Evaluating Auto-adaptation Methods for Fine-Grained Adaptable Processors

Joost Hoozemans(✉), Jeroen van Straten, Zaid Al-Ars, and Stephan Wong

Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands
{j.j.hoozemans,j.vanstraten-1,z.al-ars,j.s.s.m.wong}@tudelft.nl

Abstract. To achieve energy savings while maintaining adequate performance, system designers and programmers wish to create the best possible match between program behavior and the underlying hardware. Well-known current approaches include DVFS and task migrations in heterogeneous platforms such as big.LITTLE processors. Additionally, processors have been proposed in literature that are able to adapt (parts of) their organization to the workload. These reconfigurations can be managed using hardware monitors, profiling and other compile-time information or a combination of both. Many current solutions are suitable for heterogeneous systems, as migration penalties pose a practical limit to the maximum adaptation frequency, but not for dynamic processors that can adapt much more fine-grained.

In this paper, we present two novel concepts to aid these low-penalty reconfigurable processors - one requiring an ISA extension and one without. Our experimental results show that our approaches enable a dynamic processor to reduce the energy-delay product by up to 25% and on average 10% to 18% compared to the best performing static setups.

1 Introduction

With energy utilization as a new critical metric for computing systems, designers have devised numerous ways of configuring systems to run in various performance/power modes. The most notable examples are Dynamic Voltage and Frequency Scaling (DVFS), Heterogeneous Multicore Processors (HMPs) such as big.LITTLE, and polymorphic processors such as MorphCore [1]. In turn, researchers try to match program behavior to processor configurations in order to minimize both the energy utilization and the performance penalty associated with low-power configurations.

The time it takes to move an ARM big.LITTLE core in or out of sleep modes lies in the order of *milliseconds* and changing DVFS involves a latency of tens of microseconds. Furthermore, migrating a task to another core will introduce an additional penalty because of cold resources (cache, predictors) [2]. Because of these properties, a granularity of context-switch level (10 ms) is adequate, as adapting to the workload any faster will only result in prohibitively large penalties.

M. Berekovic et al. (Eds.): ARCS 2018, LNCS 10793, pp. 255–268, 2018.
https://doi.org/10.1007/978-3-319-77610-1_19

In contrast to this, program characteristics can change at much higher frequencies [3]. Therefore, designs have been proposed that greatly reduce these penalties for heterogeneous systems [2,4], and adaptable processors have been proposed that have very low adaptation penalties [1,5]. These processing platforms have the potential of matching the program in a far more fine-grained way (in the time domain). However, currently used monitoring-based approaches are often based on measurement windows that are far too large to drive these high-frequency adaptations.

This work aims to determine what evaluation frequency is needed to profit from fine-grained adaptable processors. As sampling performance counters at this rate will create excessive overhead, we argue that an automatic evaluation circuit is required, moving the evaluation and adaptation control loop into hardware. Next to sampling performance counters, we propose two additional auto-adaptation approaches. In one approach, we modified the compiler to insert instructions in locations that are likely to correspond with a phase boundary. When encountering this instruction, the processor starts a measurement and stores the results in a dedicated field in the same instruction word. The second approach involves a branch target buffer. At every branch, a measurement is started and results are stored in the buffer. When branching to the same target address again, the code characteristics have already been measured and can be retrieved. These two approaches aim to make adaptations more proactive.

We have applied the approaches to the ρ-VEX dynamic VLIW (very long instruction word) processor that is able to change configurations with a penalty of only 5 cycles (a pipeline flush). Results show that the ρ-VEX processor benefits from monitoring windows of approximately 75 cycles. Using the auto-adaptation approaches, the energy consumption of the adaptable processor can be reduced by 10% to 18% on average compared to the best static setup. The branch-based proactive approach slightly outperforms window-based solutions.

2 Approach

2.1 Target Processor

In this work, we target the ρ-VEX processor, an open-source reconfigurable VLIW processor [6]. It can assign datapaths in pairs to one or multiple threads or disable them to conserve energy (see Fig. 1). It has a reconfiguration penalty of 5 cycles, because it needs to flush the pipeline. The processor can switch between a 2, 4, or 8-issue configuration without changing the binary it is executing, because it utilizes generic binaries [7]. In short, generic binaries work by ensuring that each VLIW bundle of 8 operations can also be executed in 2 or 4-issue mode, by removing intra-bundle dependencies (see Fig. 2 for a simplified depiction of this).

VLIW architectures are widely adopted in embedded media and DSP applications, providing high energy efficiency (for example, in modem, audio and image processing subsystems in mobile phone SoCs) [8]. Code for VLIWs is statically scheduled by the compiler, decreasing hardware complexity. Instruction-level

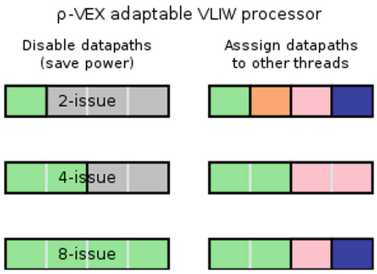


Fig. 1. Conceptual depiction of the fine-grained reconfigurable VLIW processor targeted in this work. It consists of 8 datapaths that can be split or merged in pairs (i.e., each sub-block represents a 2-issue VLIW processor). These can be assigned to a thread or powered down to conserve power (left-hand side). Multiple blocks can be assigned to a single thread to exploit as much ILP as possible, or each block can be assigned to its own thread to exploit thread-level parallelism (right-hand side - the colors represent different threads).

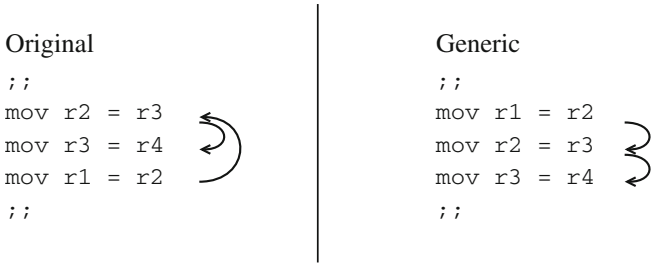


Fig. 2. The ρ-VEX is able to switch configurations at any time, because the toolchain makes sure the code can be executed in every possible configuration. It does this by 're-sequentializing' the code after it has been compiled for 8-issue. Each bundle is reordered such that the dependencies (shown as arrows) are met when executing the operations one by one.

parallelism (ILP) is explicitly encoded in the binary. This makes it possible to measure performance of different core configurations, as we will see in Sect. 3. This makes the chosen VLIW platform very suitable to evaluate the proposed techniques.

2.2 Proposed Auto-adapting Method

The main idea behind our approach is that program characteristics change during the course of execution, but characteristics of code itself is fixed. In other words, the changes are due to the control flow through the different code sections in the binary. We propose to measure these characteristics once for every code section, and store this information in such a way that we can easily retrieve it whenever we revisit that section. For each section, a measurement only needs to

be performed once for each core type (for HMPs) or configuration (for adaptable processors), after which the results for both are stored in their own field.[1] We are proposing two ways to store the measured code characteristics.

The first approach utilizes a structure that is similar to the branch target buffer (BTB) that is widely used in modern processors. Normally, the BTB is used to predict the branch target address early in the pipeline to reduce branch penalties. Our 'Branch Target Configuration Buffer' (BTCB) is a cache that is indexed by branch target addresses. Whenever a branch occurs, the BTCB is accessed to determine if there is information about the code that is being jumped to. If there is not, a measurement is triggered. When the next branch occurs, the measurement results are stored in the buffer. If there is information in the buffer, it can be used during the branch to reconfigure the processor to the most energy efficient configuration.

Our second approach introduces a special instruction we named `pchg` (phase change) that is added to the program by the compiler at certain locations that are likely to correspond with a longer, more stable phase (compared to the first approach, that operates on a basic block level). When encountering this instruction, a lookup is performed in a configuration buffer similar to the BTCB. This lookup can use the least significant bits of the PC (program counter) as index, or the compiler can assign indexes to code sections and place their index in the instruction.

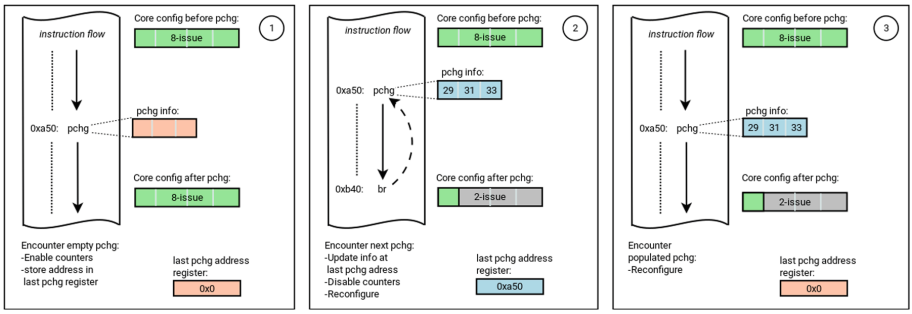


Fig. 3. Overview of the pchg approach when encountering a loop, using the PC address as configuration buffer index.

During runtime, when the processor encounters this instruction for the first time, it keeps track of the index and starts the performance counters to evaluate the program characteristics in that phase. When the measurement has completed (when encountering the next `pchg` instruction), the results of the measurement

[1] On HMPs, measuring performance on one core type does not provide information about the performance on the other core type (see [9, Sect. 6.3]). To monitor which core type is the most efficient, the program needs to be migrated back and forth continuously. The same holds for different configurations of an adaptable processor.

are written back into the configuration buffer. Each time the processor encounters the instruction again, the information is available and the processor can use it to perform a reconfiguration immediately. An overview of the `pchg` approach is depicted in Fig. 3. Both approaches have their merits. The first approach is the most fine-grained but may trigger adaptations too often. The second approach requires recompilation of binaries (note that, if this is not possible, old binaries will still execute correctly but not trigger any adaptations) and results in runtime overhead because of the added instructions.

3 Implementation

This section discusses the implementation of the different approaches in the target platform. We start with the elements that the different approaches have in common, then we discuss the window-based monitoring approach, followed by the BTCB approach, and concluding with the phase change annotations.

3.1 Common

The target processor has a controller that handles reconfiguration requests. These requests can be performed via a memory-mapped control register writable by software (user or OS). Although the platform reduces adaptation overhead to only 5 cycles, sampling and evaluating performance counters in software introduces additional overhead. At the frequencies we are proposing in this paper, this overhead becomes very significant. Therefore, we propose to use a hardware circuit to perform the evaluation and reconfiguration request directly. This section discusses this circuit.

We use a performance counter for each possible ρ-VEX core configuration. Using a scheme similar to [10], we increment these counters based on the location of a VLIW bundle marker. If a bundle is completely filled with 8 operations, the counter for the 2-issue configuration will increase by 4 and the counter for the 4-issue configuration will increase by 2 (see Fig. 4). This scheme is enough to measure the performance of the configurations. However, we propose to estimate energy utilization.

We have used the following energy estimation function:

$$E = E_{static} + E_{dynamic} \text{ where}$$
$$E_{dynamic} = (SYL * E_{syl}) + (NOP * E_{nop}) \text{ and}$$
$$E_{static} = (CYC_2 * E_{cyc2}) + (CYC_4 * E_{cyc4}) + (CYS_8 * E_{cyc8}).$$

Here, SYL is the number of execution syllables (individual operations of a VLIW bundle), NOP is the number of unfilled syllable slots, and CYC represents the number of executed cycles in 2-issue, 4-issue and 8-issue mode. The energy values depend on the hardware characteristics and should be set by the designer based on power estimations or measurements. For our evaluation we have used the values listed in Table 1. The dynamic part of the function is largely the same between configurations, so we can use a single cost value for each configuration.

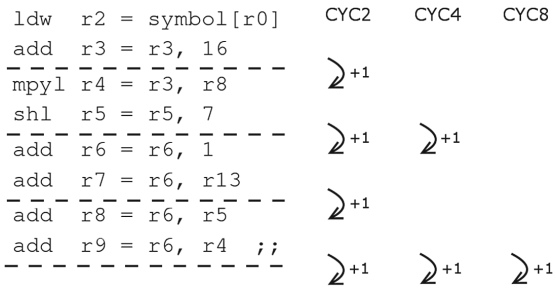


Fig. 4. Measuring the performance for different configurations is done by decoding the location of the stop bit (VLIW bundle boundaries shown as ';;'). This bundle requires 4 cycles to execute on the 2-issue configuration and 2 cycles on the 4-issue. The 8-issue counter is equivalent to the bundle counter.

Table 1. Used values for the energy estimation function in the simulator

E_{syl}	E_{nop}	E_{cyc2}	E_{cyc4}	E_{cyc8}
4	1	2	3	4

Instead of multiplying the counter values with the energy estimation values (which would be expensive in hardware), we propose to use prescaler counters. The prescaler is increased using the configuration cycle count of the bundle (as depicted in Fig. 4). When a configuration's prescaler exceeds its cost value, its energy estimation counter is increased by 1 and the prescaler is reset. The prescaler only needs enough precision to express the ratios between the cost values. The final energy estimation counters also needs limited precision, because (1) we are measuring relatively short sections of code and (2) if two estimations are very close to each other, both choices are equally suitable. In our current implementation, we are using 7 bits per configuration for the energy estimation counters. When any one of the counters overflows, all of them are right shifted by 1 position (the ratios between them stay intact). The required storage for the configuration buffer entries is 7×3 bits (one for each possible ρ-VEX configuration).

3.2 Window-Based Monitoring

Window-based monitoring is not a novel approach proposed in this paper but rather the current art to which we will compare. Using the hardware circuit from the previous section, our window-based implementation evaluates the energy estimation using a fixed period. The configuration with the lowest value is forwarded to the reconfiguration request register, and the counters are reset.

3.3 BTCB

For this approach we propose to add a buffer, the Branch Target Configuration Buffer (BTCB) that stores code information about branch targets. In case the processor already features a BTB, such as the Philips TriMedia VLIW [11], this structure can be widened to include the desired information.[2] When the processor executes a branch (conditional branches are only considered when taken), it will perform a lookup in the buffer to see if there is an entry with valid code information. If that is the case, it will perform a core adaptation.

If no such entry is found, the processor will start the performance counters. A register keeps track of the index of the entry. When a new branch is taken, this register is used to update the BTCB using the measured values. This can be done one cycle later than the new branch's BTCB lookup, to avoid requiring an additional access port. In our implementation, the BTCB is direct-mapped. Therefore, any collision (two branch addresses that map to the same BTCB entry) results in an eviction.

3.4 Phase Change Annotations

In this approach, the compiler identifies locations that are likely to correspond to a phase. In these locations, it adds an instruction, named `pchg` (phase change). The processor performs a lookup in the configuration buffer when encountering this instruction, instead of at every branch. We have modified the ρ-VEX compiler to add a `pchg` instruction at the top of every loop and every leaf function. The compiler can choose to skip loops and functions that it estimates to have a total execution time lower than a certain threshold.

4 Evaluation

4.1 Experimental Setup

To evaluate our approach, we have used the open source ρ-VEX polymorphic processor as discussed in Sect. 2.1. We have implemented our `pchg` approach in the compiler as discussed in Sect. 3 and modeled the monitoring hardware in the simulator. To measure only the behavior of the processor core, caches were disabled. Using this setup, the simulator is cycle-accurate regarding a ρ-VEX core attached to single-cycle instruction and data memories, as the code is completely statically scheduled. We will use MiBench [12] and SPECINT 2006 for our measurements. Not all programs could be used, as some are not supported by the ρ-VEX toolchain or libraries. We will use the modes listed in Table 2.

Here, the static setups represent the supported ρ-VEX configuration modes, without any runtime adaptations. The windowed modes utilize performance

[2] Note that in that case, it is no longer indexed by the branch *target* but rather the PC of the branch itself; the buffer will return the predicted branch target and we propose to add the code information for that branch target to the entry.

Table 2. Evaluated modes of execution.

Type	Modes
Static core	2-issue, 4-issue, 8-issue
Dynamic core, windowed	10,000, 1,000, 500, 250, 100, 75, 50
Dynamic core, `pchg`	pchg-0, pchg-100
Dynamic core, BTCB	BTCB-inf, BTCB-2048

monitoring with fixed windows of various sizes to perform core adaptations. The pchg modes utilize the proposed phase change annotations, with loop annotation thresholds of 0 and 100 cycles. BTCB uses the proposed branch target configuration buffer. We have evaluated a buffer with infinite entries and one with 2048 entries.

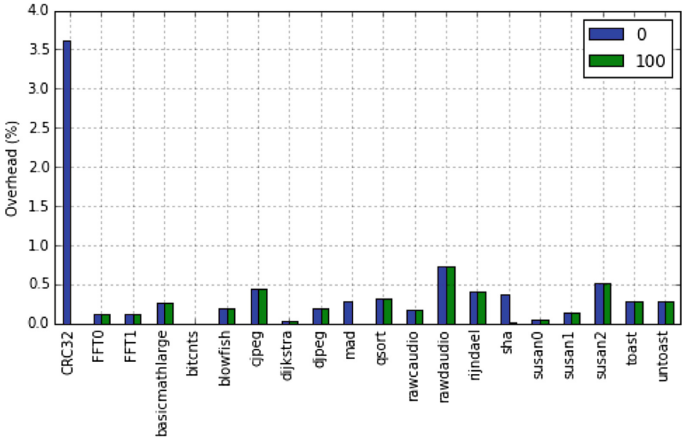
We will use the Energy-Delay Product (EDP) as metric and normalize to a static 8-issue configuration which represents the highest performing setup. Note that, due to the chosen values for the energy estimation function (see Table 1), the outcome for all measurements cannot be lower than 0.5, because no setup can execute faster than the 8-issue and the 2-issue energy estimation is 0.5× that of the 8-issue.

4.2 Results

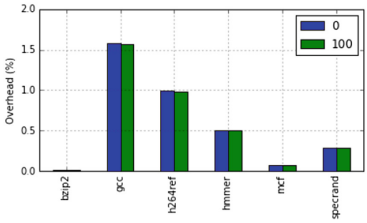
Overhead. Adding the `pchg` instructions into the programs results in runtime overhead. We have measured this overhead by running all 3 version of the binaries (not annotated, threshold 0, threshold 100 cycles) on a static 2-issue core. The results are plotted in Fig. 5. On average, the runtime overhead is quite acceptable at approximately 0.5% on average.

Window sizes. We evaluate windowed monitoring setups using various window sizes between 50 and 10,000 cycles. The results are plotted in Fig. 6. For both benchmark suites, the disadvantage (overhead) surpasses the advantage of higher frequency adaptations at approximately 75 cycles. Our measurements reveal that using a window size of 75 compared to 1000 cycles improves EDP up to 20% (for `specrand` and `rijndael`) and on average 6%, supporting our claim that code can change very frequently and a fine-grained reconfigurable processor is able to match these changes more closely.

Runlength thresholds. The energy estimation counters can use a minimum runlength threshold for a measured code section. If this threshold is not reached when the measurement is finished (because of a new `pchg` instruction, or because of a branch), the core will not perform an adaptation. We have evaluated different threshold values and the results are depicted in Fig. 7. In case the BTCB is limited in size to 2048 entries, there is a clear optimal threshold for MiBench of



(a) MiBench



(b) SPEC

Fig. 5. Overhead of adding the phase change instructions.

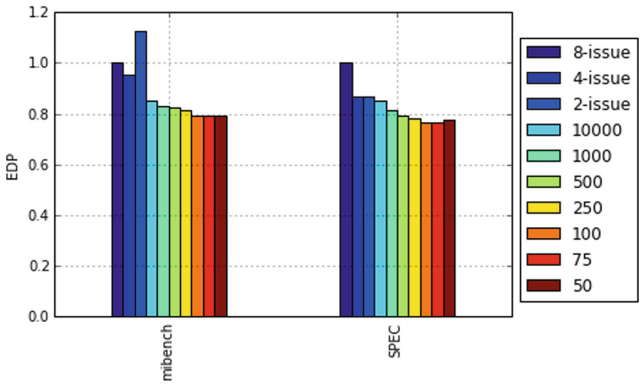
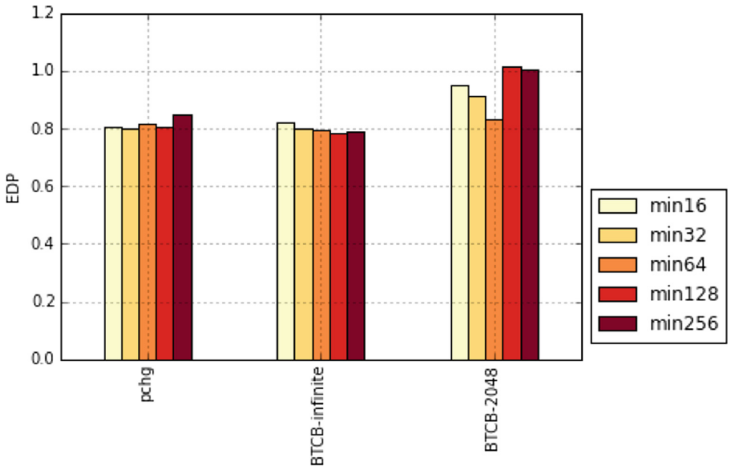
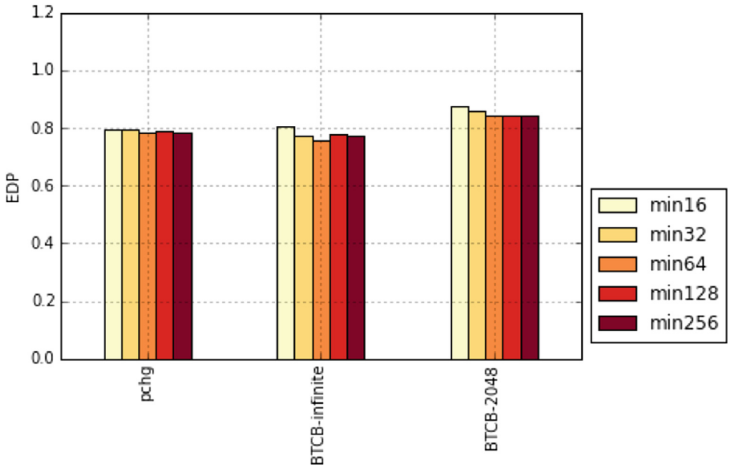


Fig. 6. EDP for different window sizes. For both benchmark suites, 75 instructions performs best.



(a) Mibench.



(b) SPEC.

Fig. 7. EDP for different runlength thresholds.

64 instruction bundles and the relative loss in performance (compared to the best performing setup with an infinite buffer) is in this case 6%. The other setups, as well as the SPEC benchmarks, are not as strongly influenced by the threshold. The loss can be attributed mostly to two outliers in the form of `basicmath` in MiBench and `specrand` in SPEC, that may suffer from a high number of collisions.

Comparing the approaches. Using the best results for each approach as reported in previous sections, we have plotted the averages of the different techniques in Fig. 8. The dynamic setups perform considerably better compared to the static cores. The first observation is that the window-75 setup performs relatively well, achieving 10% and 17% better EDP on average (for SPEC and MiBench, respectively), compared to the best performing 4-issue static core. The BTCB approach performs best, with on average 12% and 18% better EDP. The `pchg` annotations perform up to 26% and on average 10% (SPEC) and 16% (MiBench) better than the best performing static core.

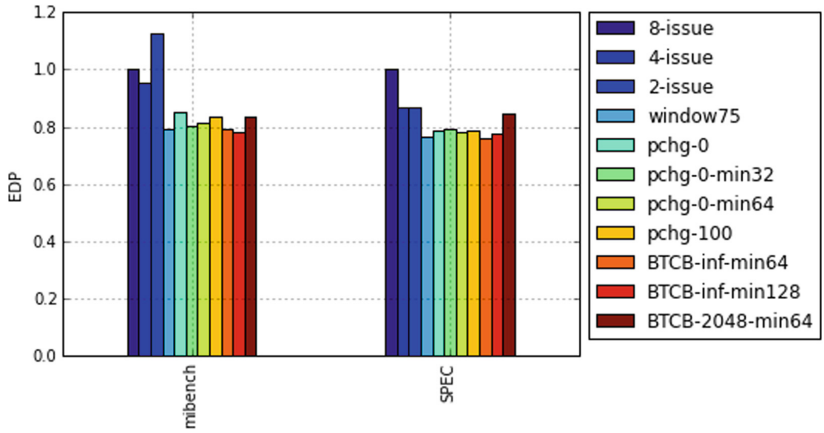


Fig. 8. EDP for the best performing setups for each approach.

For many programs, ILP variability is quite low, and the EDP for the dynamic approaches is not significantly lower than that of the best performing static setup. The largest gains are measured for the program `rawcaudio` with all approaches achieving approximately 25% better EDP than static setups. However, the window-1000 approach performs similarly for this program (indicating that fine-grained approaches do not provide an advantage) In contrast, `rijndael` does not show any improvement when using a 1000 cycle monitoring window, while our proposed BTCB approach provides 20% lower EDP compared to the best static core and 8% over the best window approach (75 cycles).

5 Related Work

The polymorphic processor used in our evaluations is discussed in more detail in [5]. Other dynamic processors that could make use of our proposed scheme are MorphCore [1], TRIPS [13] and CoreFusion [14]. Rodrigues et al. [15] propose a dynamic processor that morphs by allowing one core to take control over a functional unit residing in a neighboring core. They introduce a

dynamic phase classification scheme that uses a table to store and lookup phases. Guo et al. [10] built a windowed counter scheme for the ρ-VEX that predicts program phases and reconfigures the processor accordingly. Similarly, [16] tries to predict phases using statistical and table-based predictors. Chi et al. [17] show the advantage of combining static and dynamic profiling techniques to improve performance/energy tuning, focusing on disabling some processor resources and fetch throttling. Our approach uses compiler analysis instead of profiling as the static component.

In addition to dynamic processors, the scheme can be used by single-ISA heterogeneous multicore systems [18] such as ARM big.LITTLE processors [19], particularly, systems that were designed to have low migration penalties such as [2,4]. For schemes with similar objectives on HMPs see for example [3,9,20]. Related work in autotuning are for example [21,22], where hardware modules are introduced that perform evaluation of power and performance on a softcore processor. However, the purpose is to perform dynamic partial reconfiguration, which is very different from how the ρ-VEX works.

Sherwood et al. [23] propose a similar technique of using an on-chip buffer to store detected phases based on branches, but focusing on long, stable phases. In addition, they evaluate "Dynamic Processor Width Adaptation" similar to the ρ-VEX (but supporting only a 2-issue and 8-issue configuration). They perform a short measurement in both configurations at every phase change, which is one of the problems that our proposed solution aims to solve (see Sect. 2.2).

6 Conclusions

When targeting a highly dynamic processor that has a low reconfiguration penalty (in this work, the ρ-VEX with a penalty of 5 cycles), improvements in energy efficiency can be gained by using very fine-grained automatic adaptations. Evaluations of window-based autotuning of the configuration show that using a window of 75 cycles results in the best EDP (up to 20% better than a 1000 cycle window). This confirms that code characteristics can change very rapidly, and that the dynamic processor is able to follow the changes more closely than traditional autotuning schemes that use relatively large window sizes. Not all programs show this highly dynamic behavior.

The proposed approaches open up the possibility of superscalar-based, single-ISA heterogeneous or adaptable processors with low penalties. Using a window-based approach is not possible in this case, because it would need continuous migrations between core types to evaluate the code characteristics, negating the advantages. Using our proposed methods to store information about code sections, measurements need to be performed once in every configuration, after which the information is stored and can be retrieved when revisiting the section.

Overall, the approaches enable the reconfigurable processor to achieve up to 25% and between 10% and 18% on average better EDP compared to the best static platform. The proposed BTCB approach achieves the best results, slightly outperforming window-based autotuning.

Acknowledgements. This work has been supported by the ALMARVI European Artemis project nr. 621439.

References

1. Khubaib, Suleman, M.A., Hashemi, M., Wilkerson, C., Patt, Y.N.: MorphCore: an energy-efficient microarchitecture for high performance ILP and high throughput TLP. In: 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 305–316, December 2012
2. Brown, J.A., Porter, L., Tullsen, D.M.: Fast thread migration via cache working set prediction. In: 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA), pp. 193–204. IEEE (2011)
3. Rangan, K.K., Wei, G.-Y., Brooks, D.: Thread motion: fine-grained power management for multi-core systems. In: Proceedings of the 36th Annual International Symposium on Computer Architecture, ser. ISCA 2009, pp. 302–313. ACM, New York (2009). http://doi.acm.org/10.1145/1555754.1555793
4. Rodrigues, M., Roma, N., Tomás, P.: Fast and scalable thread migration for multicore architectures. In: 2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing, pp. 9–16, October 2015
5. Brandon, A., Hoozemans, J., van Straten, J., Wong, S.: Exploring ILP and TLP on a polymorphic VLIW processor. In: Knoop, J., Karl, W., Schulz, M., Inoue, K., Pionteck, T. (eds.) ARCS 2017. LNCS, vol. 10172, pp. 177–189. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-54999-6_14
6. Wong, S., van As, T., Brown, G.: ρ-VEX: a reconfigurable and extensible softcore VLIW processor. In: International Conference on Field-Programmable Technology (ICFPT), December 2008
7. Brandon, A., Wong, S.: Support for dynamic issue width in VLIW processors using generic binaries. In: Design, Automation Test in Europe Conference Exhibition (DATE), pp. 827–832, March 2013
8. Codrescu, L., Anderson, W., Venkumanhanti, S., Zeng, M., Plondke, E., Koob, C., Ingle, A., Tabony, C., Maule, R.: Hexagon DSP: an architecture optimized for mobile multimedia and communications. IEEE Micro **34**(2), 34–43 (2014)
9. Becchi, M., Crowley, P.: Dynamic thread assignment on heterogeneous multiprocessor architectures. In: Proceedings of the 3rd Conference on Computing Frontiers, ser. CF 2006, pp. 29–40. ACM, New York (2006)
10. Guo, Q., Sartor, A., Brandon, A., Beck, A.C., Zhou, X., Wong, S.: Run-time phase prediction for a reconfigurable VLIW processor. In: 2016 Design, Automation and Test in Europe Conference and Exhibition (DATE), pp. 1634–1639. IEEE (2016)
11. Hoogerbrugge, J.: Dynamic branch prediction for a VLIW processor. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, (PACT), pp. 207–214. IEEE (2000)
12. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: MiBench: a free, commercially representative embedded benchmark suite. In: 2001 IEEE International Workshop on Workload Characterization: WWC-4, pp. 3–14. IEEE (2001)
13. Sankaralingam, K., Nagarajan, R., Liu, H., Kim, C., Huh, J., Burger, D., Keckler, S.W., Moore, C.R.: Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In: Proceedings of the 30th Annual International Symposium on Computer Architecture, pp. 422–433. IEEE (2003)

14. Ipek, E., Kirman, M., Kirman, N., Martinez, J.F.: Core fusion: accommodating software diversity in chip multiprocessors. In: Proceedings of the 34th Annual International Symposium on Computer Architecture, ser. ISCA 2007, pp. 186–197. ACM, New York (2007). http://doi.acm.org/10.1145/1250662.1250686
15. Rodrigues, R., Annamalai, A., Koren, I., Kundu, S.: Improving performance per watt of asymmetric multi-core processors via online program phase classification and adaptive core morphing. ACM Trans. Des. Autom. Electron. Syst. **18**(1), 5:1–5:23 (2013). http://doi.acm.org/10.1145/2390191.2390196
16. Duesterwald, E., Cascaval, C., Dwarkadas, S.: Characterizing and predicting program behavior and its variability. In: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques, PACT 2003, pp. 220–231, September 2003
17. Chi, E., Salem, A.M., Bahar, R.I., Weiss, R.: Combining software and hardware monitoring for improved power and performance tuning. In: Proceedings of the Seventh Workshop on Interaction Between Compilers and Computer Architectures: INTERACT-7, pp. 57–64. IEEE (2003)
18. Kumar, R., Farkas, K.I., Jouppi, N.P., Ranganathan, P., Tullsen, D.M.: Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction. In: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture: MICRO-36, pp. 81–92. IEEE (2003)
19. Greenhalgh, P.: big.LITTLE processing with ARM cortex-A15 & Cortex-A7. ARM White Paper, pp. 1–8 (2011)
20. Van Craeynest, K., Jaleel, A., Eeckhout, L., Narvaez, P., Emer, J.: Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In: Proceedings of the 39th Annual International Symposium on Computer Architecture, ser. ISCA 2012, pp. 213–224. IEEE Computer Society, Washington, DC (2012). http://dl.acm.org/citation.cfm?id=2337159.2337184
21. Otero, A., Morales-Cas, A., Portilla, J., de la Torre, E., Riesgo, T.: A modular peripheral to support self-reconfiguration in SoCs. In: 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools, pp. 88–95 (2010)
22. Aldham, M., Anderson, J., Brown, S., Canis, A.: Low-cost hardware profiling of run-time and energy in FPGA embedded processors. In: ASAP 2011–22nd IEEE International Conference on Application-specific Systems, Architectures and Processors, pp. 61–68, September 2011
23. Sherwood, T., Sair, S., Calder, B.: Phase tracking and prediction. In: ACM SIGARCH Computer Architecture News, vol. 31, no. 2, pp. 336–349. ACM (2003)