A New Technique for Instruction Encoding in High Performance Architectures

R. F. Batistella *R. R. Santos*
R. J. Azevedo

Technical Report - IC-07-027 - Relatório Técnico

September - 2007 - Setembro

A New Technique for Instruction Encoding in High Performance Architectures

Rafael Fernandes Batistella   Ricardo Ribeiro dos Santos
Rodolfo Jardim de Azevedo

Abstract

In this paper we propose a new technique to reduce the program footprint and the instruction fetch latency in high performance architectures adopting long instruction in the memory. Our technique is based on an algorithm that factors long instructions into encoded instructions and instruction patterns. An encoded instruction contains no redundant data and it is stored into an I-cache. The instruction patterns, on the other hand, look like a map to the decode logic to prepare the instruction to be executed in the execution stages. These patterns are stored into a new cache, named Pattern cache (P-cache). The technique has shown a suitable alternative to well-known architectural styles such as VLIW and EPIC architectures. We have carried out a case study of this technique in a high performance architecture called 2D-VLIW. We have evaluated the performance of our encoding technique through trace-driven experiments with MediaBench, SPECint00, and SPECfp programs. Moreover, we have compared the execution time performance of the 2D-VLIW architecture with encoded instructions to the same architecture with non-encoded instructions. Further experiments compare our approach to VLIW and EPIC instruction encoding techniques. Experimental results reveal that our encoding strategy provides a program execution time that is up to 23× (average of 5×) faster than a 2D-VLIW non-encoded program. The results also show that the program code region, by using encoded instructions, is up to 78% (average of 69%) smaller when compared to a 2D-VLIW program using non-encoded instructions.

1 Introduction

It is well known that the rate of improvement in microprocessor speed exceeds the rate of improvement in DRAM memory speed. Processor speed has been rising dramatically at approximately 80% per year, while DRAM speed increases at 7% per year. However, the mainstream computer architecture community is still largely focused on increasing processor performance [13]. As a result, the difference between processor and memory speed has increased exponentially leading to a phenomenon known as the *Memory Wall* [13]. Recently, several companies announced that the processor frequency will not raise as it used to. Even so, Memory Wall will be still a concern for the computer architecture community.

A large number of techniques have addressed the Memory Wall problem. Many of them have focused on compression techniques as an alternative to reduce the amount of data to be stored into the main memory. Specifically, compression techniques have addressed architectures which fetch large instructions in memory and architectures targeted to specific application domains like embedded systems [14, 16, 17, 18].

This paper proposes a new approach to deal with the overhead of the instruction fetch latency and its impact on the program performance. Specifically, the paper presents a new instruction encoding technique targeted to architectures that store long instructions in memory. The technique is comprised of an encoding algorithm and a cache memory called Pattern cache (P-cache). The algorithm is called LIF (Large-Instruction Factorization algorithm) and is based on the operand factorization technique [1, 6, 8]. After the instruction scheduling and register allocation activities of a back-end compiler, the algorithm extracts redundant operands from long instructions, thus creating a new (encoded) instruction with non-redundant operands. This encoded instruction is stored into an I-cache. The algorithm keeps track of the operands original position by creating an instruction pattern that is stored into the P-cache. The pattern is a data structure that looks like a map for preparing the instruction to be executed.

A processor architecture adopting this technique fetches short encoded instructions from the I-cache and, at the decode stage, it fetches an instruction pattern from the P-cache. It is important to notice that the fetch latencies of the I-cache and the P-cache will not be the same for every fetch. Misses in the I-cache does not imply misses in the P-cache once patterns can be reused by different instructions, i.e., there is a surjection between the encoded instruction set and the pattern set. After the decode stage, an instruction can execute its operations in the processing elements (or functional units) of the processor.

We have evaluated the impact of this technique through a trace-driven simulation on I-caches and P-caches with MediaBench, SPECint00, and SPECfp programs. Our results show that, by adopting this encoding strategy, the performance (execution time) is up to 23× better than I-caches used by a 2D-VLIW non-encoded strategy, up to 83× better than I-caches used by a VLIW strategy and up to 3× better than I-caches used by an EPIC strategy. Our experiments also show that, when using encoded instructions, the size of the I-cache plus the P-cache size is up to 78% smaller than I-caches of non-encoded instructions, up to 90% smaller than I-caches of VLIW instructions, up to 38% smaller than I-caches of EPIC instructions, and up to 27% smaller than I-caches of IA64 instructions.

We outline the related work in Section 2. In Section 3 we present a general description of our encoding technique showing how it works. The LIF algorithm is discussed in Section 4. An implementation of this encoding over a multiple-issue processor architecture is presented in Section 5. Section 6 shows the results of our technique through static and dynamic experiments. Finally, Section 7 presents some concluding remarks.

2 Related Work

Previous works focusing on instruction size reduction have used concepts and techniques from the code compression area. There are several proposals [15, 16, 18, 26, 27, 28] describing compression techniques for instructions focused in VLIW (Very Long Instruction Word) architectures. However, some of these proposals try to improve the program compression ratio at the expense of the decompression overhead in the processor performance [27].

For example, in [15] a dictionary-based code compression using the instruction word isomorphism is presented. The authors attained a compression ratio of 63% in SPECint95 programs. Their approach consists of selecting the most frequent instructions and splitting up operands and opcodes into two dictionaries. The decode logic adds a new stage on the processor datapath. Our approach, on the other hand, allows the decode to take place in parallel to other datapath activities.

In [23], the authors add several levels of cache to the memory system to minimize the memory latency. They combine latency hiding techniques such as prefetching and memory speculation in a high performance processor in order to achieve reasonable efficiency. Conversely, our approach focuses on encoding long instructions to reduce the memory latency. Prefetching and speculation techniques are independent of our technique and all of them may be implemented by the target architecture.

It is important to observe that our approach is different from strategies that reduce the number of instructions of a program, such as Instruction Collapsing [10, 21, 22]. In this strategy, the instruction dependence chains are analyzed and a set of dependent instructions are put together in only one collapsed instruction. In [21], the experiments show that by collapsing dependent instructions, we can reduce the need for fast issue, quick bypass, and large instruction windows. Our encoding approach is focused on exploring the surjection between instruction and its pattern in order to decrease the instruction size in the memory.

Similar to the approach used in [1], the LIF algorithm factors long instructions into encoded instructions and instructions patterns. However, our proposal differs from theirs in two key aspects. First, we store the factored patterns into a cache memory that can be accessed in parallel to other datapath activities. Second, our building strategy is simpler than traditional decompression mechanisms since the instruction on memory has a tag which points to the P-cache line where its whole pattern is stored. At the decode stage, the encoded instruction and its pattern are used to prepare the instruction to be executed in the execution stages.

3 Understanding the Instruction Encoding Technique

Like traditional compression techniques based on operand factorization [1, 6, 8], our encoding strategy traverses program instructions factoring redundant operands and opcodes from these instructions into two elements: encoded instructions and patterns. This strategy leads to a dramatic instruction-size reduction since redundant data does not appear in the instructions of the program anymore. An *encoded instruction* has no redundant data (registers or immediates) inside it. It is stored into the I-cache as an usual program instruction. A *pattern* (or instruction pattern) is a data structure that contains pointers to the positions in the encoded instruction. These pointers are used as a map to prepare the instruction to be executed in the execution stages. A pattern is stored into a new cache, called P-cache. Figure 1 illustrates the execution flow of the technique to obtain an encoded instruction and the pattern.

One could consider that this strategy impacts on the final performance since I-cache misses imply misses in the P-cache. However, previous works [3, 25] have already demon-

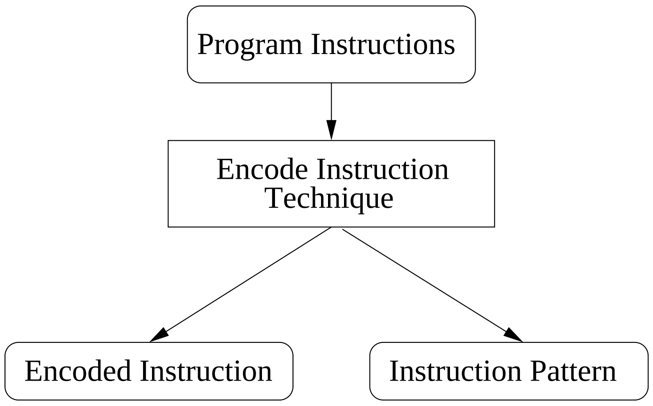


Figure 1: Execution flow of the encoding technique.

strated that many distinct instructions reuse the same pattern all over again. Thus, an ideal P-cache organization has an important feature: an I-cache miss should not imply a P-cache miss. This reuse can be modelled as a surjection function between encoded instructions and their patterns, in such a way that there is a surjection from elements (instructions) in an I-cache set SI to elements (instruction patterns) in a P-cache set SP. Thus,

$$a, b, c, d \in SI, \text{ such that } a,\ b,\ c,\ d \text{ are encoded instructions}$$
$$p \in SP, \text{ where } p \text{ is an instruction pattern}$$
$$\exists \text{ a mapping } f \text{ such that } f(a) = f(b) = f(c) = f(d) = p$$

This is the major motivation and the key aspect for factoring out patterns from the long instructions. The factored patterns can be reused all over again by different encoded instructions. A large reuse of the patterns make it possible to reduce the code region of a program since the encoded instruction size is smaller, sometimes much smaller, than a non-encoded instruction and the amount of patterns should be smaller than the numbers of encoded instructions.

In the encoding technique design, we think on the P-cache as an element of the processor datapath. In other words, fetching patterns in the P-cache does not add a new stage on the processor datapath. Actually, the encoded instruction format takes advantage of the P-cache as another element of the existent datapath and allows to perform other activities while a pattern is fetched. Moreover, the encoding technique does not restrict the encoded instruction and the instruction pattern to a specific size. The area of the encoded instructions and patterns only depend on the architectural requirements. The technique only arranges the data (operands and opcodes) in order to meet this requirement.

4 The LIF Algorithm

In order to perform other activities (e.g. read operands from the register file) while an encoded instruction fetches its pattern in the P-cache, the LIF algorithm splits up an encoded instruction into read register operands and write/immediate operands. For example, Figure 2(a) presents a code fragment composed of four operations in a MIPS-like assembly language. Figure 2(b) illustrates the corresponding long instruction. We can notice that this instruction looks like a common VLIW instruction word. In this example, an immediate occupies more bits than a register value and due to this, an immediate uses two instruction fields.

```
add r1, r2, r3
addu r4, r2, r6
addi r7, r6, 9
subu r9, r10, r6
```

(a) Code fragment.

add	r1	r2	r3	addu	r4	r2	r6	addi	r7	r6	0	9	subu	r9	r10	r6

(b) Instruction comprised of the operations from 2(a).

Figure 2: Code fragment and the respective long instruction.

Figure 3 depicts all steps to build an encoded instruction and the pattern by using the LIF algorithm over the code fragment in 2(a). Figures 3(a)-3(d) show the current state of the encoded instruction (up) and the pattern (down) after each step of the algorithm. Basically, the algorithm builds an encoded instruction and its respective pattern on a per operation basis. After looking at each operation, the algorithm updates the encoded instruction and the pattern. Take for example operation `add r1, r2, r3` in 2(a). First, opcode `add` is put into the pattern. After that, the first register (`r1`), which is the output register, is stored into field 6 of the encoded instruction (in this example, fields 0 to 5 are reserved for register file read ports) and a pointer to field 6 is stored into the pattern. The encoding of the two read registers (`r2` and `r3`) follows similar steps, but uses the reserved fields for register file read port. Notice that the second operation `addu r4, r2, r6` also uses register `r2`. Therefore, field 0 will be reused. After 3(d), a pattern tag, which points to the pattern address, is added to the encoded instruction.

The decoding process is very simple: after a pattern is fetched from the P-cache, the decoding logic uses the pointers in the pattern and the encoded instruction brought from the I-cache to build a complete instruction, while the input registers are being read from the register file (remember that every register that must be read are identified in the first 6 fields, so they can be read while the instruction is being decoded). During the decoding, the encoded instruction works as an operations dictionary to the pattern pointers.

Our technique considerably differs from the code compression approaches since we do not use neither a complex decompression hardware nor a large operand/operation dictionary. Instead, we use a new cache that stores the most used patterns. Our encoding strategy

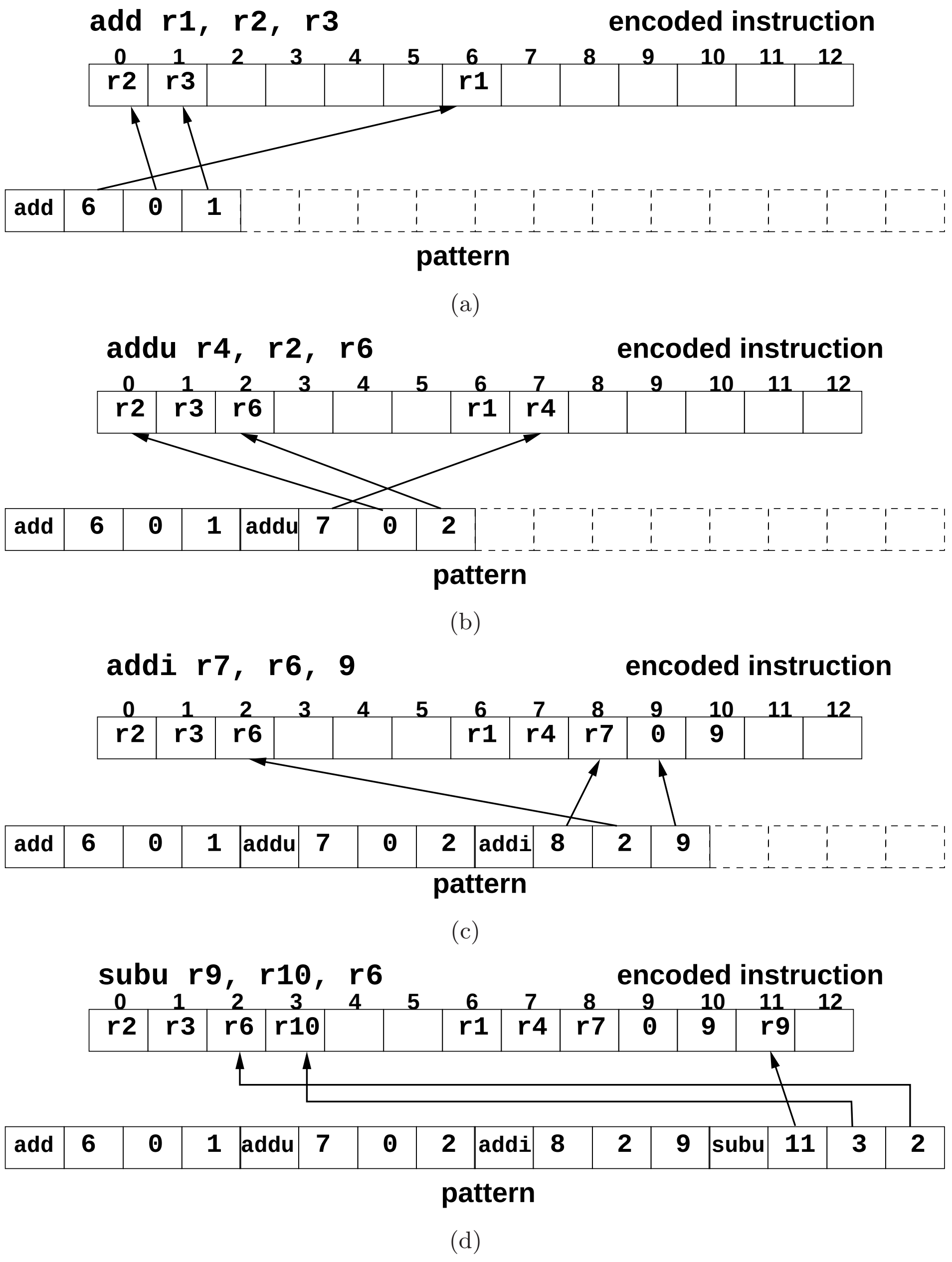
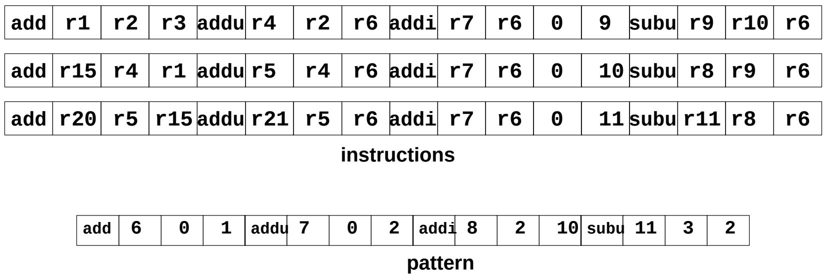


Figure 3: Execution steps of the LIF algorithm.

also allows the datapath to perform other activities (read the input registers) while the instruction is being decoded.

One can notice that the fields in a pattern store pointers to specific fields of the encoded instruction, while long instruction fields store the number of the operand register and immediate values. This feature makes a pattern reusable for different instructions. Figure 4 shows an unique pattern which is reusable for three different instructions. Furthermore, another attractive feature is that the LIF algorithm does not care about the operation dependencies in the instruction. These dependencies are not checked by the algorithm since they depend on the architecture and they should be solved either statically (by the compiler) or dynamically (by the hardware).



add	r1	r2	r3	addu	r4	r2	r6	addi	r7	r6	0	9	subu	r9	r10	r6
add	r15	r4	r1	addu	r5	r4	r6	addi	r7	r6	0	10	subu	r8	r9	r6
add	r20	r5	r15	addu	r21	r5	r6	addi	r7	r6	0	11	subu	r11	r8	r6

instructions

add	6	0	1	addu	7	0	2	addi	8	2	10	subu	11	3	2

pattern

Figure 4: Instructions sharing the same pattern.

We can understand this encoded instruction as a CISC (Complex Instruction Set Computer) instruction which is, internally, composed of RISC (Reduced Instruction Set Computer) operations. The same pattern can be reused several times for different CISC instructions in different places of the program. The encoded instruction is the equivalent of a CISC instruction. The RISC operations are represented by the operations stored in the pattern. The pattern can be seen as the microcode required to run the CISC instruction. This microcode obtains its parameters from the CISC instruction fields.

Algorithm 1 presents the main steps of the LIF algorithm. For simplicity, some parts of the algorithm such as: finding out similar patterns and adding write registers to the pattern, were left out. The input to this algorithm is a set of instructions (SI) with scheduling and register allocation already performed. Two sets are available on the output: the set of encoded instructions (SE) and the set of patterns (SP).

For each instruction S of set SI (line 2), the algorithm analyzes the instruction content (line 4) looking whether operands $op.opnd1$ and $op.opnd2$ of the current operation op are in the encoded instruction I (lines 5 and 15, respectively). If they are, pointers must be added to pattern P in order to indicate which are the correct operands of operation op (lines 13, 14, and 23). If, due to some constraint of the architecture, operations cannot be encoded into the current instruction I, these operations will be encoded into a new instruction (lines 7-9, 17-19). The algorithm has two important loops: the first one is used to obtain all the instructions of the SI set; the second one verifies each operation of an instruction S selected in the previous loop. As the amount of operations in an instruction is constant, the worst-case complexity of the algorithm is limited by the size of the SI set and the search step to look for an existent pattern into the SP set. Once the set of patterns can be, at most, at the same size of SI, the final complexity is $\mathcal{O}(|SI|^2)$.

Algorithm 1 Encoding algorithm.

```
INPUT: Set of instructions $SI$.
OUTPUT: Set of encoded instructions $SE$ and patterns $SP$.
Encoding(SET_INST: $SI$)
1)    create new $SP'$;
2) for $S \in SI$
3)    create new $I$; create new $P$;
4)    for $op \in S$
5)       if $op.opnd1 \notin I$
6)          if $free\_space(I) < 1$
7)             $SE = SE \cup I$;
8)             $SP' = SP' \cup P$;
9)             create new $I$; create new $P$;
10)         end if
11)         $I.add(op.opnd1)$;
12)      end if
13)      $P.add\_op(op.opcode)$;
14)      $P.add\_opnd =$ location of $op.opnd1$ in $I$;
15)      if $op.opnd2 \notin I$
16)         if $free\_space(I) < 1$
17)            $SE = SE \cup I$;
18)            $SP = SP \cup P$;
19)            create new $I$; create new $P$;
20)         end if
21)         $I.add(op.opnd2)$;
22)      end if
23)      $P.add\_opnd =$ location of $op.opnd2$ in $I$;
24)   end for
25)   $SP' = SP' \cup P$;
26)   $SE = SE \cup I$;
27)   if (not_exists_pattern($SP', SP$))
28)      $SP = SP \cup SP'$;
29) end for
```

5 A Case Study: The 2D-VLIW Architecture

We have implemented our encoding strategy (the LIF algorithm and the P-cache) in a high performance architecture called 2D-VLIW [19, 20]. Originally, this architecture fetches long instructions from the memory where the number of operations inside the instruction is equivalent to the number of functional units in the architecture. The operations of a long 2D-VLIW instruction runs on a matrix of functional units (FU matrix) and they can read and write values onto two register files: global register files (r), located in the decode stage, and temporary registers (tr) that are spread along the matrix. Figure 5 shows an overview of this architecture with and without a P-cache. Figure 5(a) depicts the original 2D-VLIW datapath where the encoding technique is not used. Figure 5(b) shows the 2D-VLIW datapath with the P-cache included.

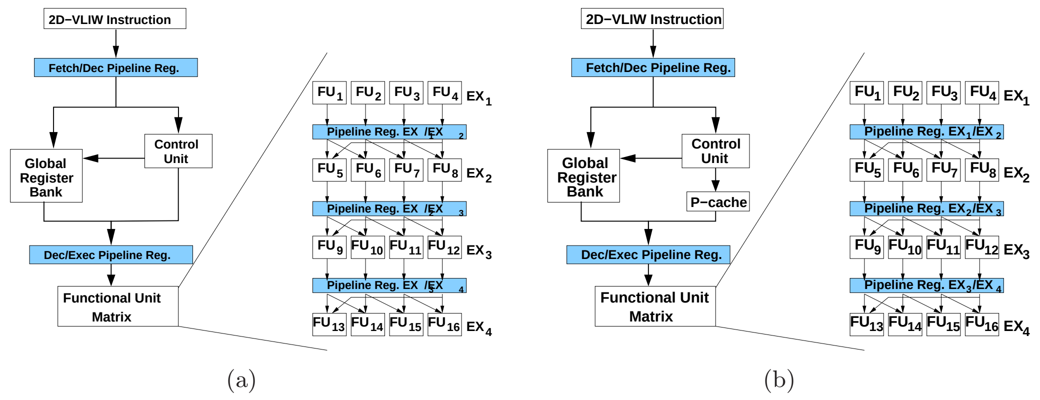
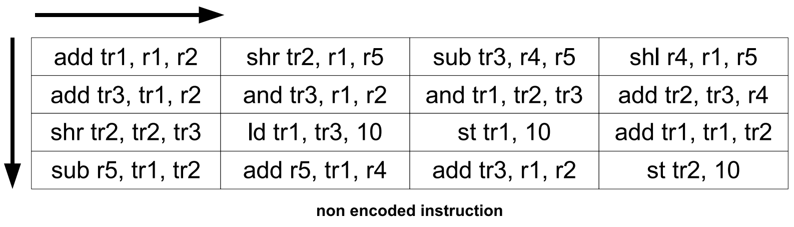


Figure 5: Original 2D-VLIW datapath and the addition of the P-cache.

This architecture executes the instructions in a pipeline style. At each stage, an instruction is fetched from the memory. In Figure 5(a), at the decode stage, the operand read registers coming from the instruction go to the register file bank and, after that, they are sent to the matrix of functional units. In Figure 5(b), at the decode stage, patterns are searched in the P-cache while registers are read from the global register bank. These patterns and data from the encoded instruction are put together to compose a complete instruction that will be executed onto the FU matrix.

The 2D-VLIW architecture fetches fixed-size instructions in the memory. The instructions are comprised of dependent and independent operations. The compiler is responsible for handling these dependencies between operations inside an instruction. Considering the architecture presented in Figure 5(a), each 2D-VLIW instruction has 16 operations (4×4 FUs) that can read up to 32 global registers (2 read registers per operations $\times 16$). However, the 2D-VLIW architectural design allows just 8 global registers ($2 \times \sqrt{16}$) to be read. Using just 8 global read registers per instruction, this would generate many sparse instructions and, as a result, an unnecessary waste of memory. One solution is to apply the encoding technique to encode 2D-VLIW instructions. The encoding technique can minimize the waste of memory by reducing the program footprint and, most important, maximizing the performance of the instruction fetch stage since the architecture will fetch short and compact encoded instructions.

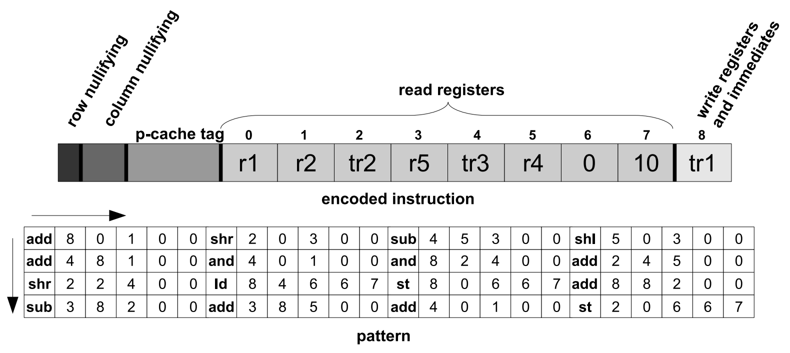
Figure 6 shows how a code fragment is organized in a simple 2D-VLIW long instruction and, after running the LIF algorithm, an encoded 2D-VLIW instruction. For the sake of simplicity, we present the 2D-VLIW instruction as a matrix of operations where each cell represents one operation that is executed by one functional unit of the FU matrix. The arrows along the top and left side indicate the operations order used by the LIF algorithm to build the encoded instruction. After all steps of this algorithm over the instructions in 6(a), we obtain an encoded 2D-VLIW instruction and the pattern in 6(b). It is important to observe that by using the encoding technique over the 2D-VLIW architectures, only the encoded instructions and patterns will exist after the LIF execution. The long 2D-VLIW instruction will not exist in the final program code.



add tr1, r1, r2	shr tr2, r1, r5	sub tr3, r4, r5	shl r4, r1, r5
add tr3, tr1, r2	and tr3, r1, r2	and tr1, tr2, tr3	add tr2, tr3, r4
shr tr2, tr2, tr3	ld tr1, tr3, 10	st tr1, 10	add tr1, tr1, tr2
sub r5, tr1, tr2	add r5, tr1, r4	add tr3, r1, r2	st tr2, 10

non encoded instruction

(a)



	0	1	2	3	4	5	6	7	8
	r1	r2	tr2	r5	tr3	r4	0	10	tr1

add	8	0	1	0	0	shr	2	0	3	0	0	sub	4	5	3	0	0	shl	5	0	3	0	0
add	4	8	1	0	0	and	4	0	1	0	0	and	8	2	4	0	0	add	2	4	5	0	0
shr	2	2	4	0	0	ld	8	4	6	6	7	st	8	0	6	6	7	add	8	8	2	0	0
sub	3	8	2	0	0	add	3	8	5	0	0	add	4	0	1	0	0	st	2	0	6	6	7

(b)

Figure 6: Non-encoded 2D-VLIW instruction 6(a). Encoded 2D-VLIW instruction and its pattern 6(b).

Taking into account that an original 2D-VLIW non-encoded instruction have 16 32-bits operations, the instruction in 6(a) has 512 bits. On the other hand, an encoded 2D-VLIW instruction has 64 bits divided into 8 5-bits fields that can contain read registers, 1 5-bits field for write operands and immediate values only, 1 13-bits field for the pattern address, 2 bits for row nullifying and 4 bits for column nullifying. The numbers of fields for the read and write registers were chosen according to previous experiments we have performed with SPECint, SPECfp and MEDIABench programs. The field for the pattern address (p-cache tag) just points to the instruction pattern in the P-cache. The 6 bits for row and column nullifying of the encoded instruction are responsible for indicating when a column and/or a row of the FU matrix cannot execute the operations due to pattern joins. Each bit of the row nullifying indicates if either the first 8 operations of the pattern (two first rows of the matrix) must be nullified or the last 8 operations (two last rows of the matrix). Each bit of the column nullifying nullifies one column of the matrix, i.e., it indicates if the set of operations that would execute in one column of the matrix must be nullified.

Pattern join is an optimization that joins two patterns into one when the sum of operations in both patterns is less than 16. The instructions mapping to the old patterns should now point to the new joined pattern. Most important, each instruction should nullify operations, in the new pattern, that are not used during its execution. This optimization runs after the execution of the LIF algorithm and it does not impact on the encoding technique performance since it is based on a comparison of each element of the SP set to the rest of its elements, thus leading to a complexity of $\mathcal{O}(|SP|^2)$. An encoded instruction has a smaller memory footprint, like CISC instructions, since it is 8× smaller than the non-encoded counterpart.

The 2D-VLIW pattern has 448 bits divided into 16 28-bits fields. Each one is divided into one 8-bit opcode field and 5 4-bits operand fields: three fields represent the write (1 field) and read (2 fields) operands. Moreover, there are two fields that are used to represent immediates. Each operand field in the pattern points to one of the nine fields presented in the encoded instruction. Notice that the total size (64 + 448 bits) is still the same of a non-encoded instruction. However, the surjection between instructions and patterns allows for a code size reduction.

6 Experiments and Results

In this section we present results of the experiments using our instruction encoding strategy on the 2D-VLIW architecture. In order to measure the benefits of our technique, we compare it to four other approaches: non-encoding instructions of the 2D-VLIW architecture, encoding instruction technique of common VLIW-like processors [7], encoding instruction strategy of the EPIC-like processors (where every operation has a bit telling whether it can be executed together with the previous one) [24], and encoding instruction strategy of the IA64 processors [4]. Each encoding strategy considers the constraints of its base machine. For example, unlike the 2D-VLIW encoding, the 2D-VLIW non-encoding does not take the read register constraint (only 8 read registers are available for each instruction) into account. The same happens to VLIW, EPIC and IA64 instructions. The instructions for all the encoding techniques (including our approach) were obtained using the same compiler parameters. There are two kinds of experiments: static and dynamic.

The static experiments indicate the number of instructions obtained in each encoding technique, the number of patterns (only for our technique), the reuse ratio and the reduction factor. The dynamic experiments compare the program execution time, considering the I-cache performance, to each instruction encoding technique. The static experiments were carried out on 15 different programs of the MediaBench (epic, g721decode, g721encode, gsmdecode, gsmencode and pegwit) [12], SPECfp (168wupwise, 179art and 183equake) and SPECint00 (175vpr, 181mcf, 197parser, 255vortex, 256bzip2 and 300twolf) [9] benchmarks. The experiments were performed using the Trimaran simulation infrastructure [2] to simulate the programs execution and to obtain the programs traces. We have also used the Dinero cache simulator [5] to simulate I-cache and P-cache performance. For all the encoding strategies but IA64, we consider each operation has 32 bits and one instruction contains up to 16 operations. The IA64 instructions are only comprised of 3 41 bits operations [4].

6.1 Static Evaluation

The first experiment compares the program code size of the 2D-VLIW encoding strategy to the other approaches. This experiment is performed in order to confirm the surjection between instructions and patterns previously declared in Section 3. In Table 1, columns **NonEnc**, **VLIW**, **EPIC**, **IA64**, and **Encoded** show the number of non-encoded 2D-VLIW, VLIW-like, EPIC-like, IA64, and Encoded instructions. Column **Patterns** is the total number of patterns after the pattern join optimization. Column **Reuse** shows the average number of encoded instructions using the same pattern. Finally, columns RF_x represent the reduction factor of programs using our encoding technique over the other approaches. The values of the RF_x columns were calculated according to Equation 1:

$$RF_x = 1 - \frac{((Encoded \times 64) + (Patterns \times 448))}{(Instructions_x \times Bits_x)} \tag{1}$$

where:

- x assumes value n when compared to 2D-VLIW non-encoded, e when compared to EPIC, v when compared to VLIW, and i when compared to IA64.
- **Encoded** is the number of encoded instructions obtained after all the steps of the LIF algorithm. Each encoded instruction has 64 bits.
- **Patterns** is the number of instruction patterns obtained by the LIF algorithm. Each pattern has 448 bits.
- **Instructions**$_x$ is the number of 2D-VLIW non-encoded, VLIW, EPIC, or IA64 instructions, according to x.
- **Bits**$_x$ is the length of a 2D-VLIW non-encoded, VLIW, EPIC, or IA64 instruction in bits, according to x.

The non-encoded 2D-VLIW instructions are obtained by a scheduling algorithm which greedily tries to put operations into the same instruction. This scheduling algorithm maximizes the occupation of one instruction. One non-encoded 2D-VLIW instruction has 512 bits to store 16 operations.

The VLIW-like instructions are composed by 16 independent (parallel) operations. One VLIW-like instruction has 512 bits to store 16 operations. NOPs are inserted when there are less than 16 independent operations in the instruction.

Programs	**NonEnc**	**VLIW**	**EPIC**	**IA64**	**Encoded**	**Patterns**	**Reuse**	RF_n	RF_v	RF_e	RF_i
168wupwise	6,191	13,273	1,534	7,103	9,423	1,114	8.46	0.65	0.84	-0.36	-0.21
175vpr	36,475	72,635	10,075	44,346	49,847	4,593	10.85	0.72	0.86	0.01	0.08
179art	7,267	14,443	1,751	8,381	10,993	1,214	9.06	0.66	0.83	-0.35	-0.16
181mcf	5,412	13,943	1,797	8,145	7,150	982	7.28	0.68	0.87	0.05	0.14
183equake	6,886	13,814	1,715	8,078	10,318	1,357	7.60	0.64	0.82	-0.40	-0.23
197parser	38,623	74,758	11,475	49,827	51,709	4,876	10.60	0.72	0.86	0.09	0.14
255vortex	73,378	121,769	25,449	89,943	102,970	3,950	26.07	0.78	0.87	0.38	0.27
256bzip2	15,829	28,105	3,989	17,819	20,660	2,111	9.79	0.72	0.84	-0.08	0.01
300twolf	55,991	116,329	16,133	66,942	74,850	5,108	14.65	0.75	0.88	0.17	0.17
epic	2,554	5,128	1,169	3,521	3,788	688	5.51	0.58	0.79	0.11	-0.22
g721decode	1,524	3,060	503	1,869	2,052	423	4.85	0.59	0.80	-0.21	-0.34
g721encode	1,535	3,023	509	1,870	2,046	431	4.75	0.59	0.79	-0.21	-0.35
gsm-decode	8,723	18,298	2,467	9,202	11,218	902	12.44	0.75	0.88	0.14	0.05
gsm-encode	11,425	24,323	2,971	12,309	14,768	1,278	11.56	0.74	0.88	0.03	0.04
pegwit	12,134	32,845	3,119	14,490	15,792	1,461	10.81	0.73	0.90	-0.01	0.10
Average	18,930	37,050	5,644	22,923	25,839	2,033	10.28	0.69	0.85	-0.04	-0.04

Table 1: Results of our encoding strategy on SPEC and MediaBench programs.

The EPIC-like instructions can group dependent operations together if they are marked as so. In other words, the operations are put in groups of 16 even if there exists dependencies among all of them. One EPIC-like instruction has 512 bits to store 16 operations and further 16 bits to prevent dependent operations to be executed in parallel. This is the reason why we multiply the number of EPIC-like instructions by 528 (512 + 16).

Finally, the IA64 instructions has almost the same scheduling algorithm of EPIC-like but its instruction has only 123 bits to store 3 operations. Besides them, the instruction has a set of 5 template bits to prevent dependent operations to be executed at the same time. Thus, an IA64 instruction has 128 (123 + 5) bits.

Table 1 allows for several conclusions. First, the results of the **Reuse** column confirm that there is an intrinsic surjection between encoded instructions and their patterns. For example, a surjection of 10.81 (the pegwit program) means that more than 10 encoded instructions use the same pattern. Another conclusion is about the program size due to the encoding technique. One can observe that the number of encoded instructions (column **Encoded**) is greater than the number of EPIC instructions for all programs. This was expected because, unlike an EPIC-like instruction, a 2D-VLIW encoded instruction takes many architectural constraints into account. As a result, some programs have not obtained size reduction compared to EPIC-like or IA64 strategies (columns RF_e and RF_i). On the other hand, our encoded strategy produces programs up to 78% (255vortex program) smaller than the non-encoded 2D-VLIW, up to 90% (pegwit program) smaller than VLIW-like, up to 37% (255vortex program) smaller than EPIC-like and up to 27% (255vortex program) smaller than IA64 encoding strategy.

6.2 Dynamic Evaluation

The second experiment (dynamic evaluation) compares the impact of using an I-cache plus a P-cache (our approach) with an I-cache (other techniques). This experiment is performed in order to determine if the I-cache plus the P-cache performance for the 2D-VLIW encoding technique can be more efficient than only an I-cache for the other encoding strategies. We use 11 programs of MEDIABench, SPECint and SPECfp benchmarks. In this experiment, we do not compare the 2D-VLIW encoding scheme to IA64 because this is the architecture with the smallest number of functional units. The program traces used in this experiment were obtained by the Trimaran simulator. We consider two parameters to evaluate: the Misses Cost and the Execution Time which are obtained according to Equations 2, 3 and 4. The misses cost for the 2D-VLIW encoding strategy was calculated by the sum of encoded I-cache and P-cache misses costs.

$$MissPenalty = 5 \times \frac{WordLength}{BytesAccess} \tag{2}$$

$$MissesCost = MissPenalty \times NumberOfMisses \tag{3}$$

$$ExecutionTime = MissesCost + NumberOfAccesses \tag{4}$$

$MissPenalty$ represents the cost (in cycles) for one cache miss. This value has been computed according to the parameters defined in [11]. $MissCost$ represents the total cost (in cycles) of all the misses that have occurred in the program. The number of executed instructions is equal to the number of fetched instructions that is exactly the same number of accesses in the cache. Considering one cycle per cache access, the $ExecutionTime$ can be calculated by number of accesses $\times$ 1 cycle + total miss cost.

We have performed all the experiments with cache size ranging from 4KB to 256KB, associativity ranging from direct mapping to 4-way set associative and finally, number of words per block ranging from 1 to 4. For each program, we have done all the combinations of the three parameters: cache size, associativity and number of words per block. For the I-cache, we consider an LRU replacement policy and transfer ratio of 4 bytes per access.

In the P-cache evaluations, we use a replacement policy that protects the most frequent patterns from conflicting with each other. The addresses of each pattern were assigned in inverse order of the pattern frequencies. The most used pattern receives the first address, the second most used pattern receives the second address and so on. This algorithm avoids useful patterns of conflicting with the same cache line since compulsory data are mapped to the same P-cache line. So, the most used patterns have more chances of staying in the P-cache without conflicting with other very used patterns.

After running all the experiments, we have done a careful analysis to define the valid range (considering the size) that must be considered for each program in the comparisons. We have considered as a valid final size, the size where firstly appear only compulsory misses independent of the encoding strategy. For example, if the size where first appears only compulsory misses was 32KB, the reference cache has 32KB. We have chosen the values of I-cache and P-cache sizes for the arrangements that have presented the best value of Execution Time, based on the reference cache. Table 2 show the comparison to the values of the other encoding strategies in an equivalent cache size.

Table 2 shows the results for all the programs in all the encoding strategies evaluated. Column **Ref. Cache** represents the cache size of the other encoding strategies that was compared to our strategy. Columns **I-cache** and **P-cache** represents the I-cache and P-cache sizes, respectively, where we have achieved the best Execution Time. Finally, columns **NonEnc, VLIW, EPIC** and **Encoded** represents the Execution Time, in cycles, for each encoding strategy.

Programs	**Ref. Cache**	**I-cache**	**P-cache**	**NonEnc**	**VLIW**	**EPIC**	**Encoded**
168wupwise	32	2	28	36,021,876,224	75,615,915,061	3,124,872,301	6,694,143,489
175vpr	32	16	14	28,110,106	929,618,529	24,250,617	11,153,580
179art	32	4	28	444,849,216	2,424,943,589	647,329,277	506,379,279
181mcf	8	4	3.5	31,759,605	2,046,507,080	27,097,321	10,206,419
183equake	8	4	3.5	949,286,427	1,703,498,074	29,175,786	280,517,240
197parser	64	8	56	231,771,520	1,279,092,303	405,658,563	272,790,863
256bzip2	32	16	14	7,500,513,024	147,417,936,010	3,366,380,083	2,594,267,721
g721decode	16	2	14	9,010,209,856	17,434,673,337	332,875,762	1,719,471,894
g721encode	16	2	14	9,218,477,968	17,530,826,294	344,411,131	1,841,219,001
gsmdecode	64	32	28	767,161,696	8,797,950,536	463,397,590	269,359,272
pegwit	64	32	28	1,335,193,720	3,941,086,183	100,732,506	57,341,005
Average	-	-	-	5,958,109,942	25,374,731,545	806,016,449	1,296,077,251

Table 2: Execution Time (in cycles) for each program using its best cache size (in KB).

Our encoding strategy provides the best performance (smaller Execution Time) in almost all cases when compared to the 2D-VLIW non-encoded strategy. The Execution Time is up to 23× (pegwit program) and the average is 5× better. Our strategy also provides the best performance in all cases when compared to the VLIW-like strategy. The Execution Time is up to 83× (175vpr program) and the average is 29× better. The main reasons for the gains of our strategy to non-encoded 2D-VLIW and VLIW are the instruction size, and the great miss penalty (80 cycles= $5 \times \frac{64}{4}$) of these strategies compared to the 2D-VLIW encoding technique. Our strategy provides a better performance than the EPIC strategy for the most part of the programs. The Execution Time is up to 2× (181mcf program) and our average execution time, considering the programs where our strategy outperforms EPIC, is 1.2× better. Despite the number of instructions of our technique is much greater than the EPIC-like strategy, our instruction size is 8× smaller and the pattern reuse is very high. Finally, the sum of encoded I-cache plus P-cache miss penalty ($10 + 70 = 80$ cycles) is less than the EPIC (83 cycles) miss penalty.

7 Conclusions

A new technique for encoding long instructions was presented in this paper. This technique decreases the instruction size stored in memory so as to minimize the instruction fetch latency. The technique consists of factoring long instructions of a program into encoded instructions and patterns. The encoded instructions are stored into an I-cache while the patterns are stored into a P-cache. The adoption of a pattern cache and the simplicity of the decoding activity distinguish our strategy to approaches based on code compression.

Results from Section 6 show that our technique can be applied to architectures which fetch long instructions from the memory. Many current architectures like EPIC-based processors, reconfigurable architectures with multiple functional units and high-performance architectures based on a matrix of functional units (or processor elements) can take advantage of this technique.

By using the same cache size, our results show that a program using our encoding strategy is up to 23× and average of 5× faster than a non-encoded scheme. Another interesting result is the program code-size comparison between our strategy and a non-encoded strategy. The existence of the instruction patterns and the large reuse of these patterns make the premises around factorizing patterns and storing them into a P-cache a viable alternative to overcome the bottleneck of fetching long instructions from the memory.

References

[1] G. Araujo, P. Centoducatte, M. Cortes, and R. Pannain. Code Compression Based on Operand Factorization. In *Proceedings of the* 31th *International Symposium on Microarchitecture (MICRO)*, pages 194–201. IEEE Computer Press, 1998.

[2] L. N. Chakrapani, J. Gyllenhaal, W. Mei, W. Hwu, S. A. Mahlke, K. V. Palem, and R. M. Rabbah. Trimaran - An Infrastructure for Research in Instruction-Level Parallelism. *Lecture Notes in Computer Science*, 3602:32–41, 2004.

[3] D. Citron and D. G. Feitelson. Revisiting Instruction Level Reuse. In *Proceedings of the Workshop on Duplication, Deconstructing and Debunking (WDDD)*, pages 62–70, May 2002.

[4] C. Dulong. The IA-64 Architecture at Work. *IEEE Computer*, 31(7):24–32, July 1998.

[5] J. Edler and M. D. Hill. Dinero IV Trace-Driven Uniprocessor Cache Simulator. [online], 1995. http://www.cs.wisc.edu/∼markhill/DineroIV/.

[6] J. Ernst, W. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting. Code Compression. In *Proceedings of PLDI*, pages 358–365. ACM, 1997.

[7] J. A. Fisher. Very Long Instruction Word Architectures and the ELI-512. In *Proceedings of the* 10th *International Symposium on Computerm Architecture (ISCA)*, pages 140–150, Los Alamitos, 1983. Computer Society Press.

[8] M. Franz and K. Thomas. Slim binaries. *Communications of the ACM*, 40(2):87–94, 1997.

[9] J. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millenium. *IEEE Computer*, 33(7):28–35, 2000.

[10] Quinn Jacobson and James E. Smith. Instruction pre-processing in trace processors. In *HPCA*, pages 125–129, 1999.

[11] L. John and R. Radhakrishnan. A selective caching technique. http://citeseer.ist.psu.edu/115017.html.

[12] C. Lee, M. Potkonjak, and W. H. Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of the* 30th *Annual International Symposium on Microarchitecture (MICRO)*, pages 330–335, Research Triangle Park - North Carolina, 1997. IEEE Computer Society.

[13] S. A. McKee. Reflections on the Memory Wall. In *Proceedings of the* 1st *Conference on Computing Frontiers*, pages 162–167. ACM, April 2004.

[14] S. K. Menon and P. Shankar. Space/Time Tradeoffs in Code Compression for the TMS320C62x Processor. Technical Report IISc-CSA-TR-2004-4, Indian Institute of Science, India, 2004.

[15] S-J. Nam, I-C. Park, and C-H. Kyung. Improving Dictionary-Based Code Compression in VLIW Architectures. *IEICE Transactions on Fundamentals*, E82-A(11):2318–2324, November 1999.

[16] J. Prakash, C. Sandeep, P. Shankar, and Y. N. Srikant. Experiments with a New Dictionary Based Code-Compression Tool on a VLIW Processor. Technical Report IISc-CSA-TR-2004-5, Indian Institute of Science, India, 2004.

[17] M. Ros and P. Sutton. Compiler Optimization and Ordering Effects on VLIW Code Compression. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 95–103. ACM, November 2003.

[18] M. Ros and P. Sutton. Code Compression Based on Operand-Factorization for VLIW Processors. In *International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 559–569. ACM, November 2004.

[19] R. Santos, R. Azevedo, and G. Araujo. 2D-VLIW: An Architecture Based on the Geometry of the Computation. In *Proceedings of the* 17th *IEEE International Symposium on Application-Specific, Architectures and Processors(ASAP)*, Steamboat Springs - Colorado, 2006. IEEE Computer Society.

[20] R. Santos, R. Azevedo, and G. Araujo. Exploiting Dynamic Reconfiguration Techniques: The 2D-VLIW Approach. In *Proceedings of the* 13th *IEEE International Reconfigurable Architectures Workshop (RAW)*, Rhodes Island - Greece, 2006. IEEE Computer Society.

[21] Peter G. Sassone and D. Scott Wills. Dynamic strands: Collapsing speculative dependence chains for reducing pipeline communication. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 7–17, Washington, DC, USA, 2004. IEEE Computer Society.

[22] Toshinori Sato and Akihiro Chiyonobu. A preliminary evaluation of timing-speculative instruction collapsing. In *Proc. of 1st Workshop on Introspective Architectures*, 2006.

[23] A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the Memory Wall: The Case for Processor/Memory Integration. In *Proceedings of the* 23th *International Symposium on Computerm Architecture (ISCA)*, pages 90–101, Philadelphia, October 1996. ACM.

[24] M. S. Schlansker and B. R. Rau. EPIC: Explicitly Parallel Instruction Computing. *IEEE Computer*, 33(2):37–45, February 2000.

[25] A. Sodani and G. S. Sohi. Dynamic Instruction Reuse. In *Proceedings of the* 24th *International Symposium on Computerm Architecture (ISCA)*, pages 194–205. ACM, 1997.

[26] Y. Xie, W. Wolf, and H. Lekatsas. A Code Decompression Architecture for VLIW Processors. In *Proceedings of the* 34th *IEEE/ACM International Symposium on Microarchitecture*, pages 66–75. IEEE Computer Press, 2001.

[27] Y. Xie, W. Wolf, and H. Lekatsas. Compression Ratio and Decompression Overhead Tradeoffs in Code Compression for VLIW Architectures. In *Proceedings of the* 4th *International Conference on ASIC*, pages 337–341, October 2001.

[28] Y. Xie, W. Wolf, and H. Lekatsas. Code Compression for VLIW Processors Using Variable-to-Fixed Coding. In *Proceedings of the* 15th *IEEE/ACM International Symposium on System Synthesis*, pages 138–143. IEEE Computer Press, 2002.