Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2021) 29: 1964 – 1977
© TÜBİTAK
doi:10.3906/elk-2005-180

ShapeShifter: a morphable microprocessor for low power

Nazlı TOKATLI[1,*], İsa Ahmet GÜNEY[2], Sercan SARI[1], Merve YILDIZ GÜNEY[1], Uğur NEZİR[1], Gürhan KÜÇÜK[1]

[1]Department of Computer Engineering, Yeditepe University, Istanbul, Turkey
[2]Department of Computer Engineering, Doğuş University, İstanbul, Turkey

Received: 25.05.2020 • **Accepted/Published Online:** 14.02.2021 • **Final Version:** 26.07.2021

Abstract: A composite core contains large and small heterogeneous microengines. The most important property of composite cores is their ability to select the most proper microengine for running applications to save power without sacrificing too much performance. To achieve this, a composite core tries to predict the performance of the passive microengine by collecting various processor statistics from the active microengine at runtime. In the method proposed in the literature, the microengine, which is more ideal for running the rest of the application, is determined by a migration-decision circuitry that is bound to collected statistics and complex functions, which are run in a sequential manner. In this study, we propose the ShapeShifter architecture that holds a single out-of-order core to switch its mode of instruction execution between out-of-order and in-order modes. With a simple mode-change decision circuitry, which is bound to only two processor statistics, we can save more than 25% power, more than 21% on energy-delay product, and more than 16% on energy-delay-square product on the average, by only sacrificing less than 5% of performance.

Key words: Reconfigurable core, low-power design, self-adaptive system

1. Introduction

In the literature, the first study on composite cores was presented by Lukefahr and his team [1]. The big.LITTLE heterogeneous multicore architecture produced by ARM is a good example of the composite core architecture. In this architecture, a 3-wide out-of-order Cortex-A15 superscalar microprocessor (large) and a 2-wide in-order Cortex-A7 superscalar microprocessor (small) are located in the same core. In his work, Lukefahr and his team define a composite core as a heterogeneous multicore architecture, consisting of one large and one small compute microengine, which can provide both high performance and high energy savings.

In the existing big.LITTLE architecture of ARM, core migration is described as coarse-grained. In this architecture, many power-saving opportunities are missed when it comes to running applications that often change behavior as cross-core application migration occurs after billions of instructions are executed. Similarly, an application that demands instant high performance while working on the small microengine also remains attached to the small microengine for long periods of time. In summary, the current method does not have the reflex to follow the instant behavior changes in running applications.

At this point, Lukefahr and his team propose a fine-grained decision mechanism for core-to-core migration, and ultimately offer a method that is more agile to adapt to the needs of running applications. Here, we refer to the work proposed by Lukefahr and his team as the composite core (CC) study. The basis of the method

*Correspondence: nnakeeb@cse.yeditepe.edu.tr proposed in the CC is based on predicting the performance of the passive microengine using a function obtained by machine-learning methods that integrate statistics collected from the active microengine.

The collected statistics include a range of statistics from the L2 cache miss and hit rates to the misprediction percentage of the branch prediction mechanism, from the ınstruction-level parallelism (ILP) obtained from the ıssue queue (IQ) to the roughly calculated memory-level parallelism (MLP) using the miss status holding register (MSHR). To collect these statistics, complex tables such as a dependency table should be integrated into the small core. The decision circuit, which is called the reactive online controller, contains the performance estimator, the threshold controller, and the core switching control mechanisms. At this point, there are two main factors that motivate our work.

1.1. Low complexity

In this study, we show that, by providing a single out-of-order core and using only two processor statistics, we can surpass the complexity and performance results reported in the CC. The complexity of the hardware trained by a machine-learning method is quite high in that paper [1]. The authors also state that the method is not generic, and the performance prediction function may be trained offline for each and every embedded application.

In the ShapeShifter architecture that we propose in this study, there is no area required to keep an in-order core along with the large out-of-order core. This results in a great amount of complexity reduction on the die area. Our decision circuit for the execution mode switch does not require any machine-learning method and any table that needs to be kept in the hardware, as in the CC. In the CC, the total area of the control circuit covers about 3% of the processor area; whereas, the ShapeShifter utilizes a negligible number of logic gates and has a negligible area.

1.2. Low power

Since the ShapeShifter utilizes considerably less number of transistors and does not require a core migration facility to transfer state information between cores at a decision point, its power requirements are far less than the power figures reported in the original composite core study. The authors of the CC roughly estimate that the in-order core dissipates three to five times less power than the large out-of-order core. We also assume similar power savings once our ShapeShifter processor switches to the low-power in-order execution mode.

1.3. High prediction accuracy

In the CC, statistical data for instruction-level and memory-level parallelism are gathered with rough estimations for the performance estimation of the passive microengine. Then, functions that are trained by an offline machine-learning method are fed with this inaccurate data. Moreover, the calculated error margins are obtained based on the performance results of the large microengine. In the end, the performance of the passive microengine is calculated with the prediction functions fed by imprecise statistics. Since the prediction results obtained for the large microengine are also used in error calculations as if they are the actual values, no comment can be made about the extent of the actual error.

In our proposed prediction method, we avoid such a multilevel calculation strategy that can create an avalanche effect accumulating small prediction errors into a large one. Our estimation method is based on the ability to simultaneously monitor both in-order and out-of-order instruction selection traffic using the number of ready instructions in the ıssue queue (IQ). The proportion of instantaneous ready instructions that can be issued to arithmetic-logic units for out-of-order and in-order execution methods gives us an accurate guess for the instruction-level parallelism (ILP) currently present in the running application. If the ILP level is high, this ratio becomes large. Otherwise, if the ILP level in the instruction stream is low, the ratio converges to 1, since an out-of-order execution engine cannot outperform an in-order execution engine in such cases.

2. Related work

There are various studies in the literature focusing on energy, power, or area efficiency. One of the main solutions to reduce the energy consumption of the processor is to dynamically turn on and off processor resources according to the running workload [5], [6], [7], [8]. In addition, leading studies in the literature try to select an appropriate core in multicore systems based on the needs of running applications [6], [9], [10]. Another area of research is based on architectural techniques to achieve similar performance figures of out-of-order processors with the help of simpler datapath structures requiring less power and area [2], [11], [12],[13],[14],[15],[16].

The use of heterogeneous cores in multicore systems is another approach that is quite popular today. The most popular studies in the literature aim to put high-performance and low-performance cores together on a single chip [1], [17], [18]. Another approach focuses on the adaptive sharing of resources among these heterogeneous cores [9], [19]. There are also studies that take this approach a step further and fuse adjacent cores to form large out-of-order cores [20], [21].

The composite core study proposed by Lukefahr and his team suggests a heterogeneous core architecture that hosts a high performance and a low power microengine that shares most of the datapath resources [1], [2]. The proposed architecture is based on accurately predicting the performance of the passive microengine after collecting some run-time statistics from the active microengine and running a migration-decision circuit.

Similarly, Afram and his team suggest the FlexCore architecture. The FlexCore is a multicluster architecture with two small cores. These cores can turn into a wide and out-of-order processor when needed. The out-of-order processor can also turn into a simultaneous multi-threading (SMT) core. However, to achieve this configuration, there must be sufficient thread-level parallelism (TLP) between the existing threads. Similar to our proposed method, various run-time statistics are collected to make an accurate estimate of the proper configuration. Cores can also switch to a low-power in-order mode [10].

In another study, Khubaib and his team proposed an architecture based on a large out-of-order core for running single-threaded programs. The processor switches to an out-of-order SMT core to accelerate parallel threads. When the number of running threads exceeds a certain threshold, the algorithm reconstructs the relevant core as a multithreaded out-of-order SMT core. This method takes advantage of the thread-level parallelism (TLP) to improve the performance of the processor [16].

Finally, as in prior work, we proposed the interactive mood detection engine (iMODE) processor with a similar configuration with trial in-order and out-of-order modes to detect the most viable execution mode issued to be next[22]. In that study, there was no prediction logic present, and we only relied on the results of the trial execution modes achieving around 17% power savings. The existence of the trial execution modes in iMODE either unnecessarily degrades performance or misses precious power-savings opportunities. In the ShapeShifter architecture, we get rid of those trial execution modes and make use of an accurate prediction logic, instead.

3. Architecture

The composite core architecture focused in the CC contains two heterogeneous microengines, one small and one large, tightly connected to each other. In that study by Lukefahr and his team, the fetch logic, the branch prediction unit, L1 instruction, and data caches are shared [1]. However, datapath structures that are only utilized by the out-of-order microengine, such as register alias table (RAT), physical register files (PRF), and load/store queue (LSQ), are not shared. In the article version of the CC, which is published later by the same authors, it is seen that the authors also suggest adding a tiny L0 cache to the small microengine [2]. The reactive online controller proposed in the CC relies on a dependency table embedded on the in-order core. The dependency table requires a static random-access memory (SRAM) space almost as large as the size of the re-order buffer (ROB) since it requires 128x32x2 bits.

In our ShapeShifter architecture, there is a single out-of-order microengine that acts like an in-order microengine whenever is needed. The ShapeShifter requires a smaller area with no extra structures for the prediction hardware. Besides, there is no need for any core-migration logic since there is a single core that can switch between execution modes. The mode switch process from the in-order execution mode to the out-of-order execution mode happens almost instantly since the datapath structures holding live instructions can continue their work without any interruption. For instance, the ıssue queue (IQ) that schedules instructions in the program-order can continue its work when the out-of-order scheduler is activated. However, the opposite mode switch (i.e. from the out-of-order execution mode to the in-order execution mode) may require some time to happen. In such a case, the fetch stage must be throttled, and all the datapath structures that currently run instruction in an out-of-order fashion must be drained, first. Then, the in-order execution mode can take place.

3.1. Mode switching logic

At the heart of the ShapeShifter architecture, the mode switching logic makes rational choices between the out-of-order (large and high-performance) and the in-order (small and low-power) execution modes, to meet the high-performance and low-power objectives of the system as much as possible. The decision mechanism is run as often as possible to capture the behavioral changes of the applications as suggested in the CC. The statistics collection process is also run at certain sampling intervals instead of being run in every cycle, and therefore, the additional power dissipation caused by the decision circuit is minimized.

3.2. Statistics

Two processor statistics feed the decision circuit: 1) instruction dispatch ratio, and 2) commit over fetch ratio. At this point, we would like to explain these statistics in detail.

- Instruction dispatch ratio (IDR): The most important feature that distinguishes out-of-order and in-order execution modes is that out-of-order mode can schedule any of the ready instructions waiting in the IQ independent from the original program order, while in-order processors have to schedule them in strict program order. For example, if any of the customers entering a market completes their shopping, they can complete their transaction at the cash register regardless of their order of entry. The logic of out-of-order processors exactly matches this scenario. The shopping process can be simplified when the cash register starts dealing with customers in their order of entry into the market. However, this time, we may need to delay some of the customers who have completed their shopping earlier than the ones who entered the market earlier. The logic of the in-order execution mode exactly matches this second scenario. We can collect the IDR value with the help of a counter used to measure the ILP level in the CC.

  As a result, when we divide the number of instructions that are currently waiting for out-of-order execution by the number of instructions that are waiting for in-order execution, we can calculate the potential speedup of an out-of-order processor over its in-order counterpart. If we perform this sampling often enough (n times) over a period of time, we can obtain a sufficiently precise IDR value as shown in Equation (1) below.

$$IDR = \frac{\sum_{i=0}^{n} Number\ of\ Ready\ Ins.in\ IQ_i}{\sum_{i=0}^{n} Number\ of\ Ready\ Ins.at\ the\ head\ of\ IQ_i} \tag{1}$$

The most important point to consider when collecting samples is that if sampling is done too often, the same instructions that are ready in the same IQ configuration can be repeatedly counted creating a misleading IDR value. For example, we set the sampling interval to 100 cycles during the tests. A slow running application like bwaves from SPEC2006 CPU suite can wait hundreds of cycles without any progress in the IQ. In such a case, we face the danger of counting and accumulating the number of ready instructions we already count at consecutive sampling times. To solve this issue, we remember the program counter (PC) of the instruction waiting at the head of the IQ at a sampling period and compare it with the PC value of the instruction waiting at the head of the IQ at the next sampling period. If these two PC values match, we recognize that the same instruction still waits at the head of the IQ, and there is no change since the previous sampling period.

- Commit over fetch ratio (CFR): The commitment stage is the last step of the instruction completion process. Some of the instructions cannot reach this stage. In particular, in speculative processors with a dynamic branch prediction mechanism, when a branch instruction is mispredicted, many instructions following that branch instruction on the mispredicted path must be discarded to continue with the healthy execution of the application. At this point, we consider the ratio of the number of instructions committed to the number of instructions fetched over a given period, as shown in Equation (2) below. If this ratio is close to 1, we can claim that all instructions that entered the processor were also been successfully committed. However, if this ratio is close to 0, we can conclude that the speculations made by the processor were constantly wrong, and very few instructions could reach the commitment stage. In such a case, the remaining instructions should be removed from the processor. We can easily calculate this ratio with counters that collect statistics multiplied by some constant regression coefficients in the CC.

$$CFR = \frac{Number\ of\ instructions\ that\ commit\ in\ a\ period}{Number\ of\ instructions\ that\ are\ fetched\ in\ a\ period} \tag{2}$$

When running an application, we need the instruction commit over fetch ratio as well as the instruction dispatch ratio to determine how well the out-of-order mode performs compared to its in-order counterpart. Equation (3) shows us that the product of these two ratios can give us the speedup value (S) of the out-of-order mode over the in-order mode.

$$S = IDR \cdot CFR \tag{3}$$

First of all, we do not expect that the speedup value obtained from Equation (3) becomes less than 1, given the obvious performance advantage of the out-of-order mode compared to the in-order mode. However, due to the value of CFR, which indirectly expresses the percentage of instructions inserted into the processor due to mispredictions in the dynamic branch prediction mechanism, the S value can get very close to 0. In such cases, we can see that the out-of-order mode loses its advantage over the in-order mode, and that might be the best time to switch from the out-of-order mode to the in-order mode to save more power. Equation (4) shows the effect of the S value in the mode selection stage of the decision circuit. The decision circuit chooses the out-of-order mode when the calculated S value is greater than a fixed threshold value (α). Otherwise, the in-order mode is selected.

$$mode = \begin{cases} out-of-order, & S > \alpha \\ in-order, & S \leq \alpha \end{cases} \tag{4}$$

At this point, it is clear that low threshold values give us more out-of-order, and high threshold values give us more in-order mode selections. In the tests and results section, we report the effect of this threshold value on performance.

Another option is to calibrate the threshold value to the requirements of applications running at runtime. In this case, it may be possible to prevent performance loss of applications over a certain upper limit and to reveal a more stable and robust decision mechanism. We leave investigating mechanisms to dynamically adapting the threshold at runtime for future work.

3.3. Periodic operation

When collecting statistics, we use two distinct periods, the sampling period (SP) and the decision period (DP). Each DP contains many SPs. As shown in Figure 1, we collect and accumulate the number of ready instructions in the IQ at the end of each SP. At the end of a DP, we calculate both IDR and CFR values and compare the S value with the threshold value. Then, we decide which execution mode is more suitable to run our application until the next DP. If a mode switching decision is made, we continue to work in the target mode.

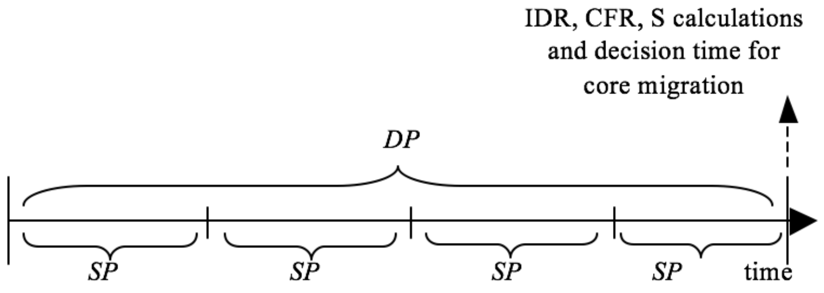

Figure 1. Periodic operation of the ShapeShifter

The DP and the SP parameters play a critical role in the performance of the ShapeShifter architecture. If SP is chosen to be too short, the additional energy consumption caused by the sampling logic outweighs the potential energy savings provided by switching to the in-order execution mode. If the SP is chosen to be too long, the accuracy of the sampling deteriorates too much to provide useful insights on the behavior of the workload.

The concerns regarding DP are similar to SP; choosing a too short DP causes frequent decision points that may cause inaccurate decisions to be made before collecting sufficient information on the behavior of the workload, and may also cause switching overheads in terms of performance. Selecting a too long DP, on the other hand, causes the system to switch infrequently, where the ShapeShifter logic may become late to timely switch execution modes and miss potential energy-saving opportunities.

Achieving optimal performance from ShapeShifter will require the SP and the DP to be selected carefully. In line with our experimental results, we choose DP to be 10K cycles, and SP to be 100 cycles as shown in Table 1.

3.4. Hardware

In the CC, miss and hit rates collected from the L2 cache, the misprediction rate of the dynamic branch prediction mechanism, counters that collect instruction-level and memory-level parallelism levels, multiplier-accumulator (MAC) that performs the performance prediction function trained by the offline machine-learning method, the reactive online controller circuit that also takes into account the error rate, and a dependency table to collect ILP information missing in the in-order processor require extra hardware that takes up about 3% of the total processor area. However, the ShapeShifter architecture proposed in this study consists of a pipelined division circuit utilized in IDR and CFR calculations, a multiplication circuit to calculate the S value, and a comparison circuit to compare it with the threshold value.

Although we assume a hardware solution for the implementation of the ShapeShifter mechanism, we can describe the operation of ShapeShifter in terms of pseudocode, which is presented in Algorithm 1. In the pseudocode, variables depicted as *current commit count* and *current fetch count* represent the commit and fetch counts already collected by many modern processors. Instead of sampling fetch and commit counts in every sampling period (which is more frequent than the decision period), ShapeShifter remembers the fetch and commit counts at the beginning of the previous decision period (represented as *prev fetch* and *prev commit*).

The decision algorithm requires two subtraction and two division operations. In order to reduce the hardware complexity, these operations can be executed back to back, allowing a single subtractor and a single divider to suffice for the algorithm. Essentially, a comparator is a subtractor. Since the subtractions and comparisons are dependent operations, the same unit can be used for both operations. Similarly, this subtractor can be used as an adder for the sampling periods. As a result, the ShapeShifter algorithm can be executed with just a comparator (used for subtraction, comparison, and addition operations in the algorithm), a divider, and a multiplier. It is also worth noting that due to the relatively short sampling and decision periods, these units need not be full 32-bit units, and simpler computation units might be sufficient.

In summary, in the method we proposed, there is neither a MAC circuit nor a dependency table complexity, except for the two statistical values already collected in the CC. Therefore, we can claim that the ShapeShifter is in an advantageous position compared to the circuit in the original operation, both in terms of delay values, area, and power consumption.

4. Experimental methodology

Gem5 simulator is used to test and evaluate the proposed composite core architecture with the ShapeShifter architecture [3]. SPEC CPU2006 applications are compiled in x86 architecture and run with reference input files until each completes 100 million instructions [4]. The properties of the simulated in-order and out-of-order processors are given in Table 1.

Simulation results are evaluated with three criteria: performance, power savings, and energy efficiency. While calculating the energy consumed by the processor per unit time, it is assumed that the out-of-order mode uses four times the energy of the in-order processor, as assumed in the CC. Power expenditure (PE) is calculated as shown in Equation (5) assuming W is the percentage of time spent in the in-order processor. Power savings (PS) reported in the results section is calculated according to Equations (6) and (7) gives the formula of the

Algorithm 1 Operation of ShapeShifter decision mechanism

```
if $current\ cycle$ mod 10000 == 0 then
    $total\_commit \leftarrow current\ commit\ count - prev\_commit$
    $total\_fetch \leftarrow current\ fetch\ count - prev\_fetch$
    $prev\_commit \leftarrow current\ commit\ count$
    $prev\_fetch \leftarrow current\ fetch\ count$
    $IDR \leftarrow total\_ready\ /\ total\_ready\_head$
    $CFR \leftarrow total\_commit\ /\ total\_fetch$
    $S \leftarrow IDR \cdot CFR$
    $total\_ready \leftarrow 0$
    $total\_ready\_head \leftarrow 0$
    if $S > \alpha$ and $current\ execution\ mode\ is$ in-order then
        $Switch\ to$ out-of-order $execution\ mode$
    else
        if $S \leq \alpha$ and $current\ execution\ mode\ is$ out-of-order then
            $Switch\ to$ in-order $execution\ mode$
else
    if current cycle mod 100 == 0 then
        if $current\ PC\ head\ != prev\_pc\_head$ then
            $total\_ready\ += \ number\ of\ ready\ instructions\ in\ IQ$
            $total\_ready\_head\ += \ number\ of\ ready\ instructions\ at\ the\ head\ of\ IQ$
```

Table 1. Processor specifications

Processor microarchitecture	x86
Processor frequency	2 GHz
Machine width	4
Re-order buffer size	192
Issue queue size	64
Load queue/store queue size	32
Physical register file size	256 int, 256 fp
L1 instruction and data cache	4-way LRU, 16 KB each
L2 cache	8-way LRU, 128 KB
Decision period (DP)	10000 cycles
Sampling period (SP)	100 cycles
a Thresholds	1, 3, 5

slowdown rate (SR) of the ShapeShifter processor compared to the baseline out-of-order processor in terms of the average instructions per cycle (IPC) metric. Finally, Equations (8) and (9) express the efficiency formula in terms of energy-delay product (EDP) and energy-delay-square product (ED2P) representation, respectively.

$$PE = 1/4 \cdot W + (1 - W) \tag{5}$$

$$PS = 1 - PE \tag{6}$$

$$SR = \frac{IPC_{baseline}}{IPC_{adaptive}} \tag{7}$$

$$EDP = PE \cdot SR \tag{8}$$

$$ED^2P = PE \cdot SR^2 \tag{9}$$

5. Tests and results

In this section, we evaluate our proposed decision logic in terms of performance, power savings, and energy-delay product savings. The results are collected for three threshold values (1, 3, and 5).

Table 2 shows the percentage of performance drop of applications when running in the in-order mode compared to its out-of-order counterpart. Performance drop percentage in some applications is almost negligible (bwaves 0.3%, leslie3d 1.4%, and sjeng 1.8%). We expect the ShapeShifter processor to prefer the in-order mode when running such applications to save more power. For some other benchmarks, we observe a large percentage of a performance drop (gromacs 68.5%, and gamess 65.6%) while running in the in-order mode. When running such applications, we expect the ShapeShifter to select the out-of-order mode to give the best possible performance.

Table 2. Performance drop of the ın-order mode

Application	**Performance drop %**	**Application**	**Performance drop %**
astar	59.4	h264	62.5
bwaves	0.3	hmmer	65.1
bzip2	16.1	leslie3d	1.4
calculix	50.3	libquantum	17.4
gamess	65.6	mcf	59.7
gems	44.6	milc	50
gobmk	23.3	namd	43.2
gromacs	68.5	sjeng	1.8
		Average	39.3

We also determine the best (or near to best) alpha thresholds that enable the ShapeShifter to limit the performance drop of each benchmark by around 5%. Table 3 shows these threshold values. Excluding the threshold-insensitive benchmarks and large values of the alpha threshold for bwaves, leslie3d, milc, and sjeng benchmarks, the average threshold value is 4.3.

The bar chart in Figure 2 shows the runtime percentages of the in-order mode when running applications on the ShapeShifter processor. As can be seen from the figure, the mechanism we proposed fully meets our expectations. While bwaves, leslie3d, and sjeng applications are running, the in-order mode matches the performance of the out-of-order mode, and, therefore, the in-order execution mode monopolizes the entire run. However, in applications where the performance difference between the two modes is large, the out-of-order mode is preferred, as expected. We also observe that when the threshold value is increased and performance losses can be tolerated more, the in-order mode is given more chance. The last column on each benchmark shows the effect of the near-optimal alpha threshold (best alpha) value retrieved from Table 3. As seen from the graph, the results for the fixed alpha threshold of 3 and the best alpha threshold that always gives us around a 5% performance penalty are very close to each other. Definitely, with the fixed threshold value of 3,

Table 3. Near-optimal thresholds

Application	α	**Application**	α
astar	7.0	h264	4.0
bwaves	insensitive	hmmer	7.0
bzip2	5.0	leslie3d	insensitive
calculix	3.0	libquantum	4.5
gamess	3.0	mcf	3.7
gems	2.5	milc	35.0
gobmk	3.0	namd	3.6
gromacs	5.0	sjeng	insensitive

we lose some power-saving opportunities in astar, bzip2, gromacs, h264, hmmer, libquantum, mcf, milc, and namd benchmarks. On the contrary, in gems benchmark, we give more power-saving opportunities in the fixed threshold case, and we pay for its large performance impact as shown in Figure 3.

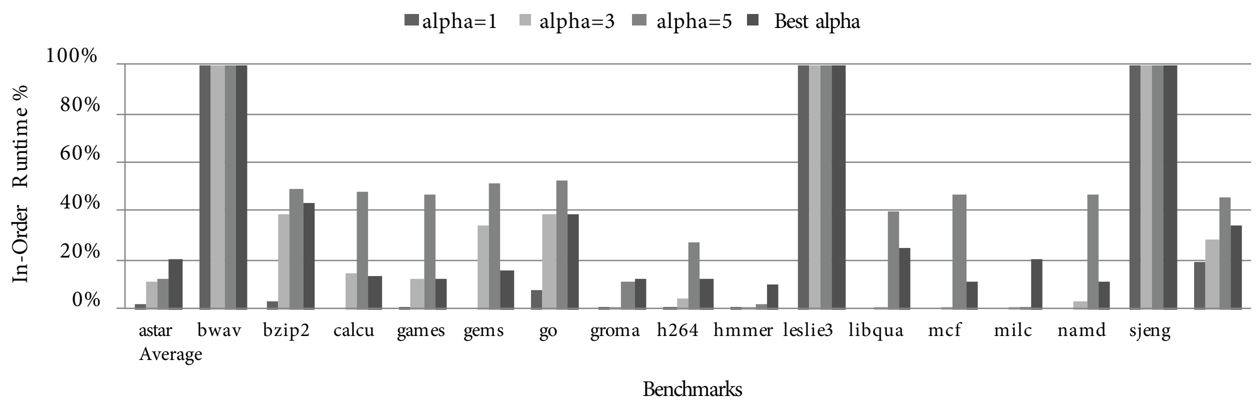

Figure 2. Runtime percentage of the in-order mode in ShapeShifter for various alpha thresholds. The best alpha represents the best performing alpha value for each benchmark, respectively.

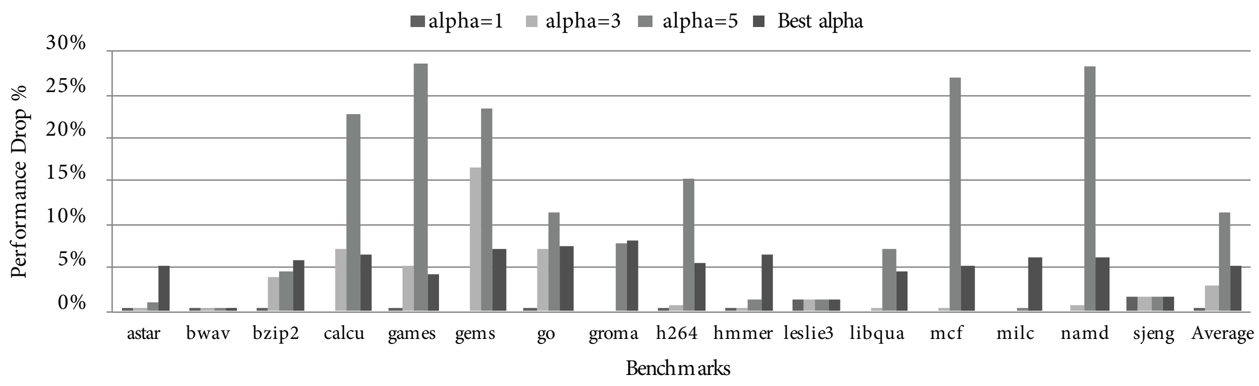

Figure 3. Percentage of performance drop of ShapeShifter compared to baseline out-of-order processor for various alpha thresholds. The best alpha represents the best performing alpha value for each benchmark, respectively.

Figure 3 shows the degree of performance loss for three fixed threshold values and the near-optimal threshold (best alpha) that we study. Here, we see that if the threshold value is selected as 1, the performance loss of the system is only 0.25% on average and the highest performance loss is 1.8% in sjeng benchmark. However, this threshold value causes many power-saving opportunities to be missed. As can be seen from the chart, the performance loss tolerance of applications to the in-order mode varies greatly. For example, bzip2 application spends almost 50% of its entire time in the in-order mode, but we observe only a 0.12% performance drop compared to the baseline out-of-order processor. With the fixed threshold of 3, calculix shows a serious loss of performance over 7%, although it only runs on the in-order mode 14% of its overall runtime. Moreover, some applications, such as milc and hmmer, are almost insensitive to studied threshold values and prefer the out-of-order mode at all times. In summary, the ShapeShifter processor senses that the loss of performance will be high if we switch to the in-order mode, and, in such cases, it keeps selecting the out-of-order mode to always stay on the safe side. The only exceptional case is encountered with the gems benchmark. The fixed alpha threshold of 3 seems to be a little too much for this benchmark (the best alpha value is 2.5 as shown in Table 3), and we observe more than a 15% performance drop. Finally, the ShapeShifter also identifies slow benchmarks that are totally insensitive to any alpha threshold (i.e. bwaves, leslie3d, and sjeng) and runs them in in-order mode 100% of the time.

Figures 4 and 5 report the results of power savings and energy-delay product (EDP) savings, respectively. As can be seen from the figures, while the threshold value is 1, the average power and EDP savings are almost 15%. However, when the threshold value is selected as 3, we observe that average power savings suddenly inclines to 21.5% although the average performance loss stays around 2.7%. The same observation is true for the EDP criterion, as well. When the threshold value is increased from 1 to 3, the average EDP savings increases to 19.1%. Note that these results are very close to the results of the near-optimal alpha threshold that is empirically found for each benchmark. Of course, the average power and EDP savings for the alpha threshold of 5 are the highest. However, this threshold value comes with a cost of 11.4% average performance penalty, which might still be tolerable in battery-operated devices, such as laptops and smartphones.

Figure 6 reports the results of energy-delay-square product (ED^{2}P) savings. This metric is less latency-tolerant, and it is more appropriate to apply this to desktop processors. As can be seen from the figure, ED^{2}P savings becomes worst when the alpha threshold goes up. In many benchmarks (i.e. calculix, games, gems, gromacs, h264, hmmer, mcf, and namd) we have negative savings meaning it is not worth saving power since the performance penalty that we get is not tolerable in these benchmarks. In the end, the best results are obtained from the fixed alpha threshold of 3 and the near-optimal threshold configurations with more than 16% savings, on average. As it is expected, the savings of the fixed threshold of 5 are worst, since it is the configuration with the highest average performance penalty.

The simulation results also show that applications come into two main categories when their performance loss and potentials for power savings are considered. The first category of applications requires high performance and they heavily rely on the out-of-order mode of execution. In these applications, there is a big trade-off between power savings and loss of performance. As a result, it is not possible to achieve remarkable power savings in this first group of applications without tolerating a considerable loss of performance. The second category of applications, on the other hand, shows a limited ILP. These applications are suitable to use the in-order mode in those low ILP program phases and are good candidates for providing significant power savings in exchange for a very limited performance loss. When the percentage of performance drop of an application is less than 30% between the out-of-order and the in-order mode, we assume that such an application is in this second category

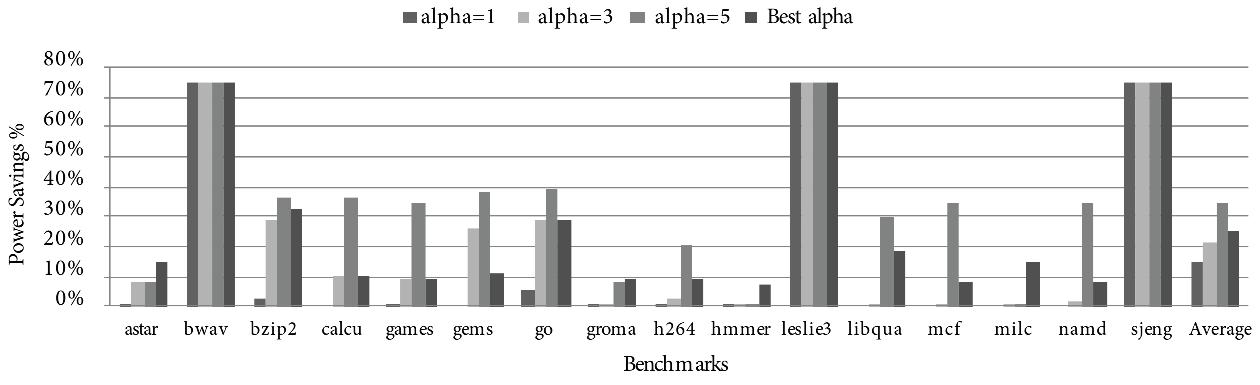

Figure 4. Percentage of power savings of ShapeShifter compared to baseline out-of-order processor for various alpha thresholds. The best alpha represents the best performing alpha value for each benchmark, respectively.

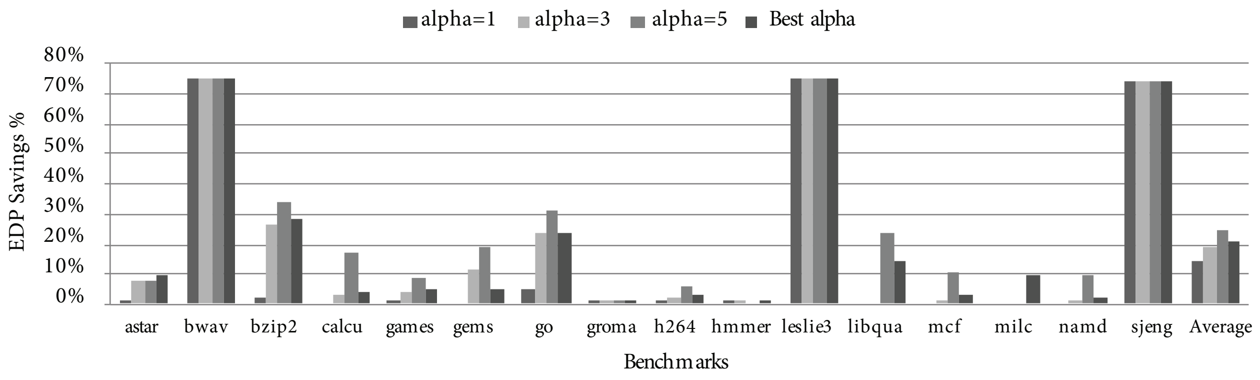

Figure 5. Percentage of EDP savings of ShapeShifter compared to baseline out-of-order processor for various alpha thresholds. The best alpha represents the best performing alpha value for each benchmark, respectively.

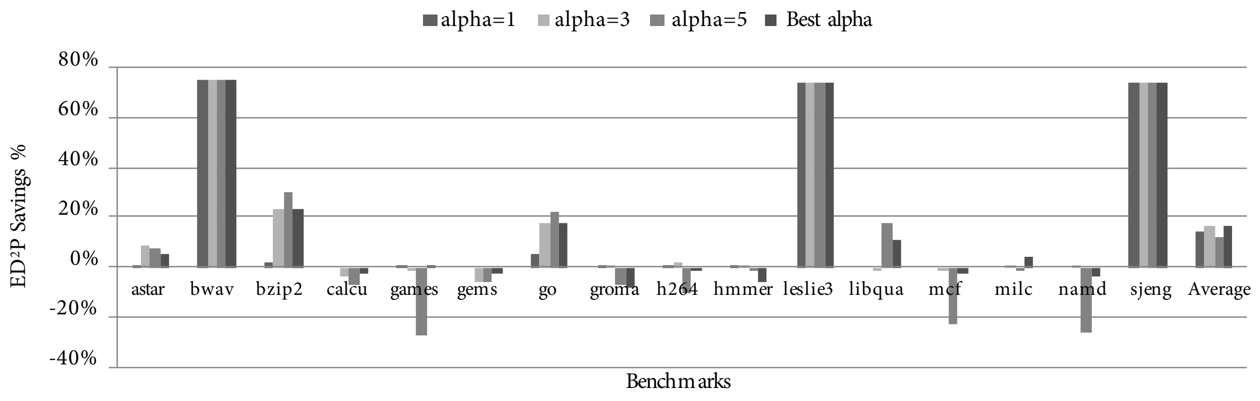

Figure 6. Percentage of ED^2P savings of ShapeShifter compared to baseline out-of-order processor for various alpha thresholds. The best alpha represents the best performing alpha value for each benchmark, respectively.

of applications. Applications bwaves, bzip2, gobmk, leslie3d, libquantum, and sjeng are in this category.

The performance loss, power savings, and energy-delay product savings results are obtained when the applications from both categories are evaluated together. However, if we focus on the second group of applications, for the fixed threshold value of 3, the average performance loss is as low as 2.4%, the average power savings is 47.2%, the average EDP savings is 45.7%, and the average ED^2P savings reaches 44%. If we move to the threshold value of 5, the average performance loss inclines to 4.4%, the average power savings becomes 55%, the average EDP savings rise to 52.2%, and, finally, the average ED2P savings goes beyond 49%.

6. Conclusion

In this study, we propose the ShapeShifter architecture, which is an out-of-order superscalar processor that sometimes acts as if it is a smaller in-order superscalar processor. At certain decision points, the mode switching circuit decides which execution mode is more suitable for running an application until the next decision point. The ShapeShifter achieves better power and energy-delay savings with the help of just two processor statistics compared to the relatively complex and power-hungry control circuit within a composite core architecture that is previously proposed in the literature. Average performance loss of the ShapeShifter compared to a fully out-of-order baseline processor is kept below 5% while saving more than 25% of overall processor power across all simulated SPEC CPU benchmarks. This result is better than the average power savings of 18% of the composite cores architecture and the average power savings of the iMODE study, which is reported as less than 20%, for similar performance drop levels.

In future work, we are planning to study the effect of resource partitioning for achieving further power, EDP, and ED^2P savings on the ShapeShifter architecture.

Acknowledgments

This work is supported by the Scientific and Technical Research Council of Turkey (TUBITAK) under Grant No. 117E866.

References

[1] Lukefahr A, Padmanabha S, Das R, Sleiman FM, Dreslinski RG et al. Composite cores: Pushing heterogeneity into a core. In: 45th Annual IEEE/ACM International Symposium on Microarchitecture; Vancouver, BC, Canada; 2012. pp. 317-328.

[2] Lukefahr A, Padmanabha S, Das R, Sleiman FM, Dreslinski RG et al. Exploring fine-grained heterogeneity with composite cores. IEEE Transactions on Computers 2015; 65 (2): 535-547.

[3] Binkert N, Beckmann B, Black G, Reinhardt SK, Saidi A et al. The gem5 simulator. SIGARCH Computer Architecture News 2011; (39):1-7.

[4] Henning, JL. SPEC CPU2006 benchmark descriptions. SIGARCH Computer Architecture News 2006; (34):1-17.

[5] Ghiasi S, Casmira J, Grunwald D. Using IPC variation in workloads with externally specified rates to reduce power consumption. In: Workshop on Complexity Effective Design, Vancouver, BC, Canada, 2000.

[6] Albonesi DH, Balasubramonian R, Dropsbo SG, Dwarkadas S, Friedman EG et al. Dynamically tuning processor resources with adaptive processing. Computer 2013;36 (12): 49-58.

[7] Manne S, Klauser A, Grunwald D. Pipeline gating: speculation control for energy reduction In: Proceedings of the 25th Annual International Symposium on Computer Architecture; Barcelona, Spain; 1998. pp. 132-141.

[8] Morancho E, Llaberia JM, Olive A. On reducing energy-consumption by late-inserting instructions into the issue queue. In: Proceedings of the 2007 International Symposium on Low power Electronics and Design; Portland, OR, USA; 2007. pp. 371-374.

[9] Homayoun H, Kontorinis V, Shayan A, Lin T, Tullsen DM. Dynamically heterogeneous cores through 3D resource pooling. In: IEEE International Symposium on High-Performance Computer Architecture; New Orleans, LA, USA; 2012. pp. 1-12.

[10] Afram F, Ghose K. FlexCore: A reconfigurable processor supporting flexible, dynamic morphing. In: Proceedings of the IEEE 22nd International Conference on High Performance Computing (HiPC); Bengaluru, India; 2015. pp. 30-39.

[11] Sembrant A, Carlson T, Hagersten E, Black-Shaffer D, Perais A et al. Long term parking (LTP): criticality-aware resource allocation in OOO processors. In: Proceedings of the 48th International Symposium on Microarchitecture; Waikiki Hawaii; 2015.pp. 334-346.

[12] Sleiman FM, Wenisch TF. Efficiently scaling out-of-order cores for simultaneous multithreading. SIGARCH Computer Architecture 2016;44:431-443.

[13] Lebeck AR, Koppanalil J, Li T, Patwardhan J, Rotenberg E. A large, fast instruction window for tolerating cache misses. In: Proceedings 29th Annual International Symposium on Computer Architecture; Anchorage, Alaska, USA; 2002. pp. 59-70.

[14] Hubner M, Tradowsky C, Gohringer D, Braun L, Thoma F et al. Dynamic processor reconfiguration. In: International Conference on Reconfigurable Computing and FPGAs; Cancun, Mexico; 2011. pp.123-128.

[15] Carlson TE, Heirman W, Allam O, Kaxiras S, Eeckhout L. The load slice core microarchitecture. In: ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA); Portland, Oregon; 2015. pp. 272-284.

[16] Suleman MA, Hashemi M, Wilkerson C, Patt YN. Morphcore: An energy-efficient microarchitecture for high performance ilp and high throughput tlp. In: 45th Annual IEEE/ACM International Symposium on Microarchitecture; Vancouver, BC, Canada; 2012. pp. 305-316.

[17] Kumar R, Tullsen DM, Ranganathan P, Jouppi NP, Farkas KI. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In: Proceedings 31st Annual International Symposium on Computer Architecture; Munich, Germany; 2004. pp. 64-75.

[18] Bahar RI, Manne S. Power and energy reduction via pipeline balancing. In: Proceedings of the 28th Annual International Symposium on Computer Architecture; Göteborg, Sweden; 2001. pp. 218-229.

[19] Kumar R, Jouppi NP, Tullsen DM. Conjoined-core chip multiprocessing. In: Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture; Cambridge, UK; 2014. pp. 195-206.

[20] Kim C, Sethumadhavan S, Govindan MS, Ranganathan N, Gulati D et al. Composable lightweight processors. In: 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO); Chicago, IL, USA; 2007. pp. 381-394.

[21] Ipek E, Kirman M, Kirman N, Martinez JF. Core fusion: accommodating software diversity in chip multiprocessors. SIGARCH Computer Architecture News 2007;35:186-197

[22] Savas ME, Guney IA, Tokatli NN, Kisinbay B, Kucuk G. iMODE (interactive mood detection engine) processor. In: 4th International Conference on Computer Science and Engineering (UBMK); Samsun, Turkey; 2019. pp. 1-6.