, an instruction prefetch unit that covers the instruction cache latency, and instruction alignment and distribution networks to deal with the variable length instructions. A novel aspect of our approach is the distiction we make between the logical and the physical instruction formats.

PICO-VLIW was designed with automatic design space exploration in mind; the VLIW synthesis in PICO-VLIW is driven by an abstract rather than a concrete ISA specification, since it is easier for the Spacewalker (or, for that matter, a human being) to specify the former. Starting from this specification, PICO-VLIW automatically generates,

1. the concrete ISA for the processor,
2. the detailed micro-architecture including the datapath and the controlpath output in the form of RTL-level structural VHDL,
3. a machine description for use by our retargetable compiler, assembler and simulator, and,
4. an architecture manual and detailed statistics for the Spacewalker.

Acknowledgements

The authors would like to thank Mike Schlansker for his contributions to the archspec definition, and Richard Johnson for his help in custom instruction template design.

References

[1] S. Aditya and B. R. Rau. Automatic architectural synthesis and compiler retargeting for VLIW and EPIC processors. Technical Report HPL-1999-93, Hewlett-Packard Laboratories, 1999.

[2] S. Aditya, B. R. Rau, and R. C. Johnson. Automatic design of VLIW and EPIC instruction formats. Technical Report HPL-1999-94, Hewlett-Packard Laboratories, 1999.

[3] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 98–105, Boston, Massachusetts, June 23–25, 1982.

[4] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. P. Papworth, and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. In *Second Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 180–192, Palo Alto, CA, October 1987.

[5] H. Corporaal and R. Lamberts. TTA Processor Synthesis. In *First Annual Conf. of ASCI*, Heijen, The Netherlands, May 1995.

[6] J. A. Fisher, P. Faraboschi, and G. Desoli. Custom-Fit Processors: Letting Applications Define Architectures. In *29th Annual IEEE/ACM Symposium on Microarchitecture (MICRO-29)*, pages 324–335, Paris, December 1996.

[7] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An instruction set description language for retargetability. In *ACM/IEEE Design Automation Conference*, 1997.

[8] G. Hadjiyiannis, P. Russo, and S. Devadas. A Methodology for Accurate Performance Evaluation in Architecture Exploration. In *Design Automation Conference*, New Orleans, LA, June 1999.

[9] V. Kathail, M. Schlansker, and B. R. Rau. HPL PlayDoh architecture specification: Version 1.0. Technical Report HPL-93-80, Hewlett-Packard Laboratories, Feb. 1994.

[10] B. R. Rau, V. Kathail, and S. Aditya. Machine-description driven compilers for EPIC and VLIW processors. *Design Automation for Embedded Systems*, 4:71–118, 1999.