Code Size Minimization and Retargetable Assembly for Custom EPIC and VLIW Instruction Formats

Shail Aditya, Scott A. Mahlke, B. Ramakrishna Rau
Compiler and Architecture Research
HPL-2000-141
October 2000

Email: {aditya,mahlke,rau}@hpl.hp.com

EPIC, VLIW, instruction format design, code size minimization, noop compression, design automation, custom templates, retargetable assembly

PICO is a fully automated system for designing the architecture and the microarchitecture of VLIW and EPIC processors. A serious concern with this class of processors, due to their very long instructions, is their code size. One focus of this paper is to describe a series of code size minimization techniques used within PICO, some of which are applied during the automatic design of the instruction format, while others are applied during program assembly. The design of a retargetable assembler to support these techniques also poses certain novel challenges which constitute the second focus of this paper. Contrary to widely held perceptions, we demonstrate that it is entirely possible to design VLIW and EPIC processors that are capable of issuing large numbers of operations per cycle, but whose code size is only moderately larger than that for a sequential CISC processor.

A version of this article appears in ACM Transactions on Design Automation of Electronic Systems, vol 5, no. 4, October 2000.

1 Introduction

VLIW (very long instruction word) processors are beginning to establish 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 sequential, RISC- or CISC-like ASIPs–primarily a matter of designing the opcode repertoire–there has been relatively little work in the area of automatically synthesizing the architecture and microarchitecture of VLIW processors or, for that matter, of processors of any kind that provide significant levels of instruction-level parallelism (ILP).

The goal of our PICO (Program-In-Chip-Out) project is to fully automate this process, so that optimized ASIP and ASIC designs can be generated automatically from a given application program written in some high-level language such as C. One aspect of PICO is a fully automated system for designing the architecture and microarchitecture of application-specific VLIW processors [2] and their generalization, EPIC (explicitly parallel instruction computing [16]) processors.[1] We refer to this process as architecture synthesis to distinguish it from behavioral or logic synthesis which are at a lower level, and we refer to this design subsystem as PICOVLIW. Given an abstract description of the target architecture, the PICOVLIW system automatically produces an RT-level, structural description of the target processor in VHDL along with a nonstructural **machine-description database** (**mdes**) which includes an automatically generated instruction format for the target processor. The mdes is used to retarget various software tools to the target architecture automatically.

The general perception of VLIW processors is that they cause great increases in code size. This can be the case with simplistically designed VLIW architectures. Consequently, a major emphasis of PICOVLIW has been on containing the resulting code size for the VLIW architectures that it designs. Since code size reductions are often at the expense of hardware complexity or performance penalties, this is a nontrivial task and constitutes one focus of this paper. There are two classes of techniques used for this purpose: those that are applied at the time that PICOVLIW architects

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

the processor, and those that are performed by the mdes-driven assembler during assembly. The design of an mdes-driven assembler in this context poses certain novel challenges, which is the second focus of this paper.

We discuss the measures taken by PICOVLIW to minimize code size in Section 2. In Section 3, we discuss the novel aspects of our mdes-driven assembler, and in Section 4, we provide experimental evidence of the effectiveness of our techniques for trading off code size with hardware cost and run-time performance of the processor. Section 5 discusses related work and Section 6 presents conclusions.

2 Minimization of Code Size

Explicit instruction-level parallelism is a defining property of VLIW; each VLIW **instruction** is able to specify a set of operations that are to be issued simultaneously. This property is termed **MultiOp** [14]. A MultiOp instruction contains multiple **operation slots**, each of which specifies one of the operations that are to be issued simultaneously. An **operation** is the smallest unit of execution in a VLIW processor and is the equivalent of a conventional RISC or CISC instruction in that it typically specifies an opcode, one or two source operands, and a destination operand.

We define the **canonical instruction format** to be the MultiOp instruction format that has an operation slot per functional unit. The operation slots need not be of uniform width, and each one can use exactly as many bits as it needs. This conserves code space. Furthermore, the correspondence between an operation slot and a functional unit can be implicitly encoded by the position of the operation slot within the instruction–a further saving in code space over having to specify this mapping explicitly, somewhere in the instruction.

However, when the parallelism in either the hardware or the program is unable to sustain the level of parallelism permitted by the canonical format, some of these operation slots will contain noops, leading to wasted code space. Worse yet, the schedule created by the compiler might be such that there is no operation whatsoever scheduled to issue on certain cycles. In a VLIW processor, with no hardware interlocks, this situation requires the insertion of one or more MultiOp instructions containing nothing but noops. These completely empty instructions can be eliminated by the inclusion of a **multinoop** field in the MultiOp instruction format that specifies the number of noop instructions that are to be issued, implicitly, after the current instruction [4].

A more challenging problem is to get rid of code wastage caused by explicit noop operations in an instruction that is not completely empty. A variety of noop compression schemes can be devised to address this problem [16]. The one that we employ involves the use of multiple instruction formats or **templates**, each of which provide operation slots for just a subset of the functional units. The rest of the functional units receive a noop implicitly, avoiding wastage of code space. In the remainder of this section, we present the class of instruction formats designed by PICOVLIW and the various measures it takes to address issues relating to code size that are unique to VLIW processors.

2.1 Our generic instruction format

Figure 1 displays the generic logical structure of the instruction formats designed by PICOVLIW. It consists of multiple templates with a template select field that identifies the specific template. Each template contains one or more operation slots, and may contain a multinoop field. Identifying a template completely determines the number of operation slots, their syntax, and their bit positions.

For convenience, the various operations for a given machine are grouped into **operation groups** (**opgroups**), each of which is a set of operations that are similar in nature in terms of their latency and their connectivity to physical register files, and that are constrained to be mutually exclusive with respect to operation issue (e.g., add and subtract operations on the same ALU). Each operation slot can specify one operation from a **super group**, which is a set of pairwise mutually exclusive opgroups, all of which have identical mutual exclusion relationships with every opgroup that is not in that super group. Using this approach, the canonical format is represented by a template in which each super group consists of precisely the opgroups supported by a functional unit in the machine.

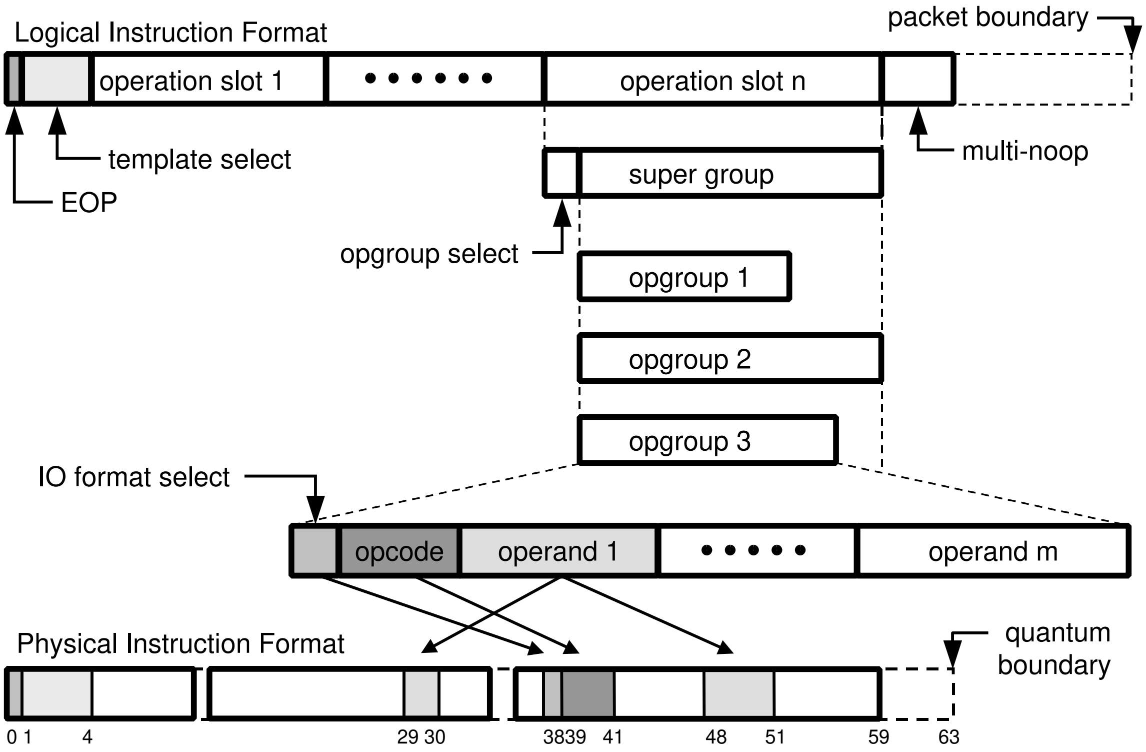


Figure 1: Our generic instruction format syntax showing hierarchical multitemplate structure (logical instruction format) and physical bit allocation (physical instruction format).

Within each operation slot, the opgroup select field selects an opgroup from the super group. The syntax of an opgroup is similar to a RISC instruction: an opcode field and a sequence of source and destination operand specifiers, collectively known as the **IO format**. The IO format select field chooses among multiple IO formats allowed for an opgroup. An operand specifier may, in general, be one of a set of instruction fields, called an **IO set**, that identify the exact kind and location of the operand (e.g., a register specifier or a literal). IO sets with multiple choices have a select field to identify which instruction field is intended. Finally, the lowest level of the syntactic hierarchy consists of various kinds of **instruction fields**, including the multinoop field, select fields, opcode fields, and register and literal operand specifiers.

2.2 Custom templates

In addition to the canonical template, PICOVLIW automatically defines a set of **custom templates** that are customized to a given application (or a set of applications) and are narrower in width. By customizing templates to an application, we are able to accommodate the widest instructions where necessary, while employing compact, restricted instructions for much of the code without constraining the parallelism or the resource requirements of the application beforehand.

PICOVLIW currently uses a very simple heuristic to select a set of custom templates, since our initial emphasis has been merely to understand the effectiveness of this strategy. (We expect ample room to improve the heuristics, subsequently.) Given a budget K in terms of the number of custom templates to be added, our current heuristic is to pick the K most frequent combinations of opgroups in the scheduled program. To do this, PICOVLIW generates a histogram of combinations of opgroups for the scheduled program. This is done by mapping the scheduled operations of an instruction back to their opgroups. A static histogram records the frequency of static occurrence of each combination within the program, and is useful in optimizing the static code size. A dynamic histogram, which weights each operation group combination with its dynamic execution frequency, can be used to improve the instruction cache performance by giving preference to the most frequently executed sections of the code. Currently, we use the static histogram in determining the custom templates in order to give preference to minimizing the overall static code size. A more detailed description of this process appears in the technical report [1].

2.3 Trading code size for controlpath complexity

The trade-off involved in the use of a multitemplate instruction format is between code compaction and the complexity of the controlpath (i.e., the instruction fetch and decode pipeline, starting at the instruction cache through the control ports of functional units, register files, and other portions of the datapath). PICOVLIW uses a stylized controlpath consisting of an instruction prefetch buffer, alignment network, instruction register (IR), distribution network, and instruction decode logic. A customized controlpath is designed automatically by PICOVLIW to specifically fit the needs of the target instruction format. Consequently, the structure and cost of the controlpath is a function of the instruction format. With only the canonical format, the instruction alignment and distribution networks are trivial if the width of the **instruction packet** (the unit of access from the instruction cache) is chosen to be the same as the width of the canonical format. Conversely with the multi-template format, these networks may become quite complex in order to handle the alignment of variable-width instructions and the distribution and decoding of control fields occurring at different positions within the IR. A more detailed description of the controlpath structure and the synthesis strategy is available in our paper [2].

An aspect of the instruction formats designed by PICOVLIW is that the instruction fields within each template can be positioned in some permuted, but fixed, way that is specified by the template select field. The fields corresponding to an operation slot need not be positioned contiguously. Furthermore, an individual field is, itself, permitted to consist of a discontiguous set of bit positions. This is possible due to the firm distinction PICOVLIW places between the **logical instruction format** consisting of the choice of templates and their syntactic structure, and the **physical instruction format** consisting of the physical bit positions of the various instruction fields within a template (refer to Figure 1). This separation reflects the fact that PICOVLIW designs the instruction format with hardware optimality in mind, and not the convenience of a human machine-code programmer.

This additional degree of freedom in the physical positioning of instruction bits can be exploited to reduce the cost and complexity of the decode and distribution network that lies between the IR and the datapath's control ports. The permutation applied to the fields of each template is selected with a view toward minimizing, for each control port, the number of distinct bit positions, across all of the templates, at which the instruction fields controlling that port are to be found. We refer to this as **affinity allocation**. If one were completely successful in doing this, the distribution network would be as simple as for the canonical format. However, if performed indiscriminately, this can also lead to template widths that are similar to that for the canonical format. PICOVLIW's design heuristics attempt to strike a balance between these competing goals, reducing the amount of multiplexing in the distribution network without causing too much increase in template widths.

The complexity of the alignment network can be partially contained by requiring that the width of all instruction templates be a multiple of some number of bits, which we refer to as the **quantum**. All shift amounts are thereby guaranteed to be a multiple of the quantum size, reducing the degree of multiplexing in the alignment network. The adverse effect of quantization is that the template width must be rounded up to an integral number of quanta. PICOVLIW, however, takes advantage of such unused bits in a template, whether due to quantization or other reasons, by opportunistically collecting them to form the multinoop field.

2.4 Trading code size for branch penalty

The instruction prefetch mechanism ensures that the processor will never stall, for want of an instruction to decode, while it is executing sequentially. Furthermore, when a branch is taken, the fact that the program schedule has been created with the branch latency in mind guarantees that there will be no stall cycle as long as the branch target fits entirely within the first instruction packet that is fetched from the target location. If not, one stall cycle is experienced. This stall cycle can be avoided by aligning a branch target, which would have straddled a packet boundary, to start on the next packet boundary. Doing so causes the last part of the instruction packet, which precedes the aligned branch target, to be empty. This empty region must be treated as an extension of the instruction preceding the empty region when the program executes sequentially through that instruction. For this purpose, the first bit in every template is what is called a **consume-to-end-of-packet** (**EOP**) bit. When this bit is set, the rest of the packet, from the end of the current instruction to the end of the current packet, is viewed by the controlpath logic as empty, and is skipped. The EOP bit makes it possible to avoid branch stall penalties by aligning branch targets. However, the resulting empty regions increase the code size.

In this section we have presented a number of measures for reducing code size while containing hardware complexity and branch stall cycles. Certain of these measures are applied by PICOVLIW at the time that the instruction format is designed. These include the use of a multitemplate instruction format, the identification of the minimal templates, the selection of the custom templates, affinity allocation, quantization, and the provision of a multinoop field and an EOP bit. The remaining measures, which are taken by the assembler, are the selection of the best available template for each instruction and the skillful use of the multinoop field and the EOP bit.

3 Machine-Description Driven Assembly and Linking

An important requirement for automatically applying the techniques discussed above is the capability to retarget the entire software tool chain, including the compiler, assembler, disassembler, simulator, etc., to an arbitrary machine architecture within the design space. To enable this, all the tools are designed to be mdes driven and to access architecture-dependent information indirectly through a **mdes query system** (**mQS**) interface.

We have shown previously [15] how to design mdes driven compilers and the mQS interface for VLIW architectures primarily supporting the instruction scheduler and the register allocator modules. In this paper, we briefly describe the additional mechanisms needed during assembly[2] to support the techniques described above. The reader is referred to our technical report [1] for a more detailed account of our mdes driven assembler and its mQS interface.

3.1 Issues in assembler design

A VLIW assembler is provided with the scheduled program consisting of a sequence of sets of architectural operations. Each set consists of the operations scheduled on the same cycle. The job of the assembler is to encode each set of operations into a single instruction and consequently the entire program into a single stream of instruction and data bits. Although the details of the instruction format are hidden behind the mQS interface and are managed by it, many important architecture-independent decisions, including code layout and instruction template selection policies, are still the responsibility of the assembler. We briefly discuss these below.

[2] In all our discussion, we use the term *assembly* to denote the combined processes of instruction assembly and program linking.

3.1.1 Code layout

Our approach of designing variable-width instruction templates and variable-width operation slots within them implies that the width of each instruction and consequently its address offset within a procedure cannot be determined simply by counting the number of operations. A consequence of this is that the resolution of symbolic addresses within the program including forward branch offsets and data and procedure references becomes a two-pass process. In the first pass, each instruction in a procedure is assigned an address offset by selecting an instruction template for it and possibly aligning it to some address boundary. In the second pass, each instruction is assembled into the template selected for it while resolving symbolic addresses to actual addresses assigned earlier.

3.1.2 Template selection

Given a set of coscheduled operations, more than one instruction template (canonical or custom) may be able to encode it. The assembler selects an appropriate template based on a preferred selection policy. Our current policy, which is geared toward reducing the overall code size, is to choose the shortest template that may encode a given set of operations, and in case of a tie, the one with the maximum number of allowable multinoop cycles. If the chosen template cannot encode all the multinoop cycles that are needed, the next instruction is chosen to be the shortest template, which will consist entirely of noops, that can encode all the remaining noop cycles.

3.1.3 Branch target alignment

The assembler is also responsible for making branch target alignment decisions while laying out the code in memory. We use a profile driven heuristic to ensure that the most frequently executed targets do not cross a packet boundary. With this approach, the compiler needs to profile the program and annotate the assembly code with the frequency with which each branch target is visited via a taken branch. Our heuristic operates by first sorting branch targets from the highest to the lowest dynamic frequency. Then, targets are classified one by one as not permitted to cross a packet boundary (noncrossing) by keeping track of two cumulative values: the dynamic fraction of targets already classified as noncrossing (initially 0.0) and the static fraction of targets that may potentially cross a packet boundary (initially 1.0). At the point when the dynamic fraction of noncrossing targets is equal to or larger than the static fraction of potentially crossing targets, the process stops and all the remaining unprocessed targets are classified as potentially crossing a packet boundary. Finally, if a branch target instruction that has been classified as noncrossing is found to cross a packet boundary during the address assignment phase, it is aligned to the next packet boundary and the EOP bit is set in the preceding instruction.

This heuristic essentially tries to achieve an even balance between the dynamic fraction of non-crossing targets (causing increase in code size) and the static fraction of potentially crossing targets (causing branch penalty) to effectively trade off branch penalty and code size. The results presented in Section 4.6 show its effectiveness.

3.2 mQS data structures and mechanisms

Given an appropriate template, the assembler uses the mQS interface to assemble the given set of coscheduled operations into a string of bits. Likewise, the disassembler uses the mQS interface to parse a given string of bits into a set of operations. In either case, format-specific information, such as the hierarchical structure of the template and various field encodings, is kept and manipulated within the mQS by a **template manager**. Below, we describe the various internal data structures and mechanisms used by the template manager to support the process of assembly and disassembly.

3.2.1 Template selection support

The template manager needs to find all instruction templates that can encode a given set of architectural operations characterized by their opcode and IO format combinations. The number of such combinations is, however, extremely large. Therefore, rather than enumerating all possible sets of

```
Procedure FindValidTemplates (List<opcode, ioformat> operationList)
    Set<Template> templates = AllTemplates ;
    Foreach (opcode, ioformat) in operationList do
        Set<OpGroup> opgroups = OpcodeMap.value(opcode) ;
        Set<Template> opTemplates = emptySet ;
        Foreach opgroup in opgroups do
            If (Match(opgroup.ioformat, ioformat))
            then opTemplates = opTemplates ∪ opgroup.templates ; endif
        endfor
        templates = templates ∩ opTemplates ;
    endfor
    return templates ;
```

Figure 2: Pseudo code to find the set of valid templates that may encode a given set of operations.

operations and the templates that may encode them, the template manager uses some auxiliary data structures and a matching algorithm to find the valid templates.

The template manager keeps a hash table called the **opcode map** which maps each architectural opcode to the set of opgroups (kept as a bitvector) in which that opcode may appear. Different opgroups in this set may implement either disjoint or overlapping IO formats for the given opcode. An opcode may end up belonging to multiple opgroups when a new combination of opcodes or certain IO format combinations of existing opcodes are separated into a new opgroup for template customization.

Another data structure kept by the template manager is the **opgroup table**, which lists all opgroups and the various IO formats they support. The size of this table is proportional to the number of opgroups supported by the architecture, which is usually much smaller than the number of architectural opcodes. Each opgroup in the opgroup table also keeps a set of templates (also kept as a bitvector) in which it can be encoded. This set is directly derived from the instruction format design process by identifying the opgroups that are present in a template.

Figure 2 shows the pseudo code used by the template manager to identify all valid templates that can encode a given instruction. The input, provided by the assembler, is a list of opcode IO format pairs characterizing the current instruction. For each such pair, the first step is to map the given opcode to the set of opgroups it belongs to using the opcode map. The next step is to filter this set for matching opgroups that can implement the desired IO format for that opcode. If more than one opgroup matches, then each is considered to be a candidate for encoding that opcode IO format pair. A union of the set of templates that may encode a candidate opgroup gives the set of all templates that may encode the given operation. Finally, the intersection of such sets across all operations gives the set of valid templates that may encode all the operations of the current instruction.

Any one of the templates found in the above process may be selected by the assembler to encode the given instruction. The template manager keeps various properties of each template in a **template table** to help in this process. Such properties include the size of the template in bits, the number of multinoop cycles that can be encoded within it, and the number of unused bits within the template due to encoded noops as well as affinity allocation of fields.

3.2.2 Template assembly and disassembly

The assembly and disassembly processes require complete knowledge about the bit positions of all instruction fields and their encodings within each template that is selected. A simple way to organize this information across all templates is to capture the hierarchical syntax of the instruction format (Figure 1) in a tree data structure called the **IF-tree** as shown in Figure 3. The leaves of this tree are the instruction fields (i.e., opcodes, operands, literals, and various selector fields) and the internal nodes represent the hierarchical syntactic elements (i.e., instruction, templates, operation slots, opgroups, IO formats, and IO sets). Each leaf also identifies a set of bit positions allocated to it. The leaves corresponding to the various selector fields (e.g., template selector field, opgroup selector field, opcode, etc.) also keep the exact encoding of the choices available at that subtree. All this information is supplied to the mQS by the instruction format design process.

The template manager uses the IF-tree not only to identify the instruction syntax and bit positions of various fields, but also to keep track of the partial state of assembling an instruction. This can be easily accomplished by keeping a pointer at each level of the IF-tree to identify the subtree under

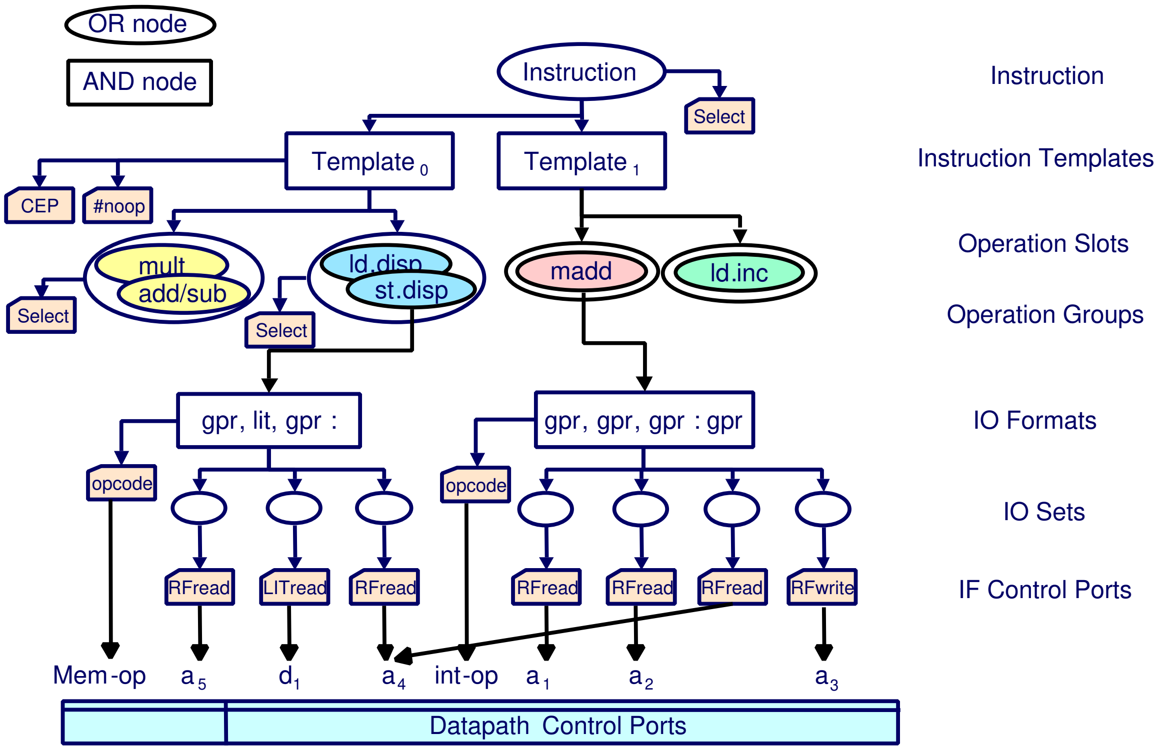


Figure 3: The structure of the IF-tree.

which an instruction field is being assembled. This permits a hierarchical, iterative interface to be built to direct the template manager through the various steps of assembly.

The process of assembly proceeds as follows. First, the template manager allocates a bitvector wide enough to encode the template chosen by the assembler and sets the template selector field bits to indicate that choice. Next, the assembler directs the template manager to encode each operation belonging to the current instruction one by one. For each operation, the assembler presents its opcode to the template manager followed by its operands in the order specified by the IO format of the operation. The template manager first converts the given opcode to its opgroup using the opcode map and identifies its subtree node in the IF-tree. This enables it to identify the operation slot occupied by this operation and set the opgroup, opcode, and IO format selector field bits. The various operands are assembled by consulting the corresponding leaves below the opgroup subtree for their bit positions and encodings. Finally, the EOP bit and the multinoop cycles are filled in as needed and the finished string of bits representing the complete instruction is returned to the assembler.

The process of disassembly proceeds in the reverse manner. First, the disassembler supplies a string of instruction bits to the template manager, whose length is equal to the maximum size instruction. The instruction to be disassembled is positioned left-justified in this string. The template manager decodes the template select field and the EOP bit to identify the template and its size. The disassembler uses this information to determine the start of the next instruction. Next, the template manager returns a list of the opcodes and IO formats of the various operations encoded within the current instruction. This information is determined by identifying the bit positions of the various selector fields within the current template and decoding their value using the IF-tree. The disassembler then uses this list to disassemble each operation one by one by supplying the opcode to the template manager and then querying its operands in the order specified by the IO format. Each such query returns the actual register number or literal value for the corresponding operand. The template manager disregards any extra bits in the supplied string that are not decoded as part of the current instruction.

3.3 The mQS interface

The set of mQS interface functions related to the process of assembly and disassembly can be classified into the following categories:

1. **Template selection** These functions enable the selection of the most appropriate instruction format template for a given set of architectural operations scheduled in the same cycle.

2. **Template assembly** These functions help to assemble a single VLIW instruction by filling the operation information provided by the assembler into the template and then returning the bit encoding of the fully assembled instruction.

3. **Template disassembly** These functions help to disassemble an instruction byte stream, identifying the set of operations scheduled within one instruction.

Template selection functions	
set_operation_tuple	Provide a set of opcode IO format pairs for template selection.
get_next_template	Obtain the next valid template for the current instruction
set_template	Select the given template for encoding the current instruction
get_size	Obtain the size of the given template in bits.
get_unused_bits	Obtain the number of unused bits in the current template (noops etc.).
get_max_multi_noops	Obtain the number of multinoop cycles allowed in the given template.
Template assembly functions	
assemble_op	Initiate the assembly of an operation with the given opcode.
assemble_pred	Assemble a predicate register operand.
assemble_pred_lit	Assemble a predicate literal (true, false) operand.
assemble_src	Assemble a register operand.
assemble_src_lit	Assemble an immediate literal operand.
assemble_dest	Assemble a destination register operand.
assemble_multi_noop	Set the number of multinoop cycles to the given value.
assemble_EOP	Set the EOP bit to the given value.
get_instruction	Obtain the fully assembled vector of bits for the current instruction
Template disassembly functions	
set_instruction	Provide a string of bits for disassembly.
get_operation_tuple	Obtain the set of opcode IO format pairs present in the current instruction
disassemble_op	Initiate disassembly of an operation with the given opcode.
disassemble_pred	Disassemble a predicate register operand.
disassemble_pred_lit	Disassemble a predicate literal operand.
disassemble_src	Disassemble a register operand.
disassemble_src_lit	Disassemble an immediate literal operand.
disassemble_dest	Disassemble a destination register operand.
disassemble_multi_noop	Obtain the number of multinoop cycles set in the current instruction
disassemble_EOP	Obtain the value of the EOP bit.
Miscellaneous functions	
get_packet_size	Obtain the instruction packet size in bits.
get_quantum_size	Obtain the quantum size in bits.
get_max_inst_size	Obtain the size of the maximum sized instruction in bits.

Table 1: The mQS interface functions.

4. **Miscellaneous instruction format information** These functions provide general information regarding the instruction format of the machine.

The functions provided in each of these categories are summarized in Table 1. A more detailed description of each function including its input and output signature is provided in the technical report [1].

4 Experimental Results

Our instruction format synthesis system provides four central features to reduce code size while limiting control hardware complexity and run-time branch stall cycles. These features consist of a multinoop field, custom templates, affinity allocation, and a consume-to-end-of-packet (EOP) bit. In this section, the effectiveness of these features is evaluated for a range of VLIW processors.

4.1 The experimental setup

The central focus of PICOVLIW is designing custom processors for applications in the embedded domain. A subset of the MediaBench 1.0 applications are selected consistent with this focus [11]. The applications are epic and unepic (image compression), pgpencode and pgpdecode (encryption), ghostscript (postscript interpretation), mipmap (3D graphics), and rasta (speech recognition).

The experiments utilize a class of VLIW processors defined by four parameters (IFMB): I denoting the number of integer units, F denoting the number of floating-point units, M denoting the number of memory ports, and B denoting the number of branch units. We consider five processors in this class defined by their IFMB tuple, 1111, 2111, 3121, 4121, 6132, which can issue up to 4, 5, 7, 8, and 12 operations per cycle, respectively. We chose to fix the number of floating-point units at one because our benchmark set contains very little floating-point computation and does not benefit from additional floating-point resources. All processors utilize the HPL-PD instruction set [9]. A constant register file configuration is assumed across all the processors, consisting of 64 general-purpose registers, 16 floating-point registers, 32 predicate registers, and 32 branch-target registers.

Experimental parameter	Range of variation
Processor issue width	1111, 2111, 3121, 4121, 6132
Operation latencies	1x, 2x, 3x
Use of the multinoop capability	no, yes
Number of custom templates	0, 3, 7, 15, 31, 63, max
Affinity allocation	none, full
Use of the EOP bit	always, never, heuristic

Table 2: Experimental parameters and their range of variation.

In addition, we assume immediate short literals are 6 bits and long literals are 32 bits. The quantum size is chosen to be 16 bits for all VLIW processors.

One additional processor, customized to each application, is defined to serve as the reference machine with respect to code size. Since we want the reference machine to have the minimum possible code size, our reference machine is a hypothetical CISC processor, referred to as the pseudo-CISC processor. It issues exactly one operation of any type per instruction and, since it is assumed to have hardware interlocks, requires no noop instructions. It utilizes variable-length encoding for instructions with as many custom templates as needed by the application, employs no affinity allocation, and has a quantum size of 8 bits to achieve as small a code size as possible. The pseudo-CISC's code size approximates the best code size that we can achieve using all the features of our instruction-format for a sequential, single-issue processor. Hence, it also estimates the best code size achievable given the HPL-PD architecture. Although an actual RISC or CISC processor might seem to be a better reference than this hypothetical one, the architectural differences between the operation repertoire of an actual processor and HPL-PD are so great that they would invalidate any attempt to measure the effectiveness of the techniques described in this paper. In the interests of performing a controlled and meaningful set of experiments, we have factored out this architectural variable by defining a pseudo-CISC processor that uses the HPL-PD operation repertoire.

The parameters listed in Table 2 are varied for different experiments. Operation latencies for the 1x case are as follows: simple integer, 1 cycle; simple floating point, 3 cycles; memory load, 2 cycles; memory store, 1 cycle; integer and floating point multiply, 3 cycles; integer and floating point divide, 8 cycles; and branch, 1 cycle. When the operation latencies are varied, the 1x latencies are uniformly scaled by two or three to examine the effects of deep pipelines on the instruction format. When the number of custom templates is set to 0, we are left with only the canonical template. PICO has the ability to apply affinity allocation either to all or to just a subset of instruction fields. Here we only study the effect of applying affinity allocation to all instruction fields (i.e., full affinity).

Each experiment reports on one or more of the following data: code size, controlpath cost, or dynamic branch stall cycles. Code size is the total size in bytes of the text portion of each application excluding shared libraries after assembly and linking are complete. Controlpath cost is the estimated area for the controlpath for the given instruction format (Section 2.3). Dynamic branch stall cycles is an estimate of the number of stall cycles obtained by counting the unaligned branch targets (Section 2.4) weighted by the frequency of visiting that branch target via a taken branch, as derived from profiling the application on a sample input set.

4.2 Comparison of the canonical and pseudo-CISC formats

As a starting point for the evaluation, the canonical code size for all of our processor configurations is compared against that of the pseudo-CISC processor in Table 3. The canonical code size refers to the code size that results without the use of any of the four features of our instruction format. From the table, we see that code size uniformly increases with both width and latency. In general, all of the code size ratios are strikingly large, with the largest ratio of 28.8 for mipmap and pgpdecode on the 6132 processor and latency 3x. Even for the 1111 (4-issue), the code size increases are quite large, ranging from 3.6 to 4.8 for latency 1x and 4.5 to 8.9 for latency 3x. Such large increases in code size are unacceptable for embedded systems where instruction memory is at a premium. These results clearly illustrate the traditional criticism of VLIW processors wherein the instruction encoding becomes increasingly less efficient as more parallelism is introduced into the processor. Because the application is not uniformly parallel, more instruction space must be wasted to encode noop operations when the full processor parallelism cannot be utilized. For the rest of this evaluation, we address the effectiveness of the features of our instruction format at

Application	Latency	Processor Width				
		1111	**2111**	**3121**	**4121**	**6132**
epic	1x	4.17	3.95	4.98	6.24	8.76
	2x	5.19	5.98	7.97	10.08	14.60
	3x	6.53	8.14	11.25	13.98	20.44
ghostscript	1x	4.09	4.79	6.05	7.67	10.40
	2x	5.56	7.39	9.72	12.43	17.00
	3x	7.14	10.07	13.66	17.04	23.61
mipmap	1x	4.44	5.61	7.28	9.28	12.60
	2x	6.51	8.86	11.78	15.17	20.77
	3x	8.51	12.19	16.71	20.85	28.84
pgpdecode	1x	4.84	5.81	7.62	9.72	13.29
	2x	6.90	9.16	12.10	15.45	21.19
	3x	8.87	12.34	16.71	20.82	28.81
pgpencode	1x	4.83	5.76	7.54	9.61	13.13
	2x	6.86	9.06	11.95	15.24	20.89
	3x	8.80	12.19	16.48	20.53	28.36
rasta	1x	3.87	4.21	5.25	6.39	8.73
	2x	5.16	6.46	8.41	10.43	14.49
	3x	6.58	8.75	11.75	14.51	20.28
unepic	1x	3.64	3.28	3.68	4.30	5.66
	2x	4.05	4.26	5.45	6.64	9.16
	3x	4.54	5.54	7.38	8.97	12.65

Table 3: Code size with the canonical instruction format as a function of processor width and functional unit latency. Each cell of the table contains the ratio of the code size achieved for a width latency pair versus the code size for the pseudo-CISC processor for each application.

reducing the effect of width and latency on code size for VLIW processors.

4.3 Effectiveness of the multinoop field

Figure 4 presents the effect of processor latency on code size, with and without the multinoop field, for all three sets of latencies. It is clear from the figure that code size dramatically increases with higher latency in the absence of the multinoop field. This increase is primarily due to cycles in which no operations are scheduled. Without a multinoop field, each such noop instruction requires an explicit template to specify it. This problem is exacerbated because we have used no custom templates in this experiment, so the only mechanism for encoding a full cycle of noop operations

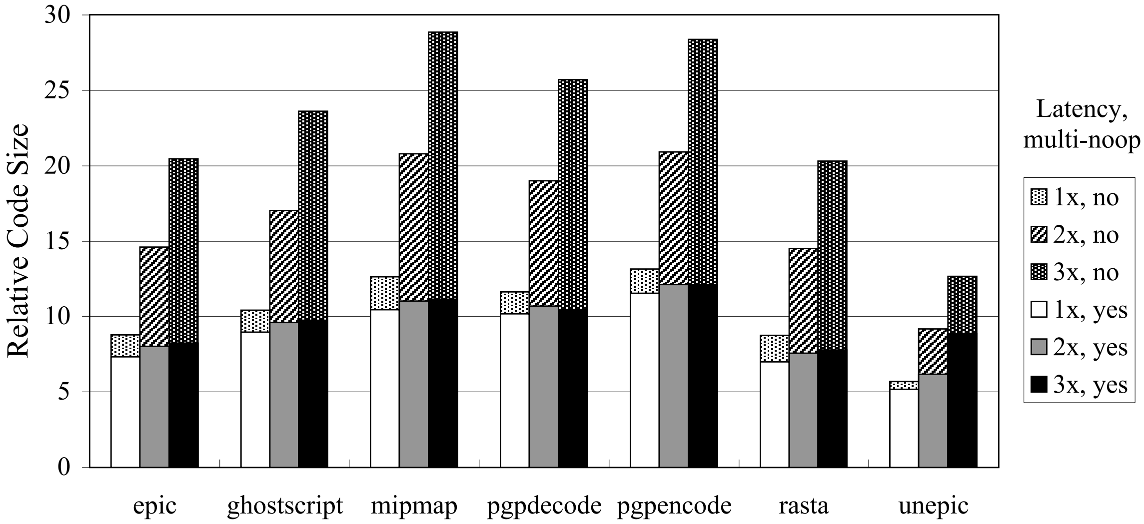


Figure 4: Effect of the multinoop capability upon the code size as a function of operation latency. The following parameters are fixed for the experiment: 6132 processor, no custom templates, no affinity, and no use of the EOP bit. Code sizes are normalized with respect to the code size obtained on the pseudo-CISC processor. The upper portion of each bar shows the relative code size without the multinoop feature, whereas the lower portion of each bar displays the relative code size with the multinoop feature.

is with the fullwidth template.

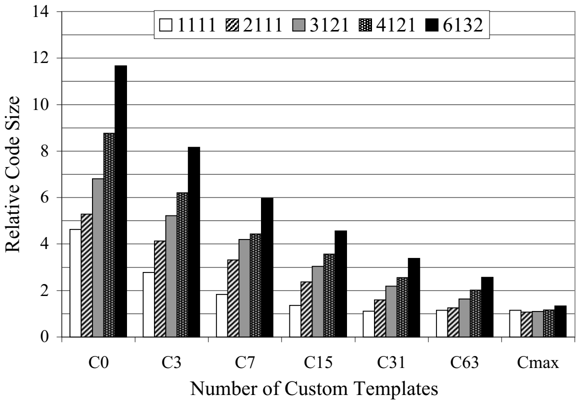
With the multinoop field, the code size is uniformly reduced across all applications and latencies (even 1x), cutting down on explicit noop instructions. Also, the sensitivity of code size to latency is almost eliminated. For example with epic, the code size increase with multinoop is reduced to 10% and 12% (as compared to 67% and 133% without multinoop), respectively, for 2x and 3x latencies. The multinoop field provides a virtually free mechanism to encode full cycles of noop operations using spare bits in the templates. Hence, all of these cycles of noop operations can be compressed out of the instruction encoding, thereby minimizing the impact of latency on code size. The rest of the experiments all assume the use of the multinoop capability.

4.4 Code size and hardware complexity with custom templates

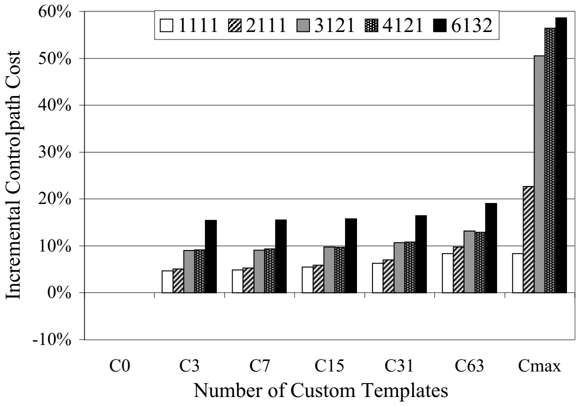
Figure 5(a) shows the effect of adding custom templates on code size for the application pgpdecode (which is representative of all applications considered). The chart shows a couple of trends. A first observation is that for a given number of custom templates, the code size grows as the machines become wider because the width of the canonical template increases. More interestingly, the code size decreases quite dramatically (by almost a factor of 9 for the 6132 machine configuration) as the number of custom templates is increased. This trend is visible across all machine configurations. This is the most obvious benefit of template customization – the fact that additional custom templates provide a narrower way of encoding additional combinations of coscheduled operations that would otherwise be encoded using the default canonical template (represented by C0).

Second, since we customize the most frequent operation combinations first, the incremental benefit of customization on the code size diminishes as more templates are added. However, what is more significant is that the variation in code size across machine configurations is also reduced quite significantly. Indeed, with the maximal number of custom templates (Cmax), the variation in code size across machines is negligible because as the fraction of code covered by custom templates increases, the remaining fraction that uses the canonical template decreases, thereby reducing the sensitivity of code size to machine width. This implies that adding custom templates is an effective way to eliminate the increase in code size as the machines are scaled in width. A final observation is that with Cmax custom templates (and using the multinoop field), the code size for all processors has been reduced to approximately that of the pseudo-CISC processor; all of the code size penalty traditionally associated with a wide-issue VLIW has been removed.

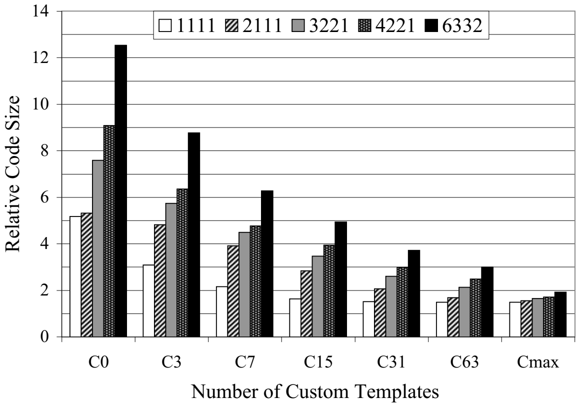
Adding custom templates, however, comes at some hardware cost. Figure 5(b) shows the incremental cost of controlpath hardware due to an increase in the number of custom templates. The most important trend is that the controlpath hardware cost increases as more custom templates are added, but not by a large amount. For the 1111 machine configuration the increase from 0 to 63 custom templates is less than 10%, and for the 6132 machine configuration it is less than 20%. This means that most of the advantages in code size shown above that were obtained by template



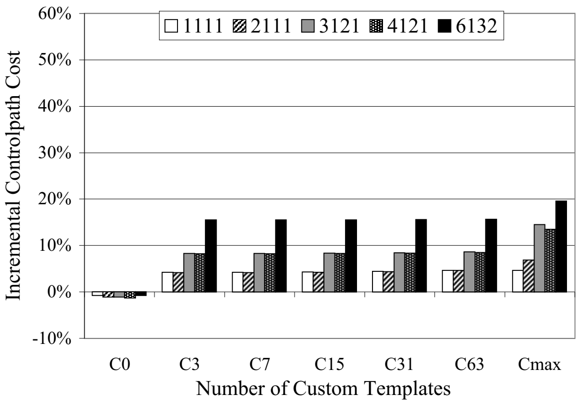
(a): Code Size (no affinity)



(b): Controlpath Cost (no affinity)



(c): Code Size (full affinity)



(d): Controlpath Cost (full affinity)

Figure 5: Effect of varying the number of custom templates for pgpdecode: (a) relative code size across various machines with no affinity, (b) incremental controlpath cost across machines with no affinity, (c) relative code size across various machines with full affinity, (d) incremental controlpath cost across machines with full affinity. The following parameters are fixed for this experiment: 1x latencies, use of multinoop, and no use of the EOP bit. The number of custom templates varies from none (C0) through 63 (C63) to the maximum number needed for that machine configuration (Cmax) upper bounded at 511 templates. The code size has been normalized to the corresponding pseudo-CISC machine. Incremental controlpath cost is expressed as a percentage of total hardware cost for the corresponding C0 machine. These numbers are based on an estimated area cost model for an 0.18u process technology.

customization come at a fairly modest cost in control hardware.

There are two major contributors to this increase in the hardware cost: the instruction fetch pipeline hardware and the instruction decode logic. The instruction fetch pipeline hardware includes the instruction alignment network before the instruction register and the instruction distribution network after the instruction register. The graph shows a jump in the control hardware cost when the number of custom templates is raised from 0 to 3. This is due to a significant increase in the cost of the instruction alignment network that is needed to prepare the instruction register for the next instruction by shifting the instruction register by the width of the current instruction (which now becomes a variable) and left-aligning the next instruction. We have reduced the complexity of this shift and align network by quantizing all templates to a multiple of a fixed quantum width (16 bits in all our experiments). Once the hardware investment is made to accommodate multiple-width templates, the cost of adding more custom templates is only due to the added complexity of the instruction distribution network and the decode logic, which is not very significant from 3 to 63 templates. Only in the extreme case of adding as many custom templates as needed (Cmax) does this cost become significant.

4.5 Code size and hardware complexity with full affinity

The cost of the instruction distribution network and decode logic becomes significant with large numbers of custom templates because packing the bits as tightly as possible for each additional custom template may cause the instruction fields controlling the same datapath control port (such as a register file address port) to be allocated at different bit positions. This introduces multiplexors into the instruction distribution path and requires additional control logic. Even though packing the templates is desirable for obtaining smaller sized templates, the hardware complexity increases controlpath cost and adds delay due to multiplexors in the critical path. Therefore, we use full affinity to ensure that all instruction fields in different templates that control the same control port are allocated to the same bit positions. This eliminates the need for any multiplexors in instruction distribution. However, it has the potential to increase the template size because fields across templates have to be aligned with each other.

Corresponding to Figures 5(a) and 5(b), Figures 5(c) and 5(d) show the effects of custom templates on code size and incremental control path cost with full affinity. Qualitatively, Figure 5(c) is similar to Figure 5(a), but an increase in code size may be observed due to full affinity: up to 12% for C0 and up to 50% for Cmax. Comparing Figure 5(d) with Figure 5(b), the important observation is that the controlpath hardware cost is generally lower with full affinity, more significantly so for higher numbers of custom templates. Indeed, the cost increment for the maximum number of custom templates (Cmax) is now only between 5% to 20% across the machines as opposed to being between 8% to 59% for the case when there was no affinity. This improvement is largely due to the almost complete elimination of the complex instruction distribution network made possible by full affinity. Note that full affinity even reduces the cost of no custom templates (C0) slightly due to the affinity across opgroups within each super group.

In short, full affinity successfully reduces instruction distribution complexity to improve critical path timing (and some controlpath hardware cost) without significantly impacting the code size. Therefore, in the remaining experiments we assume the use of full affinity with maximum number of custom templates (upper bounded at 511).

4.6 The use of the EOP bit

Figure 6 shows the effectiveness of our frequency heuristic for aligning branch targets using the EOP bit for all of the applications. The heuristic is compared against the two extreme strategies: never aligning and always aligning. The first extreme strategy, never align, never sets the EOP bit to align a branch target. This strategy achieves the smallest code size by ignoring alignment issues and packing instructions as tightly as possible. However, it suffers the maximum branch penalty because branch targets often do span multiple packets. The other extreme strategy, always align, ensures that no branch target crosses a packet boundary by aggressively setting the EOP bit whenever a crossing would have occurred. This strategy reduces the branch penalty to zero, but suffers the largest code size due to excessive alignment.

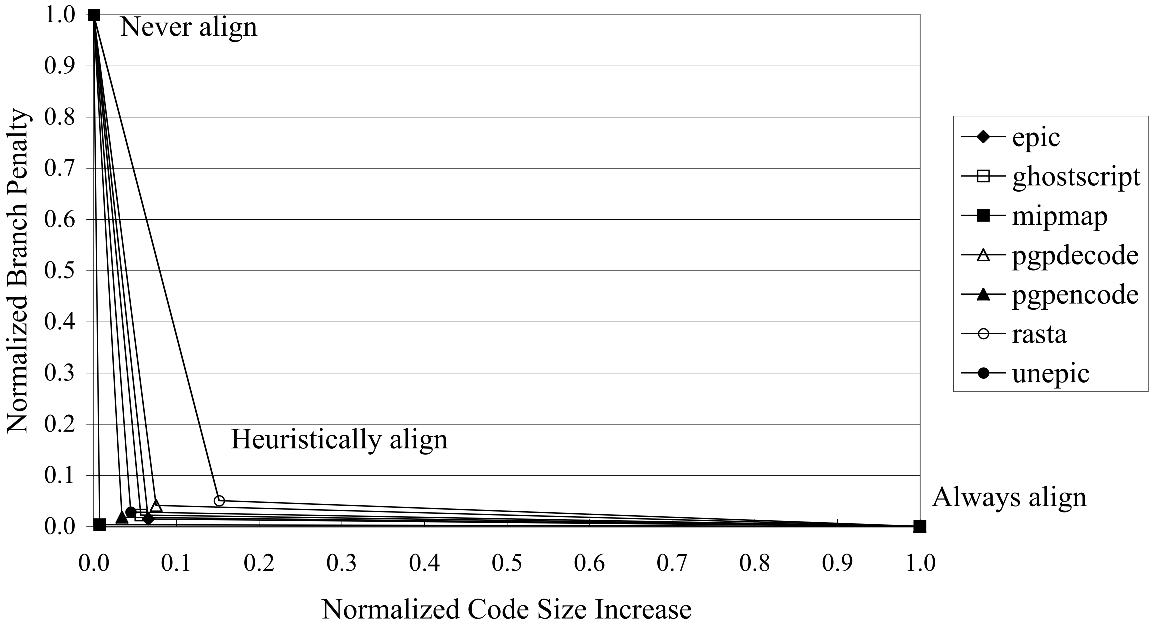


Figure 6: Effectiveness of the EOP heuristic policy in reducing the branch stall penalty without increasing the code size excessively. For this experiment, the following variables are fixed: the 6132 processor with 1x latencies, the use of the multinoop capability, the maximum number of custom templates, and full affinity. Each data point shows the normalized code size increase and the normalized branch penalty achieved with each alignment strategy for a particular application. The code size increase is defined to be the incremental change in code size above the never align case. The normalized code size increase is defined as the ratio of the code size increase to the code size increase for the always align case. The normalized branch penalty is defined to be the ratio of the branch penalty to what it would be in the never align case.

The ideal situation is to have no branch penalty with no increase in code size. Our heuristic attempts to get the best of both worlds by using alignment for a small number of important branch targets to reduce the majority of the branch penalty and no alignment for all other targets to keep the code size small. As Figure 6 demonstrates, our heuristic is very effective. The normalized branch penalty is never greater than 5%, while the normalized code size increase is not more than 15% (8% without rasta). Mipmap has almost ideal behavior with this heuristic.

4.7 The combined effectiveness of all the techniques on code size

Table 4 shows the cumulative effect of using all the features of our instruction format on code size for all processor configurations. This corresponds to what we view as a judicious balance between code size, controlpath complexity, and branch penalties. The table presents a dramatically different picture of code size as a function of processor width and latency than was measured for the canonical case (Table 3). The code size for the 1111 (4-issue) processor is now between 1.5x and 2.3x of the code size of the pseudo-CISC processor. This is about the same factor by which RISC code exceeds CISC code. Furthermore, for the 6132 (12-issue) processor, the code size ratio is also similar, ranging from 1.6x to 2.3x as compared to 5.7x to 28.8x in the canonical case. This data shows that the combination of custom templates and the multinoop field are highly effective even when full affinity and the EOP heuristic are employed (both of which tend to increase the code size to help other objectives). Together, they eliminate most of the inefficiencies in the instruction encoding for processors with higher degrees of parallelism, and thus do not incur the code bloat traditionally associated with VLIW processors.

A couple of nonintuitive behaviors are illustrated in the table. First, the code size for the 1111 processor is often larger than that for wider processors. This is because 1111 processor has less ILP than others, leading to fewer opportunities for using custom templates. Wider machines are also able to pack more operations into one instruction and hence incur less packet alignment overhead. Second, the code size for latency 2x or 3x is sometimes smaller than that for latency 1x especially for very wide (6132) processors. This behavior is somewhat subtle and is due to the nature of the resultant schedules for processors with short and long latencies. Generally, with short latencies there is a more uniform distribution of parallelism within a scheduling region. As the latencies are increased, the code becomes more bimodal with bursts of parallel operations followed by sequential operations. Since we upper bound the number of custom templates at 511, the fixed number of templates are more effective at capturing the scheduling patterns in bimodal pattern of operations than with the uniform distribution of operations.

Application	Latency	Processor Width				
		1111	**2111**	**3121**	**4121**	**6132**
epic	1x	2.13	1.63	1.73	1.74	1.90
	2x	2.22	1.77	1.86	1.87	1.91
	3x	2.25	1.85	1.91	1.83	1.88
ghostscript	1x	2.01	1.79	1.89	2.02	2.28
	2x	2.17	2.06	1.94	2.09	2.28
	3x	2.24	2.16	2.00	2.04	2.29
mipmap	1x	1.98	1.73	1.96	1.90	2.11
	2x	2.17	1.99	2.00	2.01	2.09
	3x	2.24	1.99	2.05	2.09	2.08
pgpdecode	1x	1.46	1.48	1.58	1.62	1.83
	2x	1.67	1.60	1.65	1.64	1.78
	3x	1.70	1.64	1.66	1.65	1.78
pgpencode	1x	1.46	1.47	1.57	1.64	1.82
	2x	1.67	1.59	1.64	1.64	1.77
	3x	1.70	1.63	1.65	1.64	1.77
rasta	1x	1.96	1.55	1.71	1.71	1.90
	2x	2.09	1.68	1.75	1.73	1.90
	3x	2.15	1.74	1.83	1.83	1.91
unepic	1x	2.19	1.57	1.52	1.54	1.64
	2x	2.23	1.69	1.70	1.64	1.71
	3x	2.25	1.75	1.73	1.65	1.69

Table 4: Code size using the multinoop field, maximum number of custom templates with full affinity and the EOP bit as a function of processor width and functional unit latency. Each cell of the table contains the ratio of the code size achieved for a processor-latency pair to that of the pseudo-CISC processor for each application.

5 Related Work

In the context of VLIW processors, techniques for minimizing code size fall broadly into two categories: decompress-on-miss and decompress-on-hit. In the first category, instructions are represented in compressed form in memory, but in decompressed form in the instruction cache. Decompression is performed at the time of an instruction cache miss while instructions are being transferred from memory into the instruction cache. This is the approach used in the Multiflow architecture [5]. The drawback of this scheme is that the instruction cache capacity is wasted. On the other hand, since the decompression penalty is experienced only on a cache miss, relatively complex compression methods can be employed. However, all prior work of this type has been restricted to the simpler case of sequential processors [18, 10, 12].

The alternative is decompress-on-hit; instructions, both in memory and the instruction cache, are kept in their compressed form and are expanded each time they are fetched from cache [4, 6, 8, 1, 7]. In our opinion, if the depth of the instruction pipeline is not too deep, the compression schemes must have limited complexity, such as the custom templates described in this paper. The AVIV retargetable code generator [7] also uses a fixed set of variable-width templates to minimize code size, but the templates are specified beforehand and do not use program scheduling statistics. In contrast, the Tinker project [6] uses a compression scheme that is similar to the Multiflow approach [5], except that the decompression occurs on every instruction fetch from the cache. In effect, this makes available every possible custom template, but it also has the corresponding controlpath complexity. Our approach is to limit controlpath complexity by limiting ourselves to a small, carefully selected subset of all possible templates.

There are other techniques, orthogonal and complementary to those discussed in this paper that contribute to a reduction in code size. One such technique is to recognize sequences of operations that show up repeatedly in the application of interest and to replace them with a new, complex opcode [17, 13, 3]. Another well-known technique from the microprogramming literature is the use of residual control to make certain operands or opcode qualifiers implicit.

6 Conclusions

With the canonical instruction format, the code size for VLIW and EPIC processors is extremely sensitive to both processor width and operation latency. In this paper, we presented two techniques to address this issue. Custom templates reduce code size for a given width, while the multinoop capability reduces code size by an amount proportional to the latency. Together, they make code size relatively insensitive to both width and latency.

We also presented two other techniques, affinity allocation and the profile-based use of the EOP bit, that trade off some of the reduction in code size for, instead, reducing the controlpath complexity due to custom templates and the run-time stall penalty for fetching branch target instructions, respectively. Affinity allocation, while significantly reducing the controlpath cost for large numbers of custom templates (e.g., 59% to 20% for the 6132 processor at Cmax templates), also provides better hardware timing properties due to reduced multiplexing in the instruction distribution network. Likewise, our profile-based heuristic controlling the use of the EOP bit reduced the branch stall penalty to no more than 5% of what it would have been, without increasing the code size by more than 15% (8% without rasta).

In concert, the design-time and assembly-time techniques described in this paper allow the design of extremely wide-issue and deeply pipelined VLIW and EPIC processors whose code size, controlpath complexity, and run-time branch penalty are entirely acceptable. Across all seven applications considered, with up to 511 custom templates, full affinity allocation, and the use of the EOP heuristic, the code size relative to an abstract CISC processor is between 1.5x and 2.3x for the 1111 VLIW processor and between 1.6x and 2.3x for the 6132 processor even with 3x latencies. This increase is comparable to that for a RISC processor.

A second focus of this paper was to describe the structure of a machine-description driven assembler for EPIC and VLIW processors that implements some of the code minimization techniques described above. The assembler uses a finite and well-defined set of external database queries to access all of the necessary information regarding the target processor including its instruction format. Consequently, such an assembler needs to concern itself only with the policies and heuristics for generating compact code.

Acknowledgements

The authors would like to acknowledge Richard Johnson's contributions to PICOVLIW's instruction format design capability upon which this work is based.

References

[1] S. Aditya, B. R. Rau, and R. C. Johnson. Automatic design of VLIW and EPIC instruction formats. HPL Technical Report HPL-1999-94, Hewlett-Packard Laboratories, 2000.

[2] S. Aditya, B. R. Rau, and V. Kathail. Automatic architectural synthesis of VLIW and EPIC processors. In *International Symposium on System Synthesis, ISSS'99*, pages 107–113, San Jose, California, 1999. IEEE Computer Society.

[3] M. Arnold and H. Corporaal. Instruction set synthesis using operation pattern detection. In *Fifth Annual Conf. of ASCI*, Heijen, The Netherlands, 1999.

[4] G. R. Beck, D. W. L. Yen, and T. L. Anderson. The cydra 5 mini-supercomputer: architecture and implementation. *The Journal of Supercomputing*, 7(1/2):143–180, 1993.

[5] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. A vliw architecture for a trace scheduling compiler. *IEEE Transactions on Computers*, 37(8):967–979, 1988.

[6] T. M. Conte, S. Banerjia, S. Y. Larin, K. N. Menezes, and S. W. Sathaye. Instruction fetch mechanisms for VLIW architectures with compressed encodings. In *29th International Symposium on Microarchitecture*, pages 201–211, 1996.

[7] S. Hanono and S. Devadas. Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator. In *35th Design Automation Conference*, pages 510–515, 1998.

[8] Intel Corporation. *IA-64 Application Developer's Architecture Guide*. 1999.

[9] V. Kathail, M. Schlansker, and B. R. Rau. HPL-PD architecture specification: Version 1.1. Technical Report HPL-93-80 (R.1), Hewlett-Packard Laboratories, 2000.

[10] M. Kozuch and A. Wolfe. Compression of embedded system programs. In *IEEE International Conference on Computer Design*, pages 270–277, 1994.

[11] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communication systems. In *30th Annual International Symposium on Microarchitecture (MICRO-30)*. ACM and IEEE Computer Society, 1997.

[12] S. Y. Liao, S. Devadas, and K. Keutzer. Code density optimization for embedded DSP processors using data compression techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(7):601–608, 1998.

[13] Philips Semiconductors. *Trimedia TM-1 Media Processor Data Book*. 1997.

[14] B. R. Rau. Cydra 5 directed dataflow architecture. In *COMPCON '88*, pages 106–113, San Francisco, 1988.

[15] 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):71–118, 1999.

[16] M. S. Schlansker and B. R. Rau. EPIC: Explicitly parallel instruction computing. *Computer*, 33(2):37–45, 2000.

[17] J. Van Praet, G. Goossens, D. Lanneer, and H. De Man. Instruction set definition and instruction selection for ASIPs. In *7th ACM/IEEE International Symposium on High-Level Synthesis*, pages 11–16, Niagara-on-the-Lake, Ontario, Canada, 1994. IEEE.

[18] A. Wolfe and A Chanin. Executing compressed programs on an embedded risc architecture. In *The 25th Annual International Symposium on Microarchitecture (MICRO-25)*. IEEE, 1992.