# Harnessing Horizontal Parallelism and Vertical Instruction Packing of Programs to Improve System Overall Efficiency*

Hai Lin and Yunsi Fei
Dept. of Electrical & Computer Engineering
University of Connecticut
Storrs, CT 06269
E-mail: {hal06002,yfei}@engr.uconn.edu

## Abstract

*Multi-issue processors can exploit the Instruction Level Parallelism (ILP) of programs to improve the performance greatly. How to reduce the energy consumption while maintaining the high performance of programs running on multi-issue processors remains a challenging problem. In this paper, we propose a novel approach to apply the instruction register file (IRF) technique from single-issue processor to VLIW architecture. Frequently executed instructions are selected to be placed in the on-chip IRF for fast access in program execution. Violation of synchronization among VLIW instruction slots is avoided by introducing new instruction formats and microarchitectural support. The enhanced VLIW architecture is thus able to orchestrate the horizontal instruction parallelism and vertical instruction packing for programs to improve system overall efficiency. Our experimental results show that the proposed processor architecture achieves both the performance advantage provided by the VLIW architecture and high energy efficiency provided by the IRF-based instruction packing technique (e.g., 71.1% reduction in the fetch energy consumption for a 4-way VLIW architecture with 8-entry IRFs).*

## 1 Introduction

Microprocessor designs, whether for general purpose or embedded systems, are continuously pushing for optimization of performance, power consumption, and cost. Various hardware and software design technologies often target one or more design goals at the expense of the others. Horizontal parallelism, i.e., instruction level parallelism (ILP), has been exploited in both very-long-instruction-word (VLIW) and superscalar processors for performance improvement, reducing the pressure on system clock frequency increase. Superscalar architectures rely on complex instruction decoding and dispatching hardware for run-time data dependency detection and parallel instruction identification [9]. VLIW technology, however, groups parallel instructions in a long word format, and reduces the hardware complexity by maintaining simple pipeline architectures and allowing compilers to control the scheduling of independent operations. Hence, it has large flexibility to optimize the code sequence and exploit the maximum ILP [12]. This feature of VLIW architecture makes it a good candidate for high performance embedded system implementation. Currently, the research on VLIW mainly focuses on compilation algorithms and hardware enhancement that can fully utilize the ILP and reduce waste of instruction slots, improving the performance and reducing the program memory space, cache space, and bus bandwidth [5] [13]. However, the performance improvement is usually achieved at the cost of power consumption, and techniques for both power consumption reduction and performance improvement are not fully explored. In this paper, we propose to employ the vertical instruction packing technique in VLIW architectures to greatly reduce the instruction fetch power consumption, which occupies a large portion of the overall power consumption of VLIW processors.

Our approach is based on the concept of on-chip instruction register file (IRF) [7], where frequently occurring instructions are placed in the IRF and multiple entries in the IRF can be referenced by a single packed instruction in ROM or L1 instruction cache. The principle of "fetch-one-and-execute-multiple" (through vertical instruction packing and decoding) can greatly reduce power consumption, decrease program code size, reduce cache misses, and improve performance further. By applying architectural changes and instruction set architecture (ISA) and program modifications, our approach manages to bring the advantages of the IRF technique to the VLIW domain, i.e., harnessing both horizontal parallelism and vertical instruction packing of programs for system overall efficiency improvement.

The rest of the paper is organized as follows. First, we review the related work in Section 2. Then we provide the motivation for our approach and the detailed problem analysis in Section 3. We discuss the software enhancement in Section 4, including the ISA and program modifications for adapting the IRF technologies to VLIW architecture. In Section 5, we outline the hardware modifications necessary for supporting the IRF and new ISA. Section 6 presents our experimental results. Finally, conclusions are drawn in Section 7.

## 2 Related Work

Both performance and energy consumption are important to modern embedded processors. There has been some research work that focuses on balancing energy consumption and performance trade-offs for embedded multi-issue processors [3] [10]. Various approaches have been taken to reduce power consumption of hot spots in processors. In [14] [15], the idea of instruction grouping has been employed to reduce the energy consumption of superscalar processors for storing instructions in the instruction queue and selecting and waking up instructions at the instruction issue stage. However, these techniques require on-line instruction grouping algorithms and result in complex hardware implementation for run-time group detection. They are not flexible in instruction packing, with limited grouping patterns. Moreover, they lack the ability to physically pack instructions to reduce the hardware cost, program code size,

*Acknowledgments: This work was supported by a NSF grant CCF-0541102.

978-3-9810801-3-1/DATE08 © 2008 EDAA

and energy consumption in memory. In [11], the authors managed to reduce the program code size and the memory access energy cost in VLIW architectures by applying instruction compression/decompression between memory and cache. However, it also requires complex compression algorithms and hardware implementation, and the power consumption of the processor has not been effectively reduced.

The IRF technique has been introduced in [7, 8], aiming to design a counterpart of data register file for instructions. Based on profiling information, frequently occurring instructions are placed in the on-chip IRF and multiple entries in the IRF can be referenced by a single packed memory instruction. Both the number of instruction fetches and the program memory energy consumption are greatly reduced. With position registers and a table storing frequently used immediates, this technique applies successfully to single-issue processors. However, the performance improvement is trivial.

Observing the limitations of multi-issue processors in power consumption reduction and the advantages of IRF techniques, we hypothesize that applying the IRF technique to multi-issue architectures will both reduce the energy consumption and exploit the ILP for performance improvement simultaneously. We choose the VLIW architecture for its simple hardware implementation and regular instruction coding format. Both the hardware and software need to be modified to adapt the IRF technique to VLIW architecture so that the advantages from both sides can be maintained.

## 3 Motivation and Problem Analysis

In single-issue processors with an IRF, the most frequently referenced instructions of an application are loaded into the on-chip IRF. Corresponding sub-streams of instructions in the program are grouped and replaced by single packed instructions, i.e., memory ISA (MISA) instructions. A compact MISA instruction contains several indexes in one instruction word for referencing multiple entries in the IRF, as shown in Figure 1. The indexes are used in the first half of the decode stage of the pipeline to refer to the real instructions in the IRF (denoted as register ISA - RISA). Figure 2 illustrates the enhanced pipeline stages with an IRF. By integrating an IRF in the architecture and allowing arbitrary combinations of RISAs in an MISA, not only the program code size is decreased, but also the number of instruction fetches and thus fetching energy consumption is reduced greatly. There are two ways of integrating an IRF

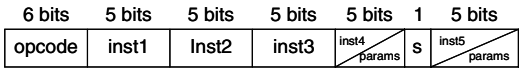


**Figure 1. Packed instruction format [7]**

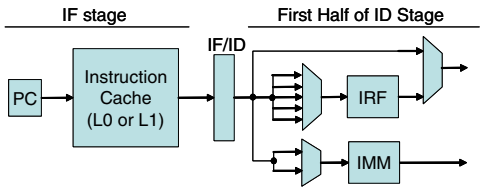


**Figure 2. Instruction register file within single-issue processor pipeline stages [7]**

in VLIW architectures. The first is a naive one by utilizing the horizontal parallelism and vertical packing in an orthogonal manner, i.e., VLIW compilation followed by IRF insertion. The RISA instructions put into the IRF are long-word instructions, and the size of each IRF entry is scaled accordingly. Program profiling for obtaining instruction frequency information and selecting RISA instructions is based on the long-word instructions. In this way, although the complexity of hardware and compiler modifications for supporting the IRF is the same as in single-issue architectures, it loses much flexibility of instruction packing. Different combinations of the same sub-instructions would be considered as different long instruction candidates, thus reducing the efficiency of IRF usage greatly.

The second way is to couple the horizontal parallelism and vertical packing in a cooperative manner, i.e., VLIW compilation and IRF insertion are integrated. An IRF is built to store the most frequently executed sub-instructions, and the size of each entry is the same as that for single-issue processors. The instruction packing is along the instruction slots. This approach allows higher flexibility in packing the most efficient RISAs for each instruction slot. Thus, the IRF resource is better utilized. Figure 3 illustrates an example. At the profiling phase, there are three long instructions executed in a sequence, each with an execution frequency of 1. If we have an IRF size of four sub-instructions, in the first way, there is only one-entry in the IRF and one long instruction can be referenced. In the second way, each long instructions is broken down to sub-instructions, we choose 4 most frequently executed sub-instructions and put them into the IRF, e.g., $I_1$, $I_2$, $I_4$, and $I_5$ in Figure 3. A total number of 9 sub-instructions are referenced from the IRF instead of the cache. Thus, the second way can potentially save code size and cache access times. We can either build a global IRF with multiple ports across the slots, or dedicate an individual IRF to each slot. A global IRF is more capable in exploiting the execution frequency of sub-instructions among the slots when the VLIW pipes are homogeneous. However, separate IRFs are suitable when each instruction slot corresponds to certain execution units in the datapath and is dedicated to a subset of the ISA. In our design, we adopt the separate IRFs for different slots, as the pipes are heterogeneous in typical VLIW architectures.

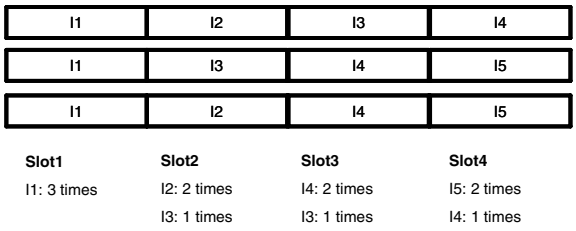


**Figure 3. Analyzing execution frequency of sub-instructions in long-word instructions**

However, it is not feasible to directly pack sub-instructions of each instruction slot in VLIW architectures and maintain the horizontal parallelism among the multi-way execution units. The original VLIW compiler schedules the instruction sequence. With an IRF inserted, the sub-instructions are packed for each slot. At an execution cycle, those instruction slots that receive such compact instructions refer to multiple RISAs in the IRF, and thus it takes multiple cycles to finish execution. Since the number of sub-instructions may vary among different slots, the original synchronized behavior of the slots may be destroyed and the parallelism between the independent operations cannot be guaranteed. Figure 4 demonstrates the code sequence of an example program for an original 2-way VLIW architec-

ture. Each slot contains its own sub-instruction and they work in a synchronized manner. Assuming that the sub-instructions included in the ovals (shown in Figure 4) can be packed, the directly grouped MISA instruction sequence is shown in Figure 5. The first instruction word contains two packed sub-instructions, one refers to three RISA instructions (I1, I2, and I3) and the other refers to two (I1' and I2'). Only when both of the two slots have finished execution, the subsequent instructions can be executed. Thus, the first VLIW instruction word takes three cycles to execute, with the second slot idling in the third cycle. When the second instruction word is fetched and executed, one slot is executing two sub-instructions in a sequence (I4 and I5), and the other only has one (I3'). If there is a data dependency of I4 on I3', this VLIW instruction has internal RAW (read-after-write) data hazard and may cause malfunctioning. Although the code size and the total number of instruction fetches are reduced, the behavior of the execution units is unsynchronized and may cause extra pipeline stalls. To address this problem, we propose a novel approach through ISA and program modifications and architecture enhancements to regain synchronization among all the slots. Therefore, both the performance advantage of VLIW architecture and the code size and energy consumption reduction by applying the IRF instruction packing technique can be achieved.

It is noteworthy that our code size reduction mechanism

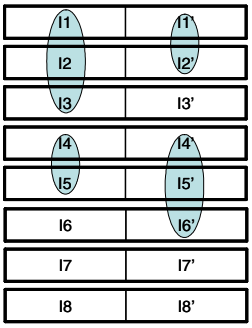


**Figure 4. Original VLIW code sequence**

| # of RISAs in slot 1 | Slot 1 | Slot 2 | # of RISAs in slot 2 |
|---|---|---|---|
| 3 | I1-I2-I3 | I1'-I2' | 2 |
| 2 | I4-I5 | I3' | 1 |
| 1 | I6 | I4'-I5'-I6' | 3 |
| 1 | I7 | I7' | 1 |
| 1 | I8 | I8' | 1 |

**Figure 5. The directly packed instruction sequence**

through IRF insertion is orthogonal to the traditional VLIW code compression algorithms. VLIW compiler statically schedules sub-instructions to exploit the maximum ILP, and NOP instructions may be inserted in some instruction slots if the ILP is not wide enough. Since these NOP instructions introduce large code redundancy, state-of-the-art VLIW implementations usually apply code compression techniques to eliminate NOPs to reduce the code size in memory [4]. Extra bits, such as $head$ and $tail$, are inserted to the variable length instruction words to annotate the beginning and end of the long instructions in memory [4]. A decompression logic is needed to retrieve the original fixed-length instruction words before they are fetched to processor. In contrast, our instruction packing algorithm is along the vertical dimension, and no sub-instructions are eliminated in the long instruction word. The code is compressed in a way that one MISA instruction contains indexes for referring to multiple RISAs in the on-chip IRF. It takes place before the traditional code compression mechanisms, and thus transparent to them. To make a fair comparison, we will compare the program code size with our approach against that after traditional VLIW code compression techniques.

# 4 Software Modification

This section describes the software changes necessary for an ISA to support incorporating and referencing to an IRF in VLIW architectures.

## 4.1 ISA Extensions

Simple slot-based sub-instruction packing would introduce violations of original ILP, as illustrated in Figure 5. To eliminate these violations, an enhanced ISA is proposed in our approach. All the IRF-related instructions are classified into four categories spanning two hierarchy levels, as shown in Figure 6. The first two instruction formats are at the lower hierarchy level that targets the instruction slots in a long VLIW instruction. They are similar to those instructions for single-issue architectures, as listed below.

- RISA instruction represents the primitive sub-instructions put in the IRF, i.e., basic operations such as $add\_i$.

- MISA instruction is defined as the sub-instruction that can occupy one VLIW instruction slot. An MISA sub-instruction can be a regular single sub-instruction, or in a compact style, referring to number of m RISA instructions, where m is between 2 and n (n is determined by the instruction word length and the IRF size, e.g., n=5 for 32-bit long instructions and an IRF of 32 entries for single-issue MIPS architectures). In [7], the authors have discussed in detail about how to pack and decode these RISA instructions. We apply the similar instruction packing method in our approach.

The other two instruction formats listed below are at the upper hierarchy level that targets the whole VLIW instruction word stored in memory. They consist of multiple MISA sub-instructions.

- PISA instruction is the regular parallel long-word instruction. The MISAs that it contains in different instruction slots are dispatched to corresponding execution units (we simply call pipes) simultaneously at run-time. This kind of instruction is referred to as parallel ISA - PISA.

- SISA instruction is a special kind of long-word instruction that we propose to compensate the mismatch of sub-instruction sequences among slots caused by the IRF-based instruction packing technique. At run-time, all the MISA sub-instructions contained in this kind of instructions are dispatched to one pipe in a sequential order. This type of instruction is referred to as sequential ISA - SISA. Several reserved bits in the instruction word are encoded to indicate the instruction type and its target pipe (for the SISA type).

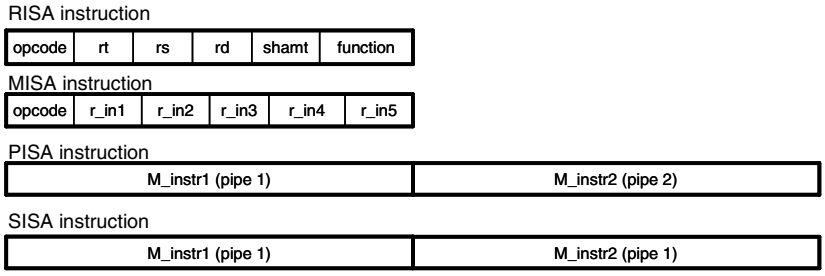


**Figure 6. Four categories of instructions related to VLIW architectures with an IRF**

## 4.2 Program Modifications

With these aforementioned four types of instructions in an ISA, we expect to realize slot-based sub-instruction packing while maintaining the parallelism of these sub-instructions. This section provides an overview of program recompilation and code rescheduling for the new VLIW architecture. We revisit the example used in Section 2. For the original code sequence in Figure 4, at compile-time, the sub-instructions in each slot are packed to MISA-style based on the IRF contents (RISAs). Then the MISA sub-instructions are reorganized within each long word, and necessary SISA instructions are inserted to reconcile the pace mismatch among the pipes caused by their different number of RISAs. In our approach, whenever there is a mismatch, an SISA instruction is inserted to occupy the pipe that has the least number of RISA sub-instructions. For example, Figure 7 shows the reorganized code sequence, where SISAn represents the sequential long instruction word for pipe $n$. Since there is a mismatch in the number of RISAs between the two slots of the first PISA instruction word (v1), i.e, 3 for the first slot and 2 for the second slot, the whole second instruction word (v2) will go to the shorter pipe to compensate the pace (SISA2). The third instruction word (v3) will go to pipe 1, i.e., SISA1, because it is shorter after the first two instructions. At the end of instruction v3, the two pipes are re-synchronized, and the following instruction (v4) will be a PISA with parallel sub-instructions.

Figure 8 depicts the detailed cycle-accurate behavior of the two pipes, assuming all the slots in an instruction word share the same fetch stage but each has its own decode stage, and ignoring the non-ideal execution cases like multi-cycle execution, instruction/data cache miss, etc. We can see the cycle time when instruction fetches occur, e.g., v1 is fetched in cycle 1, v2 in 3, v3 in 4, etc. The italicized fetch behavior (e.g., $F_{V2}$ in pipe 1) indicates that there is an instruction fetch occurring in that cycle but no MISA instruction is dispatched to the specific pipe for execution, i.e., it is a SISA instruction for other pipes. The total execution time for the instruction sequence is 12 cycles, the same as that for the original VLIW architecture without IRF. However, the number of instruction fetches is 5, as compared to 8 for the original architecture.

# 5 Hardware Enhancements

The hardware architecture has to be enhanced accordingly to support the new features of the ISA. The modified architecture for a two-way VLIW processor is shown in Figure 9. All the modifications are made in the first half of decoding stage, and they are almost identical for different pipes. The IRF reference logic is the same as in single-issue processors [7] and is simply duplicated for multi-way VLIW architecture. It interprets the incoming MISA instruction, and issues either a single sub-instruction or refers

| Instruction sequence | # of RISAs in slot 1 | Slot 1 | Slot 2 | # of RISAs in slot 2 |
|---|---|---|---|---|
| V1: | 3 | I1-I2-I3 | I1'-I2' | 2 |
| V2 (SISA2): | 0 | I3' | I4'-I5'-I6' | 4 |
| V3 (SISA1): | 3 | I4-I5 | I6 | 0 |
| V4: | 1 | I7 | I7' | 1 |
| V5: | 1 | I8 | I8' | 1 |

**Figure 7. The re-organized and re-scheduled instruction sequence for an VLIW architecture with IRF**

to multiple RISA instructions in the IRF and issues them sequentially to the targeted pipe.

Figure 9 also shows that extra PISA/SISA decode logic is inserted before the IRF reference module, to interpret the incoming PISA or SISA instructions and control the instruction flow in each pipe. When the incoming instruction is a regular VLIW PISA instruction, signals are generated by the *ISA type/pipe detection logic* (put in the IF stage, ignored in the figure) for multiplexers $MUX_{1,1}$ and $MUX_{1,2}$ to select and pass M_instr1 to the IRF reference logic. Similarly for pipe 2, M_instr2 is selected by $MUX_{2,1}$ and $MUX_{2,2}$. When a SISA instruction is incoming, e.g., SISA2 for pipe 2, $MUX_{2,1}$ selects M_instr1 and the tri-state gate $T_2$ is enabled to buffer M_instr2 for future execution. The control signal for $MUX_{2,2}$ is generated to feed M_instr1 and M_instr2 sequentially to the IRF reference module. In the other pipe (pipe 1), however, none of the new sub-instructions will be selected and the pipe is just continuing its pre-scheduled operations from previous instructions. The extra PISA/SISA decode logic ensures that different types of ISA instructions are identified and the MISA sub-instructions are dispatched to the right pipes. Note that to successfully fetch SISA instructions to compensate the vertical execution length mismatch, a new instruction should be fetched as long as one of the pipe has finished all its sub-instructions. This can be implemented by a fetch enable logic generator in the IF stage. A status signal is generated for each pipe when the pipe is empty. An OR logic is used to take in the two pipe's status signals and output a fetch control signal for the instruction cache in IF stage.

There are several non-ideal execution cases, such as multi-cycle instruction execution, instruction cache miss, and data cache miss, which need to be handled by the enhanced VLIW architecture. On an instruction or data cache miss, all the pipes will be stalled, just in the same way as the original VLIW architecture. In addition, the buffers used in the IRF reference logic stop issuing RISA instructions

| Cycle / MISA Instr. | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I1-I2-I3 | $F_{V1}$ | $D_1$ | $E_1$ | $M_1$ | $W_1$ | | | | | | | |
| | | | $D_2$ | $E_2$ | $M_2$ | $W_2$ | | | | | | |
| | | | $F_{V2}$ | $D_3$ | $E_3$ | $M_3$ | $W_3$ | | | | | |
| I4-I5-I6 | | | | $F_{V3}$ | $D_4$ | $E_4$ | $M_4$ | $W_4$ | | | | |
| | | | | | | $D_5$ | $E_5$ | $M_5$ | $W_5$ | | | |
| | | | | | | | $D_6$ | $E_6$ | $M_6$ | $W_6$ | | |
| I7 | | | | | | | $F_{V4}$ | $D_7$ | $E_7$ | $M_7$ | $W_7$ | |
| I8 | | | | | | | | $F_{V5}$ | $D_8$ | $E_8$ | $M_8$ | $W_8$ |

(a) Pipe 1

| Cycle / MISA Instr. | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I1'-I2' | $F_{V1}$ | $D_{1'}$ | $E_{1'}$ | $M_{1'}$ | $W_{1'}$ | | | | | | | |
| | | | $D_{2'}$ | $E_{2'}$ | $M_{2'}$ | $W_{2'}$ | | | | | | |
| I3'-I4'-I5'-I6' | | | $F_{V2}$ | $D_{3'}$ | $E_{3'}$ | $M_{3'}$ | $W_{3'}$ | | | | | |
| | | | | $F_{V3}$ | $D_{4'}$ | $E_{4'}$ | $M_{4'}$ | $W_{4'}$ | | | | |
| | | | | | | $D_{5'}$ | $E_{5'}$ | $M_{5'}$ | $W_{5'}$ | | | |
| | | | | | | | $D_{6'}$ | $E_{6'}$ | $M_{6'}$ | $W_{6'}$ | | |
| I7' | | | | | | | $F_{V4}$ | $D_{7'}$ | $E_{7'}$ | $M_{7'}$ | $W_{7'}$ | |
| I8' | | | | | | | | $F_{V5}$ | $D_{8'}$ | $E_{8'}$ | $M_{8'}$ | $W_{8'}$ |

(b) Pipe 2

**Figure 8. The cycle-accurate behavior of the two pipes in a two-way VLIW architecture with IRF**

to avoid dynamic execution hazards. For multi-cycle execution, since it occurs in the pipeline stage later than the decode stage, where our instruction packing and IRF referencing mechanism take place, the handling mechanisms are transparent to our packing methods. For example, the stalls caused by multi-cycle execution can be implemented by NOP insertion at compile-time [6]. At runtime, the sub-instructions of each slot are recovered to the original execution sequence after IRF referencing. Thus, the multi-cycle handling mechanism for the original VLIW architecture applies here.

## 6 Experimental Results

To evaluate our enhanced VLIW architecture, we use Trimaran [2], an integrated compilation and performance simulating environment released by HP Labs, as the infrastructure for our experimental setup. Table 1 lists the architecture configuration for a 4-way VLIW [1]. The original VLIW program code is generated by the compiler, and the modified simulator is used to profile the program for runtime information. The profiling data is captured by our analysis tool, *irfgen*, to select the best candidate instructions for the IRF. Then the program code is modified and reorganized to use the newly developed instructions, including MISA, PISA, and SISA. Currently, we restrict the instruction packing within hyper-blocks of VLIW code and do not include branch instructions. We finally simulate the modified code and obtain the execution statistics.

A set of benchmarks have been tested to evaluate the effectiveness of our approach in code size reduction and energy saving. These benchmarks are provided by Trimaran and represent typical embedded applications for VLIW architectures, such as system commands ($strcpy$ and $wc$),

[1]Note that the previous analysis is based on 2-way VLIW architectures just for the sake of simplicity.

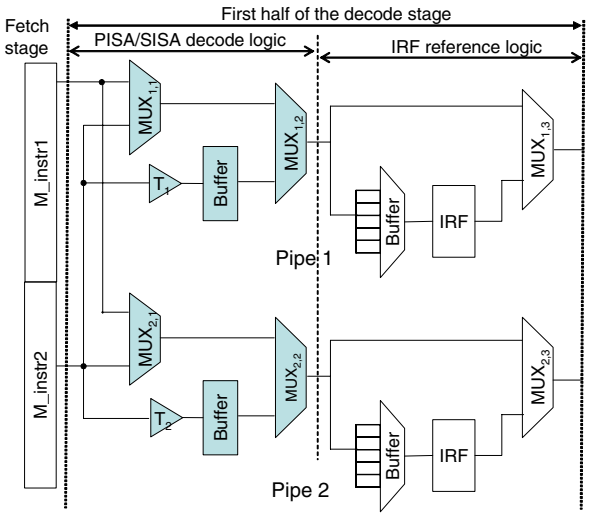


**Figure 9. The enhanced pipeline stage for the VLIW architecture with IRF**

**Table 1. VLIW processor configuration**

| | |
|---|---|
| **Number of slots** | **4** |
| **Integer Units** | **4** |
| **Floating Units** | **2** |
| **Memory Units** | **2** |
| **Branch Units** | **1** |

matrix operations ($bmm$ and $mm\_double$), arithmetic functions ($hyper$ and $eight$) and other special test programs ($wave$ and $test\_install$). Results show that the program memory size is reduced through instruction packing. To make a fair comparison, we compare our program code size with that under the traditional VLIW code compression, where all the NOPs are removed. Figure 10 demonstrates that over eight benchmarks the average reduction rate of the static code size is 14.9% for VLIW processors with 4-entry IRFs, and 20.8% for 8-entry IRFs.

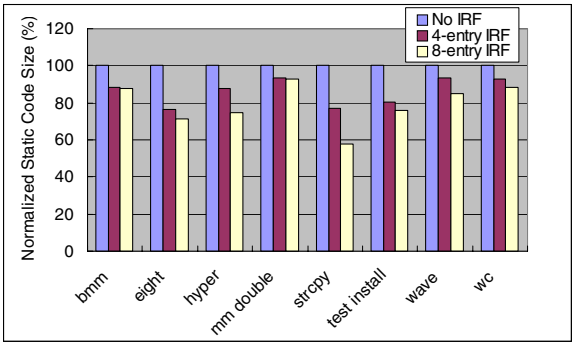


**Figure 10. Static code size reduction for 4-way VLIW with different IRF sizes**

Table 2 shows the instruction fetch numbers under different IRF implementations. The fetch number is reduced greatly for a 4-way enhanced VLIW processor. The average reduction rate over the eight benchmark applications is 65.5% for 4-entry IRFs and 71.8% for 8-entry IRFs. Note that the reduction rate for a 4-way VLIW processors with 4-entry IRFs is larger than that for a single-issue processor with a 16-entry IRF [7] [8], due to the advantage of select-

ing sub-instructions of different slots separately for IRFs in our approach.

**Table 2. Fetch number reduction rate**

| Bench-marks | Total fetch number | | | Reduction rate(%) | |
|---|---|---|---|---|---|
| | No IRF | 4-entry | 8-entry | 4-entry | 8-entry |
| bmm | 74657 | 24721 | 18181 | 66.9 | 75.6 |
| eight | 3488 | 1343 | 1177 | 61.5 | 66.2 |
| hyper | 2477 | 789 | 693 | 68.1 | 72.0 |
| mm_d | 67645 | 22166 | 17416 | 67.2 | 74.3 |
| strcpy | 13391 | 4534 | 3765 | 66.1 | 71.8 |
| test instl | 1050070 | 325030 | 275030 | 69.0 | 73.8 |
| wave | 14665 | 5540 | 4385 | 62.2 | 70.1 |
| wc | 1481025 | 550701 | 434164 | 62.8 | 70.7 |
| Average | | | | 65.5 | 71.8 |

Previous research [7] has shown that the instruction fetch energy can reach up to 30% of the total energy consumption for current embedded processors. The large reduction in the total fetch number can save a lot of instruction fetch energy, and thus reduce the total energy consumption significantly. We adopt a simple energy estimation model for the fetching energy consumed by both instruction cache access and IRF referencing based on previous work [1], as shown below.

$$E_{fetch} = 100 * Num_{L1\_access} + Num_{IRF\_access} \quad (1)$$

In the model, the energy cost for accessing L1 cache is 100 times of that for IRF due to the tagging and addressing logic. For simplicity, we assume that all the VLIW instructions fetches hit in the L1 instruction cache, and ignore the extra cache miss energy consumption. In reality, with smaller code size and fewer cache misses, the energy reduction of our approach would be larger. Figure 11 demonstrates the fetch energy reduction for a 4-way VLIW architecture with the IRF size varying between 4 and 8. The average reduction rate of the fetch energy consumption for VLIW architectures with 4-entry IRFs is 64.8%, and 71.1% for 8-entry.

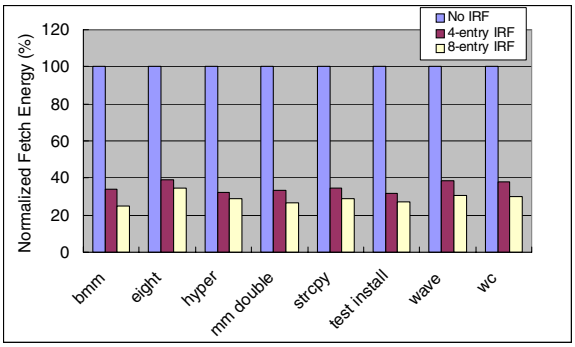


**Figure 11. Fetch energy reduction for 4-entry and 8-entry IRF implementations**

As our approach recovers the original VLIW sub-instruction sequence for execution at run-time, the multi-issue VLIW instruction execution can be preserved without any performance degradation. Our design adds simple PISA/SISA decoding in the decode stage, which may introduce small delay and negligible energy overhead in the decoding cycle. However, since normally the critical path of pipeline is in the execution stage, we can assume that the clock cycle time is not increased by the extra decoding logic. If for some architecture it is not the case, the PISA/SISA decoding logic can be moved to the end of fetch stage to shorten the critical path of decode stage.

In our experiments, we set the maximum number of RISAs in an MISA instruction as 5, which is used in [7] for an IRF with 32 entries and the instruction word length of 32 bits. In our experiments, when IRF entry number is reduced to 4 or 8, the index bit-length changes to 2 or 3, and more IRF instructions can be referred by one MISA instruction. This should lead to even larger static code size reduction and higher fetch energy saving.

## 7 Conclusions

VLIW architecture can exploit the horizontal parallelism of programs, ILP, to improve the performance greatly, and has the advantage of simpler hardware implementation than superscalar architectures.

In this paper, we incorporate IRFs to VLIW architecture to utilize the execution characteristics of programs for reducing the energy consumption. The ISA is modified to orchestrate the instruction grouping along the two dimensions, and both the program and microarchitecture are modified correspondingly. The experimental results show that our approach can reduce program code size by 20.8% and save the instruction fetch energy consumption by 71.1% for 4-way VLIW architecture with 8-entry IRFs without any performance degradation. It offers an effective technique for improving the overall efficiency of embedded processors.

## References


[1] SIMPLESCALAR-ARM POWER MODELING PROJECT. [http://www.eecs.umich.edu/ panalyzer/].

[2] Trimaran. [http://www.trimaran.org/].

[3] G. Ascia, V. Catania, M. Palesi, and D. Patti. System-level framework for evaluating area/performance/power trade-offs of VLIW-based embedded systems. In *Proc. Asia & South-Pacific Design Automation Conf.*, pages 940–943, Jan. 2005.

[4] 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 *Proc. Int. Symp. Microarchitecture*, pages 201–211, Dec. 1996.

[5] E. Gibert, J. Sanchez, and A. Gonzalez. Effective instruction scheduling techniques for an interleaved cache clustered VLIW processor. In *Proc. Int. Symp. Microarchitecture*, pages 123–133, Nov. 2002.

[6] S. Haga, Y. Zhang, A. Webber, and R. Barua. Reducing code size in VLIW instruction scheduling. *Journal of Embedded Computing*, 1(3):415–433, Aug. 2005.

[7] S. Hines, J. Green, G. Tyson, and D. Whalley. Improving program efficiency by packing instructions into registers. In *Proc. Int. Symp. Computer Architecture*, pages 260–271, May 2005.

[8] S. Hines, G. Tyson, and D. Whalley. Improving the energy and execution efficiency of a small instruction cache by using an instruction register file. In *Proc. of Watson Conf. on Interaction between Architecture, Circuits, & Compilers*, pages 160–169, Sept. 2005.

[9] M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, 1991.

[10] H. S. Kim, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. A framework for energy estimation of VLIW architecture. In *Proc. Int. Conf. Computer Design*, pages 40–46, Sept. 2001.

[11] A. Macii, E. Macii, F. Crudo, and R. Zafalon. A new algorithm for energy-driven data compression in VLIW embedded processors. In *Proc. Design Automation & Test Europe Conf.*, pages 10024–10030, Oct. 2003.

[12] Philips-Inc. *An Introduction to Very-long Instruction Word (VLIW) computer architecture*. Philips Semiconductors, 1997.

[13] Y. Qian, S. Carr, and P. Sweany. Optimizing loop performance for clustered VLIW architectures. In *Proc. of Int. Conf. on Parallel Architectures & Compilation Techniques*, pages 271–280, Sept. 2002.

[14] H. Sasaki, M. Kondo, and H. Nakamura. Energy-efficient dynamic instruction scheduling logic through instruction grouping. In *Proc. Int. Symp. Low Power Electronics & Design*, pages 43–48, Oct. 2006.

[15] J. Sharkey, D. Ponomarev, K. Ghose, and O. Ergin. Instruction packing: reducing power and delay of the dynamic scheduling logic. In *Proc. Int. Symp. Low Power Electronics & Design*, pages 30–35, Aug. 2005.