Run-time Phase Prediction for a Reconfigurable VLIW Processor

Qi Guo*, Anderson Sartor†, Anthony Brandon‡, Antonio C. S. Beck¶, Xuehai Zhou**, Stephan Wong§

Univ. of Science and Tech. of China
{*gqustc@mail., **xhzhou@}ustc.edu.cn

Univ. Federal do Rio Grande do Sul
{†alsartor, ¶caco}@inf.ufrgs.br

Delft University of Technology
{‡a.a.c.brandon, §J.S.S.M.Wong}@tudelft.nl

***Abstract*—It is well-known that different applications exhibit varying amounts of ILP. Execution of these applications on the same fixed-width VLIW processor will result (1) in wasted energy due to underutilized resources if the issue-width of the processor is larger than the inherent ILP; or alternatively, (2) in lower performance if the issue-width is smaller than the inherent ILP. Moreover, even within a single application distinct phases can be observed with varying ILP and therefore changing resource requirements. With this in mind, we designed the ρ-VEX processor, which is a VLIW processor that can change its issue-width at run-time. In this paper, we propose a novel scheme to dynamically (i.e., at run-time) optimize the resource utilization by predicting and matching the number of active data-paths for each application phase. The purpose is to achieve low energy consumption for applications with low ILP, and high performance for applications with high ILP, on a *single* VLIW processor design. We prototyped the ρ-VEX processor on an FPGA and obtained the dynamic traces of applications running on top of a Linux port. Our results show that it is possible in some cases to achieve the performance of an 8-issue core with 10% lower energy consumption, while in others we achieve the energy consumption of a 2-issue core with close to 20% lower execution time.**

I. Introduction

The ρ-VEX processor [1] is a recently introduced VLIW processor design that is capable of dynamically reconfiguring certain aspects of its organization. In this way, traditional drawbacks of (fixed) VLIW processors, e.g., high NOP count and resource under-utilization, can be overcome while maintaining simplicity and high performance. The main reconfiguration parameter of the ρ-VEX processor is the issue-width, which can be changed at run-time between 8, 4, or 2 [2]. Combined with the use of generic binaries [3], only a single binary needs to be generated to run on the different issue-width core instances.

This capability of the ρ-VEX processor allows it to adapt itself to only use the necessary resources that are needed by the applications, i.e., matching the inherent ILP of the application or even phases within applications. The freed resources can be switched off to save energy (focus of this paper) or be used to run additional applications, basically turning the ρ-VEX design into a multicore computing platform. For example, Fig. 1(a) depicts the different phases (based on ILP) of the $jpeg$ application. When a fixed 4-issue VLIW processor is used, much of its resources will be wasted at the beginning of the application leading to increased energy consumption. If a fixed 2-issue VLIW processor is used, execution of the latter phase of the application will lead to longer execution time.

When executing applications, we consider two important metrics, namely delay and energy (consumption). Considering the same application, Fig. 1(b) and Fig. 1(c) depict the delay and energy measurements when executing it on a fixed 2-, 4-, and 8-issue VLIW processor. The 8-issue core is used as the reference in both graphs. We can clearly observe that depending on whether the priority is on performance or (reduced) energy consumption, different phases prefer a different optimal issue-width core. In this paper, we propose three strategies to decide at run-time which configuration is best depending on where the priority is placed on: delay, energy, or energy delay product (EDP). For this purpose, the execution time is divided into intervals (during execution) and using certain metrics, a prediction is made what the best configuration (8-, 4-, or 2-issue) is for the subsequent interval. In order to avoid unstable predictions, i.e., each execution interval requiring a change compared to its predecessor, a smoothing algorithm is introduced with a certain threshold that must be reached before a change is actuated.

Clearly the usefulness of this approach depends on phases of different ILP being present in the first place. As presented in Table I, our phase detection scheme is able to differentiate between 50 to 50000 phases of different issue-widths depending on the application. These applications are taken from the Mibench [4] embedded benchmark suite. This indicates that there should be abundant opportunity for our approach to optimize energy consumption and performance.

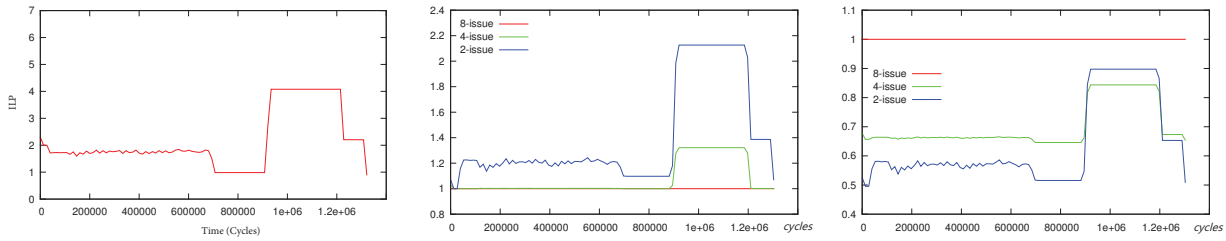
In this paper we present the following contributions:

- We proposed three algorithms and associated hardware modules to gather the execution information at run-time.
- We proposed three strategies to perform the trade-off between energy and delay. And, we designed an algorithm to predict the application phases.
- We compared the results with the fixed VLIW cores in terms of energy and delay.

II. Related Work and Motivations

Reconfigurable processors [5] have emerged to the choice for application execution when flexibility and adequate per-

978-3-9815370-7-9/DATE16/



(a) ILP-Phases (b) Delay (c) Energy

Fig. 1. ILP-Phases, Delay and Energy of *jpeg* application for different issue-widths.

TABLE I
NUMBER OF PHASES DETECTED IN MIBENCH BENCHMARK SUITE.

benchmark	8-issue	4-issue	2-issue	total
dijkstra	0	29	28	57
blowfish	0	16407	16406	32813
gsm-untoast	0	90	89	179
tiff2rgba	1	934	932	1866
patricia	0	36	35	71
FFT	22552	25616	3258	51426
jpeg-cjpeg	0	1954	1953	3907
rijndael	0	1663	1662	3325
tiffmedian	2	837	835	1674
gsm-toast	0	208	207	415
bitcount	7	43	35	85
adpcm-rawcaudio	0	1388	1387	2775
sha	0	148	147	195
adpcm-rawdaudio	0	2142	2141	4283
CRC32	0	365	364	729
tiff2bw	1	790	788	1578
basicmath	36498	49177	13986	99661
qsort	0	98	97	195
susan	199	242	340	781
jpeg-djpeg	0	759	758	1517
pgp	212	2779	2947	5938

formance are the main (design) requirements. Design-time and run-time reconfiguration are usually fine-grain or coarse-grain via a set of parameters in order to deal with changing execution environment of application requirements. In [6], the authors propose a template for a reconfigurable instruction set for the VLIW processor. But the instruction set can only be changed at design-time. A dynamic reconfigurable processor is presented in [7], namely i-core, which can migrate at run-time to a special-purpose processor by adapting its micro-architecture. But it does not present a complete scheme to trigger the migration. In [8], multi-objective design space exploration is used to enable run-time resource management for reconfigurable architectures. However, the run-time resource management only manages/reconfigures resources between applications. During the application execution, the resource management still needs to be considered, especially when the application has a long execution time. In our approach, we target the dynamic reconfigurable VLIW processor and process a finer-grained resource management, which is based on instruction monitoring.

Traditional VLIW processors only have a fixed issue-width. Many researches aim at exploiting the instruction-level parallelism of programs to maximize the utilization of all data-paths. In [9], the authors present an instruction set exploration method for clustered VLIW processor. In [10], the streaming programming model is applied on the NoC-based processor. An embedded multi-ported RAM that can be customized to match the issue-width of VLIW processors, which makes wide-issue VLIW processor viable on the FPGAs is introduced in [11]. Although the utilization of the VLIW processor resources is increased, there are still periods where the parallelism is not guaranteed, e.g., during a memory access or when there is a sequence of dependent instructions. Consequently, matching the resources to the ILP during run-time can lead to improved resource utilization and reduced energy consumption.

In addition, the power-efficiency is becoming an increasingly important research topic. Many researchers focus on proposing an efficient way to trade off the power consumption and performance. This is more keen on the embedded domain, which is constrained by power/resource usage. In [12], the authors present run-time phase monitoring and prediction on a real-time system. They use the branch prediction history to guide the dynamic voltage and frequency scaling (DVFS) to save power. But their approach is only applicable on a traditional processor, which supports DVFS. Locality profiling and run-time prediction are combined in [13] to predict the following execution. However, they only use the information from the first phase to predict all its later execution, which may incur wrong prediction for longtime execution. In our approach, we divide the execution time of an application into executing intervals and utilize monitoring information from preceding intervals as a prediction for the current interval. Additionally, with priorities on delay, energy, or EDP, the predictor decides to effect the reconfiguration.

III. DESIGN SPECIFICATIONS

For our approach, we target an implementation of the ρ-VEX processor, which is a run-time reconfigurable VLIW processor based on the VEX (VLIW Example) [14] ISA. The issue-width can be dynamically reconfigured to 2, 4, or 8. The reconfiguration process only takes 5 cycles to write the control registers and flush the data-paths. This is achieved by using generic binaries, i.e., a single binary for all configurations. This allows for precise execution suspension before and execution continuation after a configuration change

at any moment in time without the need for check-pointing. Furthermore, the state of the program execution is maintained in the core alleviating the need for (expensive) context switching. The ρ-VEX core also contains several control registers that autonomously control the reconfiguration. This simplifies our approach in designing new monitoring and predicting algorithms to drive the reconfiguration.

A. Prediction framework overview

On the ρ-VEX processor, we introduce a run-time phase prediction system to guide the reconfiguration of the processor core. The key idea of our approach is to utilize simple hardware modules to predict how the code would execute (in terms of delay and energy) when it is executed on a 8-, 4-, and 2-issue core per interval. This prediction is based on the delay and energy information from the past interval(s) used to reconfigure the core for the next interval. For the power-efficiency, 3 strategies are used to trade off between delay and energy. The prediction system is designed as an integrated hardware module that runs in parallel to the reconfigurable ρ-VEX core.

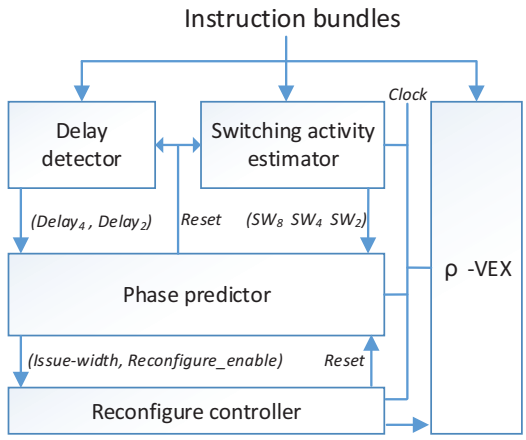


Fig. 2. Prediction system framework

The framework of the system basically contains 3 hardware modules: the delay detector, the switching activity estimator, and the phase predictor. These three modules work as depicted in Fig. 2. The input of the prediction system is the instruction bundles from the instruction fetch unit of the ρ-VEX core. The instruction bundle with 8 syllables is fetched from the instruction cache and sent to both the ρ-VEX core and the prediction system. The instruction bundles are sent to the delay detector and the switching activity estimator. The delay detector calculates the delays of 4- and 2-issue cores compared to an 8-issue core. The switching activity estimator counts the switching activities of the 8-, 4-, and 2-issue cores. After a certain time interval, the phase predictor reads the registers in the delay detector and the switching activity estimator. Based on these numbers, the phase predictor predicts the issue-width for next execution phase. If reconfiguration is needed, the phase predictor sets the *reconfigure_enable* signal to inform the reconfiguration controller and reset the registers in the delay detector and the switching activity estimator. The reconfigure controller reconfigures the ρ-VEX core according to the predicted issue-width coming from the phase predictor.

B. Delay detector

The delay detector utilizes the execution time of the 8-issue core as the reference, and estimates the delays of 4- and 2-issue cores. For the hardware implementation, a bit named stop-bit in the syllable is set to indicate if it is the last syllable with an actual operation in the instruction bundle. This bit is only set when the syllable is the last one of the bundle. That is to say, the syllables after the stop-bit are no-operations (NOPs).

Fig. 3 depicts the structure of the delay detector, and the procedure of the delay detecting is depicted in Fig. 4. The $SB_0\ SB_1\ \ldots SB_7$ are the stop-bits of an instruction bundle. One of them will be set to '1' if it is the end of a bundle. The decoder decodes these 8 bits to a binary with 3 bits according to the position of '1' in the 8 bits from the left side. For instance, '0000 0010' will be decoded to '111' since '1' is the seventh counted from the left side. After that, the RS_2 and RS_1 shift the decoded binary to right by 2 bits and 1 bit, respectively. By the shifting, we can get the correct delays of the 2- and 4-issue cores, which is stored in the Reg_2 and Reg_4.

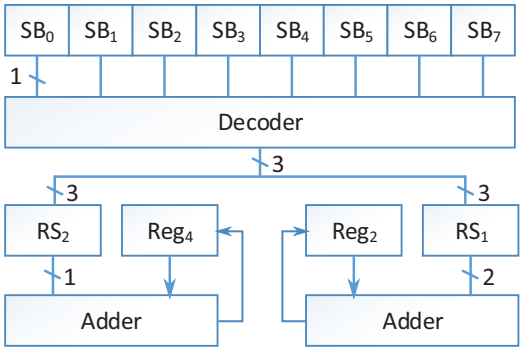


Fig. 3. Delay detector

Input:
$SB_0SB_1SB_2SB_3SB_4SB_5SB_6SB_7$: Stop-bits.
Output:
$delay_4, delay_2$: The delays of 4- and 2-issue cores.

1: **procedure** DELAY_DETECTOR
2: $b_2b_1b_0 \leftarrow decode(SB_0SB_1SB_2SB_3SB_4SB_5SB_6SB_7)$
3: $delay_4 \leftarrow delay_4 + b_2$
4: $delay_2 \leftarrow delay_2 + b_2b_1$
5: **end procedure**

Fig. 4. Procedure of delay detecting

Here is an example of the delay detecting process. Assuming there is an instruction bundle with 8 syllables in the following pattern: '$op_1, op_2, nop, op_3, op_4, nop, nop, nop$'. The op_x is the actual operation. The corresponding stop-bits should be

'0000 1000'. The decoded value is 5 (the bit position of the stop-bit). We right shift this value by two to get the number of additional cycles the core would take to execute this bundle on a 4-issue core, compared to an 8-issue core. We right shift by 1 to get the delay for the 2-issue core. This way we can quickly calculate that it takes 1 cycle, 2 cycles, and 3 cycles to execute this bundle on in 8-, 4-, and 2-issue cores, respectively.

C. Switching activity estimator

The switching of data-path can be classified as *inactive switching* and *active switching* according to the energy cost. The *inactive switching* does not cost energy while the *active switching* does. The NOP-to-NOP switching is the only *inactive switching* on the VLIW core and the others are *active switches*. In order to estimate the power consumption of data-paths, the switching activity estimator is designed to count the *active switches* of all data-paths.

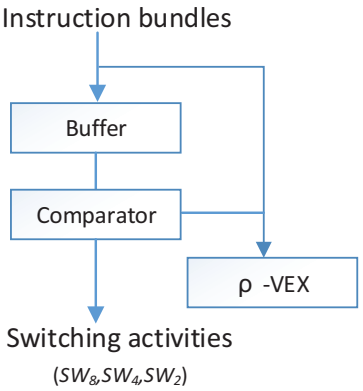


Fig. 5. Structure of switching activity estimator

Fig. 5 presents the structure of the switching activity estimator. The input instruction bundles are buffered and sent to the execution units. After that, the current instruction is compared with the previous one that is stored in the buffer. For a more efficient comparison, the instruction bundles are converted to an 8-bit binary representation, where the bit value indicates that it is actual operation ('1') or NOP ('0') in each data-path. Fig. 6 shows the procedures to calculate the switching activities of different issue-width core. It is worth mentioning that switching estimation for all three issue-widths is done in parallel. The split function splits the binary representation of the instruction bundle into bundles with the size of the issue-width. These bundles are compared with the previous bundle using a bitwise *or*. Subsequently, the number of '1's in the result is the number of switches when comparing the previous instruction bundle to the current one.

The power consumption of processor can be divided into two parts:

- *Dynamic part:* the power consumption of the data-paths, which is dynamically changed during the execution.
- *Static part:* the power consumption of register files and related logic, which keeps stable during the execution.

Input:
$b_7b_6b_5b_4b_3b_2b_1b_0$: Converted binary of the current instruction.
Output:
sw_8, sw_4, sw_2: Switching activities of 8-, 4-, and 2-issue core.

```
1: procedure CALCULATE_SWITCHING
2:     for all i ← 2, 4, 8 do
3:         for all b ← SPLIT(b7b6b5b4b3b2b1b0, i) do
4:             sw_i ← sw_i + BIT_COUNT(OR(b, prev_b_i))
5:             prev_b_i ← b
6:         end for
7:     end for
8: end procedure
```

Fig. 6. Procedure to calculate switching activity.

TABLE II
POWER CONSUMPTION OF EACH ISSUE-WIDTH

Issue-width	$P_{dynamic}$ (*pJ/cycle*)	P_{static} (*pJ/cycle*)
8	0.306	0.533
4	0.153	0.325
2	0.077	0.225

We use the Cadence Encounter RTL compiler [15] to measure these two parts of power consumption of the 8-, 4-, and 2-issue cores in the scenario that all data-paths are fully occupied. The result is presented in Table II. Using this data, the power consumption can be estimated as Eq. 1 depicts, where *active_sw* and *inactive_sw* are the counts of active and inactive switches.

$$Power = \frac{active\_sw}{active\_sw + inactive\_sw} * P_{dynamic} + P_{static} \quad (1)$$

D. Phase predictor

The phase predictor is used to predict the issue-width for the next execution phase. In our approach, the *phase* can be defined from two perspectives:

- For the application, the *phase* is a period of execution that requires similar issue-width in terms of delay and energy, which may include several intervals.
- For the VLIW processor, the *phase* is a period between two actual reconfigurations of the issue-width.

The predictor periodically samples the delay and switching activity information from the previous modules and resets the registers in those modules. A threshold is introduced to minimize the overhead of reconfiguration. In the current phase (from last reconfiguration to the current), only if the proportion of certain issue-width predictions exceed the threshold, the reconfiguration will be performed, and the core is reconfigured into a new phase.

Fig. 7 presents the phase prediction algorithm. Except for the delay and switching activity information from the previous steps, *interval* and *threshold* are also included as the inputs. The outputs are the predicted *issue_width* for the

Input:
$delay_4, delay_2, sw_8, sw_4, sw_2, interval, threshold.$
Output:
$issue\_width, reconfigure\_enable$

procedure PHASE_PREDICTION
▷ *Calculate Delay, Energy and EDP for each issue-width:*
  for all $i \leftarrow 2, 4, 8$ **do**
    $D_i \leftarrow delay_i + interval$
    $E_i \leftarrow ((sw_i/i) * P_{dynamic_i} + P_{static_i}) * D_i$
    $EDP_i \leftarrow D_i * E_i$
  end for
▷ *Make prediction based on strategy ①, ②, or ③:*
  ① *Delay prior:*
  $c_x \leftarrow c_x + 1 \mid delay_x \leftarrow min\{D_8, D_4, D_2\}$
  ② *Energy prior:*
  $c_x \leftarrow c_x + 1 \mid energy_x \leftarrow min\{E_8, E_4, E_2\}$
  ③ *EDP prior:*
  $c_x \leftarrow c_x + 1 \mid EDP_x \leftarrow min\{EDP_8, EDP_4, EDP_2\}$
  $issue\_width \leftarrow \{y \mid c_y \leftarrow max\{c_8, c_4, c_2\}\}$
  $numPredictions \leftarrow numPredictions + 1$
  if $numPredictions \bmod 10 = 0$ **then**
    $majority \leftarrow max(c_8, c_4, c_2)$
    if $majority > threshold$ **then**
      $reconfigure\_enable \leftarrow 1$
      $c_8, c_4, c_2, numPredictions \leftarrow 0$
    end if
  end if
end procedure

Fig. 7. Phase prediction algorithm

next phase and the *reconfigure_enable* signal that triggers the reconfiguration. In addition, the *reconfigure_enable* signal is also used to reset the counters/registers in the delay detector and switching activity estimator. First, the execution times of the 8-, 4-, and 2-issue cores are calculated. Subsequently, using Eq. 1, the energy of the past interval can be calculated. Finally, the EDP can be calculated by simply multiplying energy and delay. Next, we place different priorities on delay and energy to trade off these two metrics:

- *Delay prior:* the issue-width with the shortest delay will be chosen. If there are two issue-widths with the same delay, then the smaller issue-width is chosen, leading to lower power consumption.
- *Energy prior:* the issue-width with the smallest energy consumption will be chosen. If there are two issue-widths with the same energy consumption, then the higher one is chosen, which results in better performance.
- *EDP prior:* the issue-width with the smallest EDP value will be chosen.

There are 3 counters (c_8, c_4, c_2) to record how many times an issue-width of 8, 4, or 2 is chosen. After 10 predictions, the majority value will be calculated to verify that it is the right time to perform the reconfiguration. If the majority value is larger than the threshold, the *reconfigure_enable* signal will be set to trigger the reconfiguration and the counters will be reset to '0'. Otherwise, the predictor will enter the next time interval and keep the values of delay detector and switching activity estimator.

IV. Evaluation

To evaluate our approach, we selected 21 benchmarks from the Mibench [4] embedded benchmark suite. All the benchmarks are compiled with the VEX compiler and assembled into generic binaries that allow execution in different issue-widths. The benchmarks are executed on an 8-issue ρ-VEX and the instruction trace is retrieved for analysis afterwards. The ρ-VEX is prototyped on a Xilinx ML605 board using GRLIB [16] to connect to external peripherals such as DDR memory, UART, and timers and it is running the Linux OS when executing the benchmarks. The instruction traces generated on the FPGA are then parsed with a script which implements the prediction scheme and estimates the effect of changing issue-widths on performance and energy consumption. Since our approach is not specific to FPGAs and we wish to target low power applications, we estimate the energy consumption using the Cadence Encounter RTL compiler.

From the instruction trace of each benchmark we determine the energy and delay results for six possible run-time configurations, namely: the three fixed issue modes (2-, 4-, and 8-issue) and the three strategies mentioned earlier (delay prior, energy prior, EDP prior). Fig. 8 shows the results for each benchmark for each of these configurations. The values are normalized to the highest values for energy and delay.

We can identify applications with different characteristics in this figure. For example we can see that the *gsm-toast* benchmark has low ILP, because the delay of the 4- and 8-issue execution are close to each other. We can also see that *FFT* has high ILP, because the 8-issue core is about 15% faster than the 4-issue core. In this figure we can see that depending on the strategy chosen we can make a trade-off between performance and energy consumption. More importantly, this technique allows us to run an application with low ILP like *gsm-toast* on an 8-issue core without the need to pay the full energy cost of running it on an 8-issue core. Meanwhile, *FFT* can be run on the same core without sacrificing performance. In short, our approach allows us to run 2-issue applications on an 8-issue core with the energy cost similar to that of a 2-issue core, while also allowing us to run an 8-issue application with the full performance of an 8-issue core.

From the same figure we can also see that for certain applications we can achieve better results than when using a static issue-width. The *delay prior* strategy for the *pgp* application achieves the same performance as the 8-issue core, for 5% less energy. Similarly, for *FFT* we can actually achieve lower energy consumption than the 2-issue, because different parts of the application have different ILP. Of course the prediction is not perfect. The *blowfish* benchmark shows that the *delay prior* strategy will result in slower execution and higher energy consumption than a static 4-issue. This is due

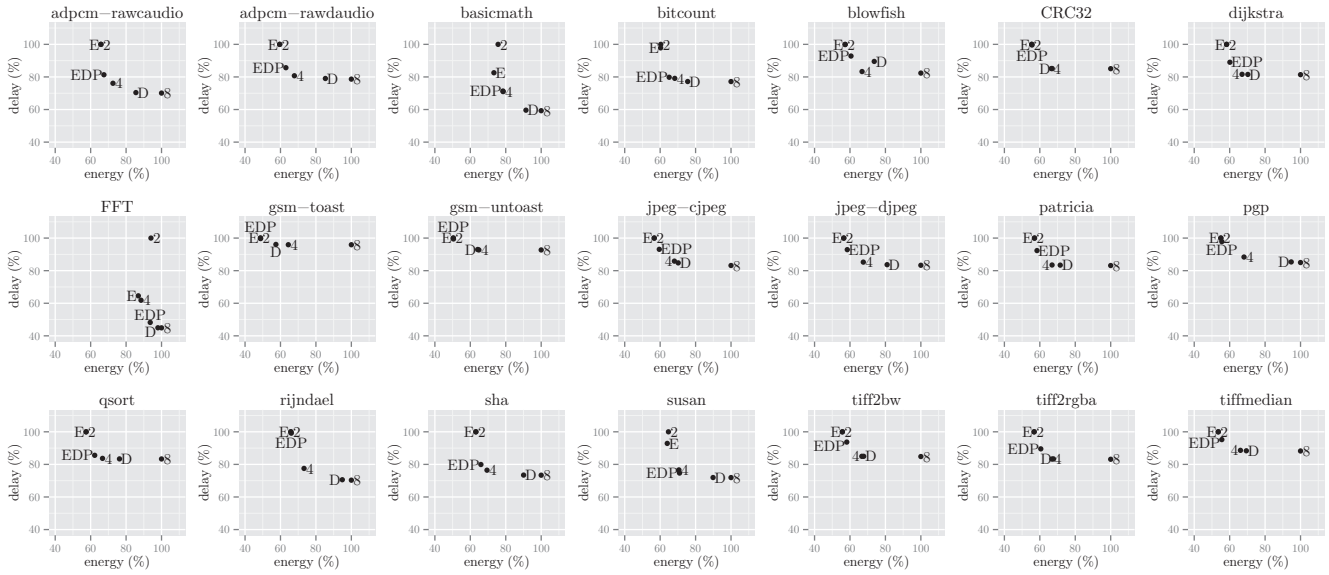


Fig. 8. Pareto plots showing energy and delay in percentage for each benchmark, normalized to the highest values. The points indicate the results of running a benchmark on a fixed issue-width (2-, 4-, or 8-issue), or with a particular issue-width selection strategy (E, D, or EDP). Some of the points overlap.

to phases changing in such a way that there are many miss-predictions.

V. Conclusion

In this paper, we proposed a run-time prediction scheme to drive the reconfiguration of a dynamic VLIW processor. The prediction scheme is based on 3 strategies for 3 different optimization goals: higher performance, reduced energy, and lower EDP. We implement the prediction module that runs in parallel with the ρ-VEX core. The experiments are demonstrated using a prototype running on an FPGA board using the Mibench benchmark suite. The results show that all 3 strategies can achieve the target optimization purpose. Specifically, the "delay prior" and the "energy prior" can achieve the best performance and the lowest energy consumption, respectively. And the "EDP prior" trades off between performance and energy. We do notice that for some benchmarks the strategies do no make the best decisions. Therefore, in future work, we will investigate ways to improve our phase prediction technique. This would result in better utilization of the processor and a reduction of energy consumption.

References

[1] S. Wong, T. van As, and G. Brown, "p-vex: A reconfigurable and extensible softcore vliw processor," in *ICECE Technology, 2008. FPT 2008. International Conference on*, Dec 2008, pp. 369–372.

[2] F. Anjam, M. Nadeem, and S. Wong, "Targeting code diversity with run-time adjustable issue-slots in a chip multiprocessor," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*. IEEE, 2011, pp. 1–6.

[3] A. Brandon and S. Wong, "Support for dynamic issue width in vliw processors using generic binaries," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '13, 2013, pp. 827–832.

[4] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, Dec 2001, pp. 3–14.

[5] A. Chattopadhyay, "Ingredients of adaptability: a survey of reconfigurable processors," *VLSI Design*, vol. 2013, p. 10, 2013.

[6] P. O. De Beeck, F. Barat, M. Jayapala, and R. Lauwereins, "Crisp: A template for reconfigurable instruction set processors," in *Field-Programmable Logic and Applications*. Springer, 2001, pp. 296–305.

[7] M. Hubner, C. Tradowsky, D. Gohringer, L. Braun, F. Thoma, J. Henkel, and J. Becker, "Dynamic processor reconfiguration," in *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*. IEEE, 2011, pp. 123–128.

[8] G. Mariani, V. Sima, G. Palermo, V. Zaccaria, C. Silvano, and K. Bertels, "Using multi-objective design space exploration to enable run-time resource management for reconfigurable architectures," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*. IEEE, 2012, pp. 1379–1384.

[9] R. Jordans, R. Corvino, L. Józwiak, and H. Corporaal, "Instruction-set architecture exploration strategies for deeply clustered vliw asips," in *Embedded Computing (MECO), 2013 2nd Mediterranean Conference on*. IEEE, 2013, pp. 38–41.

[10] G. Jiang, Z. Li, F. Wang, and S. Wei, "Scheduling stream programs with improving arithmetic unit usage on noc-based vliw multi-core architectures," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*. ACM, 2015, p. 18.

[11] M. Purnaprajna and P. Ienne, "Making wide-issue vliw processors viable on fpgas," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, p. 33, 2012.

[12] C. Isci, G. Contreras, and M. Martonosi, "Live, runtime phase monitoring and prediction on real systems with application to dynamic power management," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 359–370. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2006.30

[13] X. Shen, Y. Zhong, and C. Ding, "Locality phase prediction," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XI. New York, NY, USA: ACM, 2004, pp. 165–176.

[14] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded computing: a VLIW approach to architecture, compilers and tools*. Elsevier, 2005.

[15] "Cadence palladium dynamic power analysis," http://www.cadence.com/eu/Pages/rtl_compiler.aspx, accessed May 1, 2015.

[16] I. GRLIB, "library users manual," *Version*, vol. 1, no. 0, p. B4100, 2012.