IMODE INTERACTIVE MOOD DETECTION ENGINE

by

Muhammed Emin Savaş

Submitted to Graduate School of Natural and Applied Sciences
in Partial Fulfillment of the Requirements
for the Degree of Master of Science in
Computer Engineering

Yeditepe University
2019

IMODE INTERACTIVE MOOD DETECTION ENGINE

APPROVED BY:

Assoc. Prof. Dr. Gürhan Küçük (Thesis Supervisor) (Yeditepe University)	.........[illegible].........
Prof. Dr. Sezer Gören Uğurdağ (Yeditepe University)	.........[illegible].........
Assist. Prof. Dr. Alp Arslan Bayrakçı (Gebze Technical University)	.........[illegible].........

DATE OF APPROVAL: ..../..../2019

ACKNOWLEDGEMENTS

I am heartily thankful to my supervisor, Assoc. Professor Gürhan Küçük, whose encouragement, guidance and support from the initial to the final level enabled me to write this thesis report.

I am also thankful to İsa Ahmet Güney for his helps.

Lastly, I offer my regards and blessings to all of those who supported me in any respect during the completion of the project.

ABSTRACT

IMODE INTERACTIVE MOOD DETECTION ENGINE

Applications change their mood from time to time. A memory-intensive application does not need to be always memory-intensive from its first instruction to its last. Similarly, it is highly usual that a computation-intensive application generates heavy memory traffic at certain points of its entire run. Meanwhile, processors are designed to have a fixed resource configuration, which is expected to serve all kinds of applications with all kinds of program phases. In this study, we propose a new processor, which tracks down instant mood changes of running applications and applies immediate processor mode changes between in-order and out-of-order modes for either saving power or keeping up with the high-performance demands of applications. In our experiments, we observe that the proposed processor can really track the mood changes of applications, accurately, and achieves 17 percent power savings with only less than 1 percent of performance drop, on the average across all simulated benchmarks.

ÖZET

IMODE İNTERAKTİF MİZAÇ BELİRLEME MOTORU

Uygulamalar çalışma zamanları içerisinde mizaçlarını zaman zaman değiştirirler. Bellek-yoğunluklu bir uygulama baştan sona bütün komutlarında bellek-yoğunluklu çalışmaz. Yine benzer bir şekilde, yüksek hesaplama-yoğunluklu bir uygulamanın çalışma zamanının bir döneminde yüksek bellek-yoğunluklu bir trafik oluşturduğunu görürüz. Fakat buna rağmen işlemcilerin sabit kaynak dizilimleri ile her türden uygulamaya her türlü çalışma fazında hizmet vermeye çalıştığını görmekteyiz. Biz, bu çalışmada uygulamaların anlık mizaç değişikliklerini algılayan ve buna göre hızlı bir şekilde işlemcinin modunu yüksek performansı yakalayabilmek adına sırasız çalışma ve güç tasarrufu yapabilmek adına sıralı çalışma modları arasında değiştirebilen bir işlemci tasarımı sunuyoruz. Yaptığımız deneylerde, sunduğumuz işlemcinin gerçekten uygulamaların mizaç değişimlerini doğru bir şekilde takip ettiğini gözlemlemekte ve benzetim edilen denek uygulamalarda sadece yüzde 1'in altında performans kaybı ile ortalama yüzde 17 güç tasarrufu elde edildiği görülmektedir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS/ABBREVIATIONS

CPU	Central processing unit
ILP	Instruction level parallelism
iMODE	Interactive mood detection engine
IO	In-order
IQ	Issue queue
IPC	Instruction per cycle
ISA	Instruction set architecture
IWB	Instruction window buffer
LQ	Load queue
MLP	Memory level parallelism
OoO	Out-of-order
OS	Operating system
PE	Postponed number of epochs
PT	Postpone threshold
RF	Register file
ROB	Re-order buffer
SMT	Simultaneous multi-threaded
SQ	Store queue
TLP	Thread level parallelism
QoS	Quality of service

1. INTRODUCTION

Processors are general-purpose hardware interpreters, which are responsible for running various types of applications from games to any type of mission-critical software. For a long time, the speed was a processor's major and only concern when running applications. Nowadays, power-related concerns even surpassed all performance-related concerns on any processor that can be encountered on any type of system. Obviously, a power-aware processor plays an important role in improving the lifetime of many modern technologies (e.g. smartphones, laptops). Somewhat less obviously, a power-aware processor also plays an important role in the success of a real-time, mission-critical system that explores space and other planets. Besides, by 2025, it is predicted that data centers of the world will consume one-fifth of Earth's power, and a power-aware processor may provide enormous power savings when it is deployed on data centers in large quantities.

The well-accepted philosophy on today's processor design is still ”one-size-fits-all” kind of realization with a fixed set of datapath structures and a fixed mode of execution. For instance, when an application has a program phase with a low Instruction Level Parallelism (ILP) degree runs on a 4-way superscalar, aggressively speculative, out-of-order (OoO) processor, most of the processor resources are underutilized and all power saving opportunities are lost. On the contrary, when an application, which has a program phase with a high ILP degree, runs on a 2-way superscalar in-order (IO) processor, it receives a huge performance penalty due to insufficient processor resources. Both of these scenarios point out the inefficiency of the current fixed-mode processors. In this study, we propose a processor that can track down the instant mood patterns of running applications and apply appropriate mode changes (i.e. either from IO to OoO or from OoO to IO) for saving power without sacrificing too much processor performance.

The idea of having heterogeneous cores within a processor is not new. ARM's power-aware big.LITTLE architecture is one of the best examples of such realizations, and it requires a physical area to keep multiple cores on the same chip. This is also a valid approach in almost all heterogeneous architectures that are proposed in the literature, and, hence, 2 substantial area increase is inevitable [1, 2, 3, 4, 5]. However, our proposed design deviates from this traditional approach. We propose a specialized OoO core that can act either as a traditional OoO core or an IO core whenever it is suitable. As a result, the area requirement of our proposed processor is almost identical to the area requirement of a traditional OoO processor.

One of the major advantages of our approach, which we call interactive MOod Detection Engine or iMODE, in short, is its minimal cost on thread migration. In a heterogeneous core, when a thread is needed to be migrated from one core to another, substantial amount of data traffic has to be generated. This also results in substantial amount of power dissipation overhead in current state of the art. However, in our design, the source and the target cores are physically the same core, and we only change the execution mode of the core. Thus, during a thread migration in iMODE, all datapath structures holding instructions and processor state can still keep their content, and, no migration cost in terms of latency and power is involved.

To better motivate our study, we would like to show how accurately iMODE tracks down the mood of a running application. The x-axis of the chart given in Figure 1.1 represents the number of instructions that are executed from the SPEC2006 gobmk benchmark, in million-instruction granularity. After one million instructions are executed within a period, we calculate the average Instruction Per Cycle (IPC) for that specific period. The y-axis in the chart plots those corresponding IPC values for each million-instruction period for three distinct designs: OoO, IO and iMODE. In the early execution phase of gobmk benchmark, OoO and IO modes go nearly head to head, giving similar IPC values. As a result, the iMODE processor switches to and stays in IO mode for more than 290 epochs (290 million cycles and around 34 million committed instructions), since there is no performance gain running in OoO mode within that long time period. In our current design, each epoch is empirically chosen to be one-million cycle long. After executing 34 million instructions, the benchmark shifts into a new program phase, where the OoO mode can give a much better performance than the IO mode. During that phase change, iMODE processor moves into the OoO mode and starts tracking down the IPC of the OoO processor. Note that there is a slight performance drop in iMODE processor compared to the original OoO processor due to the existence of short trial periods that iMODE tests the performance of the IO mode from time to time.

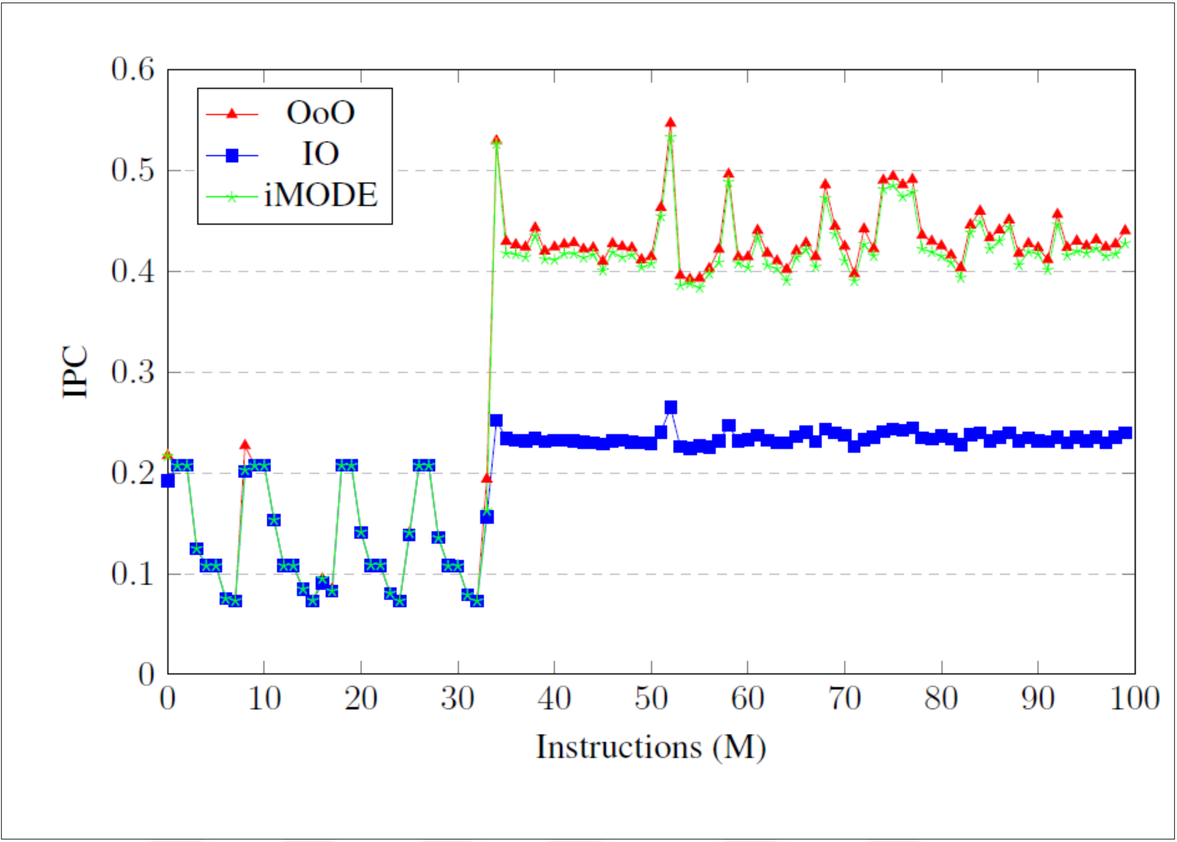

Figure 1.1. Performance trace of gobmk benchmark on OoO, IO, and iMODE

As a result, iMODE gives improvements in three major metrics:

- Power: As the above example clearly shows, iMODE can stay in IO execution mode for long epochs. During the IO mode, structures that are used for guaranteeing correct OoO execution (i.e. register renaming mechanism, load queue, reorder buffer and physical register files) can be turned off. This makes iMODE a very low-power processor as long as the IO execution mode is in action. However, when the mood of the running application asks for an OoO mode, the processor goes into a performance mode, in which iMODE dissipates a considerably larger amount of power.
- Cost of Thread Migration: One of the important features of heterogeneous cores is thread migration. These processors allow a running thread to migrate to a suitable core so that they either save power or keep an application's performance at its peak. However, thread migration takes a considerable amount of time, especially when the source and target cores are physically away from each other. The state of the

source core should be transferred to the target core, and larger state data means higher migration latency and higher power cost. In computer architecture research, data transfer is one of the major challenges, which should be minimized at all costs. For this exact reason Intel moved from separate physical and architectural register structures in Pentium III to a combined register file that holds both physical and architectural registers in one structure in Pentium 4. In iMODE design, we follow the same strategy, and we make both source and target core the same core. Thus, during a thread migration all datapath structures holding instructions and processor state can still keep their content, and, no migration cost in terms of latency and power is involved.

- Area: This is one of the strongest fields of the iMODE design. Processors with heterogeneous cores physically hold multiple cores on the same chip. When there are multiple active threads utilized to each of these heterogeneous cores, there seems to be no problem. However, when only one core is active and the rest of the cores are passive at certain times [1], the area efficiency of the processor is immediately questioned. iMODE is actually an OoO processor, which sometimes acts as an IO processor. Therefore, its area requirement is almost identical to a traditional OoO processor.

2. BACKGROUND

2.1. IN-ORDER AND OUT-OF-ORDER PROCESSORS

Instructions are fetched and executed in compiler-generated program order in IO processors. On the other hand, in OoO processors, while instructions are fetched and decoded in program order, they are executed in arbitrary order. This enables OoO processors not to stall every time they encounter an instruction that is not ready for execution. In IO processors, all in-flight instructions wait for the execution of the oldest instruction, and if that instruction stalls, the whole processor pipeline stalls. OoO processors, on the other hand, avoid this problem by introducing an OoO scheduler that can select a ready instruction ignoring its program order. In this new selection mechanism, even the youngest instruction can have a chance for execution before the rest of the older instructions as long as it has no data dependency. In an n-way superscalar processor, n instructions can be fetched, decoded, scheduled and executed in a given clock cycle. The number of instructions that can be executed in parallel is directly proportional to a term, which is known as the Instruction Level Parallelism (ILP). When the ILP degree is high, it means that there are a number of instructions present in the Instruction Queue (IQ) that are ready for execution. The main purpose of OoO processors is to extract this ILP hidden in instruction streams and achieve better performance figures compared to IO processors. The biggest handicap of this case is there can be data, control and structural hazards among instructions that reside in the IQ. The performance gap between an IO and OoO mode processor shrinks when the number of these pipeline hazards increase within a processor. In such a case, running an OoO processor becomes meaningless, since an IO processor can achieve a similar performance trend. However, when those pipeline hazards rarely exist, the OoO processor can easily provide two to three-fold speedup compared to an IO processor, and, typically, this type of a performance improvement is too large to be ignored by any kind of application scenarios.

When looking from the structural hazard point of view, both IO and OoO processors can experience structural hazards due to insufficient number of functional units. However, from the data hazard point of view, OoO processors exploit the possibility of executing independent instructions in parallel, and, hence, they perform better compared to IO processors. Today, OoO processors meet the demands of high performance applications. However, as mentioned in Chapter 1, OoO processors spend more power to achieve higher performance figures, while IO processors could be more power efficient.

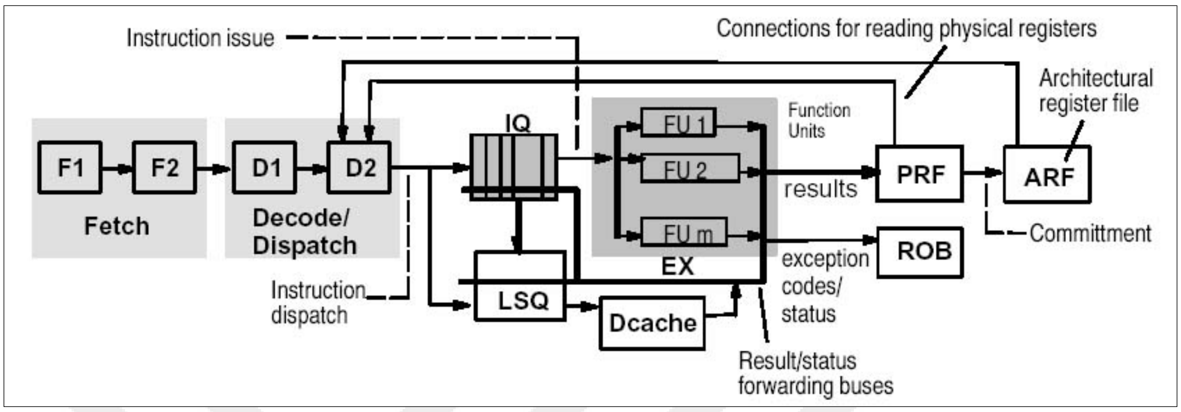

Figure2.1. Example datapath figure from an OoO processor

Nowadays, modern processors aiming high performance almost always use OoO superscalar architectures. A typical OoO processor contains seven fundamental pipeline stages as shown in Figure 2.1. Although, an IO processor has nearly identical implementation stages, some of the stages might be simplified or even excluded (such as commitment stage, at the end).

- Fetch Stage: The mechanism responsible for receiving instructions from the instruction cache or instruction memory. According to the superscalar width (say, n) of the processor, the number of instructions, which can be fetched in a single clock cycle, can change from 1 to n. Fetch stage is the first stage of the in-order front-end of a superscalar processor, and all instruction are fetched in program order. Instructions are inserted into a queue structure, which is known as the Fetch Queue (FQ). The size of the FQ is usually selected as twice the size of the superscalar width, so that, in a single clock cycle, half of the queue is updated with incoming instructions, meanwhile the instructions from the other half are read by the next pipeline stage, which is known as the Decode stage. In the next clock cycle, partitions of the FQ are exchanged, and the half that is drained by the decode stage is updated by new incoming instructions by the fetch stage, meanwhile the other half is consumed by the decode stage. This exchange process continues as long as the processor is up and running. As shown in the Figure, this stage is usually divided into two sub-stages to reduce the clock cycle time, and, hence, to reduce the clock frequency.

- Decode Stage: In this instruction interpretation stage, which is represented as the box labeled D1 in the Figure, various critical information from the opcode to source and destination registers and to immediate data values of each instruction are decoded. The first stage of the register renaming process also takes place within this stage. The physical register mappings of the source registers of the decoded instruction are looked up from a structure, which is known as the Rename Table (RT). Intel uses the Register Alias Table (RAT) name for the same structure. Since an n-way superscalar processor can decode up to n instructions within a single clock cycle, the decode stage also utilizes a circuitry named Dependency Checking Logic (DCL). In the DCL logic, there are comparators for detecting match conditions among destination registers of older instructions and source registers of younger instructions. When the DCL logic detects a match between a destination register of instruction I and a source register of instruction J, the processor learns that there is a Read After Write (RAW) data dependency between instruction I and instruction J.
- Dispatch Stage: This stage is responsible for inserting decoded instructions into a structure, which is known as the Instruction Dispatch Buffer (IDB) in OoO processors. This structure is formerly named as the Instruction Queue (IQ) in IO processors, and this name is also still widely used in OoO processors. Normally, in IO processors when all of the source registers of an instruction become valid and the instruction is among the oldest n instructions within the IQ, then it can be sent to its corresponding functional unit. As a result, if an instruction, which is from the set of the oldest n instructions, is not ready, the IO processor stalls at the dispatch stage. Unlike IO processors, in OoO processors a ready instruction can be sent to its functional unit in an arbitrary order for extracting ILP among instructions that reside in the IQ. Some researchers also call IQ as the Instruction Window Buffer (IWB), since degree of ILP extraction is actually limited by the window size of the IQ.

  There are two control circuitry that are directly related to OoO instruction scheduling in the dispatch stage: 1) the wake-up and 2) the selection Logic. First one checks the true dependencies between instructions, and is responsible for waking up the instructions that become ready for execution. When two source registers of an instruction becomes valid, the instruction becomes ready and it is woken up by the wake-up circuitry. The selection logic is responsible for selecting and issuing (transferring an instruction for execution) n ready instructions in an n-way superscalar processor.

Again in this stage, all instructions are inserted into a circular-queue structure known as Re-Order Buffer (ROB). Moreover, all load and store instructions are written into the Load Queue (LQ) and the Store Queue (SQ) according to their program order. In some architectures these queue structures are combined into a single structure, which is known as the Load/Store Queue (LSQ).

Finally, the last part of the register renaming algorithm, where new physical register mapping is set for each instruction with a destination register, is also run in this stage.

- Execute Stage: This stage is responsible for executing ready instructions, and forwarding their results back to awaiting instructions in the IQ. Generally, there are multiple functional units for serving multiple instructions in parallel. The bypass (or forwarding) circuitry is responsible for transferring results of functional units to the inputs of all functional units in case there is a possibility of back-to-back execution of dependent instructions.
- Memory Stage: This is a stage with a non-deterministic latency. When there is a store instruction, the data of the store instruction is immediately written to the LSQ structure, and data is also forwarded to the first level cache. The lower level caches and the memory is updated depending on the write mode of the cache levels. When the cache is a write-through mode cache, it immediately transfers the written data to the lower level memory structures. However, this mode generates enormous amount of cache traffic and dissipates enormous amount of power. On a write-through cache, though, an extra dirty bit is attached to each cache line. When there is a store instruction, the dirty bit of the corresponding cache line is set to indicate that the data in that cache line is not synchronized with the memory. The data transfer is realized only when that cache line is evicted from the cache and its dirty bit is set.

In OoO processors, the LSQ structure is accessed when there is a possibility of a load bypass. The load bypass can occur, when a younger load instruction accesses memory before an earlier load instruction. The LSQ holds address information for each memory instruction, and, therefore, such controls are viable. Since, there is no possibility for a load bypass in IO processors, there is no need for keeping the complete LSQ structure in those processors.

Another important feature of the LSQ structure is known as store data forwarding. When the address of a younger load instruction overlaps with the address of an older store instruction, the load instruction can receive its data from the data field of the store instruction within the LSQ. This kind of a data forwarding is the fastest memory operation, and it may take as fast as a single clock cycle. Today, even the fastest first level cache access can be at least 3 or 4 cycles long.

- Writeback Stage: This is the stage where the speculative results are written to speculative physical register files. At this stage, the ready bit of the corresponding instruction within the ROB is also set to indicate that this instruction already completed its execution and is ready for exiting the pipeline. In OoO processors, the writeback stage cannot be the last stage of execution, since the OoO completion makes implementation of the precise interrupt mechanism quite complicated. Therefore, and extra in-order backend stage for completion is usually takes place. On the other hand, in IO processors, the in-order execution and in-order writeback makes the use of ROB totally meaningless.
- Retirement (or Commitment) Stage: This last stage utilizes a circular queue structure known as the Re-Order Buffer (ROB). The head of the ROB holds the oldest instruction within the processor, meanwhile, the tail of the ROB holds the youngest instruction. The retirement stage, enables completion of instructions in program order. When an instruction reaches the head of the ROB and when it’s ready for retirement bit is set by the writeback stage, then, the instruction becomes free to exit the processor. During the exit, the speculative data value stored in a corresponding PRF entry is transferred to its corresponding precise Architectural Register File (ARF), and the Rename Table is updated to indicate this update. The old physical register mapping of the instruction is also deallocated, again, in this stage.

In an n-way superscalar processor, n number of oldest instructions, which reside in the head of the ROB, can retire at the same cycle.

2.2. RELATED WORK

There is much prior work in the literature which strives for either better energy/power efficiency or die area efficiency. A straightforward solution to reduce energy consumption of the processor is to dynamically disable processor resources according to runtime needs of workloads [13, 15, 16, 19]. Additionally, some prior work proposed dynamic migration among out-of-order high performance cores and in-order low energy cores based on the nature and demands of running applications [1, 14, 22]. Another research direction is storing certain instructions in simpler datapath structures, achieving similar performance with less power and area compared to expensive out-of-order resources (or exploiting these leftover resources for higher performance) [17, 18, 20, 21, 24, 26].

A popular approach is to use a heterogeneous configuration in multicore processors, i.e. processors which contain both high-performance power-hungry cores and low-performance power-saving cores together in one chip [6, 12, 25]. One particular approach is to achieve heterogeneity among cores not by design, but dynamically by sharing resources among cores [8, 9]. Taking this approach one step further, some prior research proposed a reconfigurable architecture that enables the formation of wider-issue out-of-order cores by fusion of neighboring small cores [10, 11].

Lukefahr et al. propose a heterogeneous core architecture composed of a high-performance core (big Engine) and an energy efficient core (small Engine) sharing most resources such as L1 caches and the branch prediction unit. In their design, there is only one Engine active at a time, and the execution switches dynamically between Engines to adapt to application characteristics. The proposed architecture collects a few runtime statistics on the active core and tries to accurately predict the performance of the inactive core for the migration decision [1].

Afram et al. propose FlexCore, a multi-clustered architecture with two narrow cores, which can fuse into a wide OoO core whenever it is needed. The OoO core can also turn into a Simultaneous Multi-Threaded (SMT) core if there is enough TLP among running threads.

Again, various runtime statistics are collected to make accurate predictions for the mode switch operation. Cores in FlexCore can also turn into a low-power IO mode [4].

Finally, Khubaib et al. propose an architecture which uses a large OoO core as a base to run single-threaded programs. The processor switches to IO SMT core mode to accelerate parallel threads. When the number of active threads exceeds a certain threshold, their technique reconfigures the core into a highly-threaded IO SMT core, which exploits Thread Level Parallelism (TLP) and provides high throughput by hiding long latency operations of individual threads [22].

3. DESIGN AND IMPLEMENTATION

We know that the design of the OoO processors is based on the exploitation of the instructions which can be executed in parallel, and the more instructions that can be run independently, the higher the ILP. However, applications do not perform with high ILP from their beginning until their completion. The high rate of true data dependencies, memory latencies, and input/output operations are the main reasons for ILP drops. Whenever an application enters a low ILP phase, the processor cannot take advantage of the complex and power-consuming OoO logic. The heart of the iMODE processor based upon this observation. In such phases where ILP is low, running in OoO mode would result in unnecessary power consumption. To solve this problem and save power, our iMODE processor can switch between OoO and IO execution modes without losing too much performance. In this study, we assume that a 5 percent performance drop is the maximum level tolerable for the general purpose processor users.

Switching between the two execution modes contains two main mechanisms in our iMODE processors. The first mechanism is the Decision, and the second mechanism is the Enforcement. In the decision mechanism, we decide whether to stay or not to stay in current mode while in the Enforcement Mechanism we are implementing necessary changes on the processor. These mechanisms are discussed in the following sections in detail.

3.1. DECISION MECHANISM

A useful decision mechanism should successfully determine the suitable time to switch OoO mode to avoid intolerable performance loss or to switch IO mode to save power without losing too much performance. To be able to implement such a decision mechanism one should have some indicators that show what is going on right now and/or what would happen if the processor switch to the other execution mode. Such indicators would be branch misprediction rate, cache miss rate, ILP rate, and memory-level parallelism (MLP). Nevertheless, these indicators produce not direct information to switch but indirect and imperfect information to predict whether the decision mechanism should switch. Instead of using indirect indicators iMODE exploits the direct result by switching and trying the opposite execution mode via executing it in a small duration and collects one performance statistics. A major advantage of trying the intended mode by actually switching to it is its simplicity. By using this approach, iMODE is freed to implement any prediction circuitry which collects some statistics to make complex calculations. Another advantage is the precision of the accumulated result. Using the prediction would give us a result from the past of the execution while in our direct switch gives us the actual results of the alternative mode. Actually, in our direct approach, we are sampling the opposite execution mode by actually switching to it. In this approach, our sampling frequency and switch cost should be negligible in terms of performance.

Throughout this section, both design and implementation will be explained. We use Gem5 simulator to test our iMODE processor [7]. It simulates the passing of time as a series of discrete events, in other words, it is a trace-driven simulator. It is also designed in detail to be able to rearranged, extended, and parameterize its components. It supports multiple Instruction Set Architectures (ISAs) including ARM, ALPHA, MIPS, Power, SPARC, and x86. x86 ISA is chosen for this study since it is widely used for the general-purpose processors. Additionally, Gem5 ISA structure allows developers to simulate compiled code according to supported ISA on the Gem5 directly.

3.1.1. Epochs

In this study, the execution of the benchmarks are divided into epochs which is a constant duration in the number of cycles shown in Figure 3.1. To be able to catch when the switch decision should occur, the cpu source code of the simulator is modified. Since the simulator is trace-driven it is not guaranteed that it will execute the cycle function of the cpu in every cycle simulated. To solve that Algorithm 3.1 is used since it immediately find if the given length of the epoch is caught or passed when the execution reached the cycle function of the cpu. In this point, our proposed design should decide if a mode switch should occur or not. 1K cycles for epoch duration is an empirical result which gives us good results in our experiments.

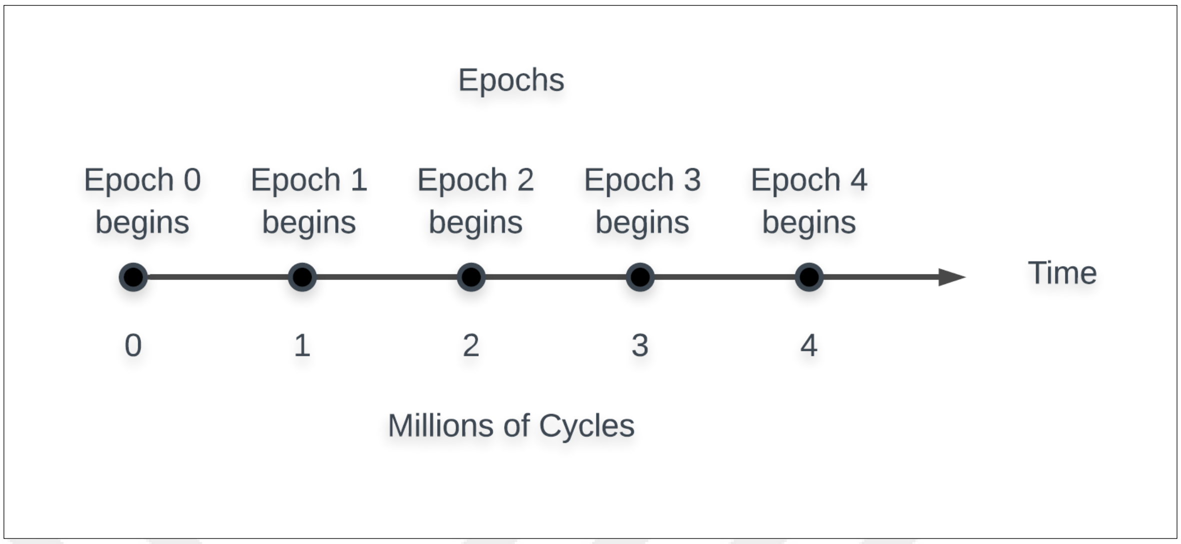

Figure 3.1. Epoch illustration

Algorithm 3.1. Epoch incrementing algorithm

```
1: $current\_cycle,\ epoch \leftarrow\ 0,$
2: $epoch\_length \leftarrow\ 1M$
3: if idiv($current\_cycle,\ epoch\_length$) $>$ epoch then
4:     $epoch\ =\ epoch\ +\ 1$
5:     $Decision\ Algorithm$
6: end if
```

As we mentioned above, iMODE processor have two execution modes, namely OoO and IO. In addition to the execution modes iMODE also have two different durations which are main duration and trial duration. While the execution modes affect the way of the execution principles execution durations determine the length of the execution modes.

3.1.2. Decision Points

Except for the first epoch, every single epoch consists of two execution durations. Starting with the first epoch point which is the end of the first millions of cycles, iMODE will try the opposite execution mode in trial duration. As a result, we can say that every epoch starts with trial duration and continues with a main duration. Again, the exceptional case is the first epoch which has not a trial duration. Ultimately, we can say that every epoch ends with a main duration. At the beginning of each epoch, our iMODE tries the opposite execution mode for a shorter trial duration and reach the decision point. In the decision point iMODE have two major sources of information, which are the IPC of previous main duration and the IPC of the trial duration. This is the essence of our iMODE, by looking this two information, iMODE decides to stay in the tried-execution-mode or to switch back to execution mode of the previous main duration.

iMODE processor aggressively tries the opposite execution mode at the start of every epoch to assure it is executing in the right mode. Periodically checking the opposite duration hurts the performance especially if the general trend of the execution is OoO mode. In the end, iMODE processor trading off a tolerable amount of performance drop with saving power.

In order to clarify better, Figure 3.2 illustrates an example run iMODE which shows execution durations and execution modes. In this study, when called nth epoch, it is meant the number of cycles from n million to n + 1M cycles. As seen in the figure, every epoch lasts for one million cycles and at the end of a million cycles one epoch ends and another one starts. From 1 million cycles to 2 million cycles, the first epoch is labeled as ”Epoch” in Figure 3.2. The first dashed line labeled as ”Epoch Start” shows the end of the second epoch and start of the third epoch as an example while the second dashed line labeled as ”Decision Point” represents an example decision point for iMODE processor in the third epoch. As mentioned above, every epoch has two duration which are trial and main durations. An example of a trial duration and a main duration labeled in the first epoch. Last but not least is the execution modes shown in orange and green boxes represents OoO and IO modes respectively.

As it is seen, the zeroth epoch which is the only epoch which has not a trial duration started in OoO mode and executed until the first epoch starts. iMODE immediately tries the opposite execution mode (IO mode) throughout the trial duration of the first epoch. After the end of the trial duration in the first epoch, iMODE decides whether to stay in the IO mode or switch back to OoO mode. In this example, iMODE decided to go back to OoO mode. every end of a trial duration is a decision point for the iMODE processor. After a switch-back decision to OoO mode, a main duration is executed in OoO for the first epoch.

Trying the IO mode in trial durations and switching back to OoO pattern continues until the sixth epoch. At the end of the main duration of the fifth epoch executed in OoO, iMODE processor tries the IO mode in throughout the trial duration of the sixth epoch, but this time decides to stay in IO mode and completes the main duration of the sixth epoch in IO mode. After this point we can see that main execution mode becomes IO mode and iMODE tries OoO mode in seventh, eighth, and ninth epochs in trial durations.

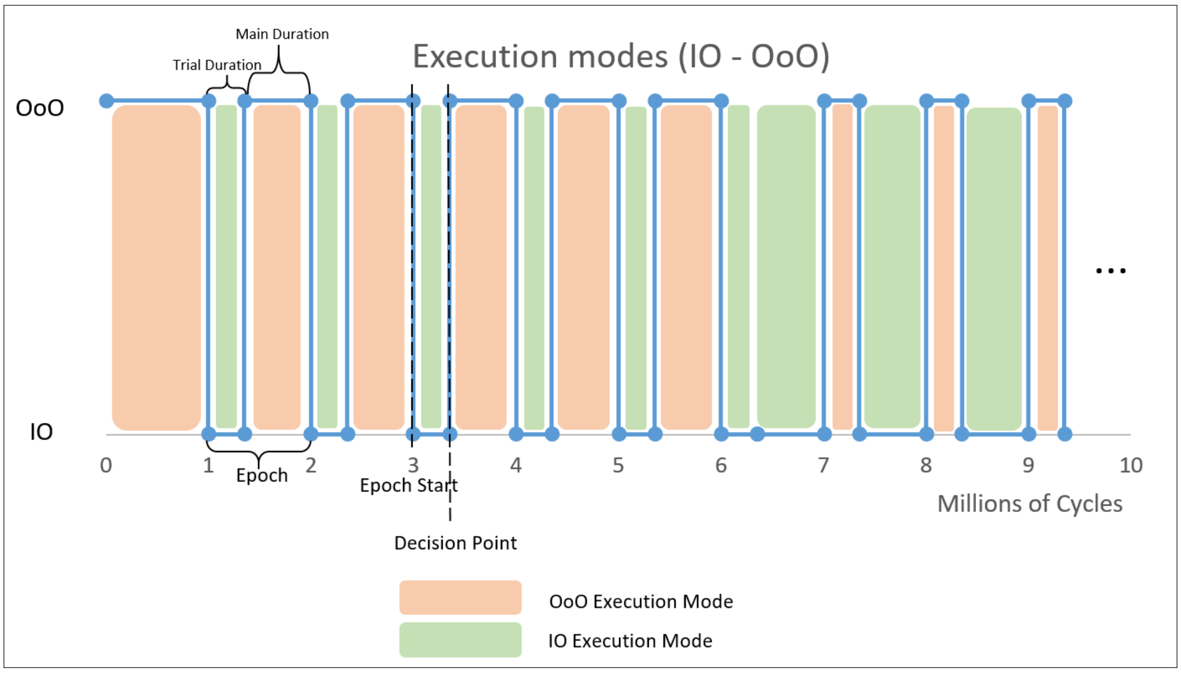

Figure 3.2. Example execution showing modes and phases

3.1.3. Decision Algorithm

At the end of each trial duration, iMODE processor decides to stay in the current tried-mode or switch back to the opposite execution mode. At the decision point iMODE processor have two different statistics. First one is previous epoch’s main duration IPC and second is current epoch’s trial duration IPC. Also, at decision point two threshold values are used which are *perf-loss-thresh* and *perf-gain-thresh* values. These threshold values are adjustable by the operating system in accordance with the performance need of the execution. iMODE processor calculates the performance difference between previous epoch’s main duration IPC with current epoch’s trial duration IPC. If the trial duration executed in IO mode then it can be said there is a performance loss and if the trial mode executed in OoO mode there is a performance gain.

Performance losses determine should the execution switch back to OoO mode or not while performance gains determine should the execution switch back to IO mode or not. At decision points, as mentioned above, there are two threshold values, first one, *perf-loss* thresh determines the maximum performance loss iMODE can tolerate when trying the IO mode, while the second one, *perf-gain-thresh*, determines how much the execution needs the performance. If these threshold values and execution changes are perceived in the perspective of OoO mode, they can be understood easily. To sum up, if performance loss is too high then go back to OoO mode and if the performance gain is too high then stay in the OoO mode since execution needs performance immediately.

Algorithm 3.2. Decision algorithm

```
$t_{md} \leftarrow Throughput\ from\ previous\ epoch's\ main\ duration$
$t_{td} \leftarrow Throughput\ from\ current\ epoch's\ trial\ duration$
$prev\_main\_duration \leftarrow Execution\ mode\ of\ prev\ epoch's\ main\ duration$
if $prev\_main\_duration$ is OoO then
    $perf\_loss = (t_{md} - t_{td})/t_{md} * 100$
    if $perf\_loss >$ perf-loss-thresh then
        $Switch\ back\ to\ OoO\ mode$
    end if
else if $prev\_main\_duration$ is IO then
    $perf\_gain = (t_{td} - t_{md})/t_{td} * 100$
    if $perf\_gain <$ perf-gain-thresh then
        $Switch\ back\ to\ IO\ mode$
    end if
end if
```

Algorithm 3.2 shows how iMODE decides when reached to the decision points. To clarify the algorithm, Several cases can be considered. Also, Figure 3.2 can help visualize the example cases. For example, let’s assume *perf-loss-thresh* is 10, *perf-gain-thresh* is 20, and execution is on the decision point of epoch 5. Note that, for the fifth epoch. Current epoch's trial duration is executed in IO mode. Previous epoch's main duration is executed in OoO mode. In this case iMODE should calculate a perf loss using the IPC from the trial duration and IPC from the main duration. After the end of the trial duration in epoch 5, it is seen that execution mode is switch back to OoO mode which means calculated performance loss is above 10 percent. iMODE tried the opposition mode which is IO mode and decided that IO mode causes to drop performance above the tolerated threshold and switched back to OoO mode. Looking at next epoch, in epoch 6, iMODE again tries the IO mode in the trial duration of the epoch 6. At the decision point of the epoch 6, the perf loss is calculated using the epoch 5's main duration IPC and epoch 6's trial duration IPC. However, this time calculated perf loss is under *perf-loss-thresh* and thus, execution stayed at IO mode. It means that at this time iMODE decides that it can tolerate the performance loss due to IO mode and save power efficiently.

3.1.4. Decision Postponing Algorithm

Despite that it is a small percentage of execution time, iMODE spends its time in alternate execution mode at the start of every epoch until the decision points. By this aggressive approach, iMODE always tries to find the efficient execution mode in a power sawing oriented way. This situation leads to performance losses particularly when the mood of the

Algorithm 3.3. Decision postponing algorithm

```
1:  $next\_mode,\ perf\_loss \leftarrow Decision\ Algorithm$
2:  if $next\_mode$ is OoO then
3:      if $perf\_loss > PT_{high}$ then
4:          $Postpone\ Decision\ for\ PE_{high}\ epochs$
5:      else if $perf\_loss > PT_{med}$ then
6:          $Postpone\ Decision\ for\ PE_{med}\ epochs$
7:      else if $perf\_loss > PT_{low}$ then
8:          $Postpone\ Decision\ for\ PE_{low}\ epochs$
9:      end if
10: end if
```

application wants OoO mode. To overcome this problem iMODE postpones the unconditional switches at the start of every epoch. In other words, the epoch interventions are deactivated during this time and iMODE runs on OoO mode without any interrupts. The idea behind this strategy is that if the calculated perf loss is very different from the *perf-loss* thresh value then it can be considered that the unconditional switches at the beginning of the epochs could be delayed for several epochs to not lose performance due to trials. Algorithm 3.3 shows the postponing algorithm where PT stands for the postponed threshold and PE stands for the postponed number of epochs.

3.2. ENFORCEMENT MECHANISM

Switching IO and OoO modes between each other should be as smooth as possible. At this point the enforcement mechanism is engaged in iMODE processor and implements the necessary changes in the hardware. Although our proposed design largely uses the traditional OoO layout, some of the components need to be changed or may be shut down when switching to the IO mode. One of the main differences between datapath structures of OoO and IO processors is the Issue Queue (IQ). In the IO processors, IQ is not different from traditional queue structure where elements are entering from the tail and exiting from the head of the queue. While the head of the IQ, points the oldest instruction, tail of the IQ points the newest instruction according to the program order. In IO processors, according to machine width of the processor, n a number of ready instructions are scheduled from the head of the IQ. Readiness is determined according to the source operands of the instruction, if all of the source operands become valid for the instruction, it is considered to be ready. On the other hand, in OoO processors instruction scheduling is much more complex compared to the IO processors. Since program order is not important in OoO instruction scheduling, any instruction which becomes ready can be scheduled independently from the age of the instruction. Also, ready instructions may reside anywhere in the queue which is held as fetched instructions. As a result, this structure cannot be called traditional queue anymore since it becomes more like buffers in OoO processors. Usually, this mechanism is called as Instruction Dispatch Buffer (IDB) in OoO processors. In IDBs, any instruction can become ready and ready instructions could be anywhere in the buffer. For this reason, such as instruction selection logic and wake-up mechanisms are used which make the OoO instruction scheduling more complex compared to IO instruction scheduling.

When execution switches to the IO mode from the OoO, instructions must schedule in program order. For this reason, in iMODE processor, the enforcement mechanism blocks the instruction scheduling until the instruction pipeline drains and become empty. Due to some long-latency instructions, such as floating point division and square root instructions, waiting for draining the pipeline can last long number of cycles. Especially the nondeterministic memory instructions in the IQ leads draining operation become unbounded. To overcome this issue, iMODE enforcement mechanism bound the draining operation to five thousand cycles (an empirical duration which gives us good feedback in our experiments). After this duration, whether or not the pipeline becomes empty, everything in it flushed by the enforcement mechanism. After the pipeline become empty, OoO instruction scheduling is deactivated and simpler IO instruction scheduling is activated. As soon as the IO scheduling is activated, instructions are started to dispatch into IQ in program order.

Going to OoO from the IO mode is much more simple. Due to the fact that the native execution mode in iMODE processor is OoO, switching to IO mode requires certain mechanisms to be shut down or modified. On the other hand, switching back to native execution mode is simpler since no modification is required. When switching to OoO mode, activating the OoO instruction scheduler is enough. Thus, switching to OoO mode has no significant delay.

To sum up, after every switch event occurred, new execution mode is marked and according to the new execution mode enforcement mechanism gets involved and enforce the necessary changes in the iMODE processor. If the new mode is OoO, only the instruction scheduling mechanisms should be switched, no need to do any further action, see line 7 in Algorithm 3.4. However, if the new execution mode is IO, then some other controls and actions should be done. See Algorithm 3.4, when the new mode is IO, after deactivating the OoO instruction scheduling, iMODE marks the current time to understand when the switch is started, and set the is in transition register to be able to do further actions. Hence, if the processor is in the transition state, iMODE will be waiting for the pipeline to be drained or forced to flush all of the instructions, see line 11 in Algorithm 3.4. If the forced flush cycles tolerance time in cycles is not done yet, enforcement mechanism is constantly checking the number of instructions in the ROB to understand whether the pipeline is drained, see line 13 in Algorithm 3.4. After a pipeline drain or a forced flush, in_transition bit is resetted to false again.

Algorithm 3.4. Enforcement algorithm

```
1:  $forced\_flush\_cycle\_tolerance \leftarrow 5000$
2:  $execution\_mode \leftarrow New\ execution\ mode$
3:  if $execution\_mode$ is IO then
4:      $Deactivate\ OoO\ instruction\ scheduling$
5:      $is\_in\_transition \leftarrow true$
6:      $transition\_start\_time \leftarrow current\_cycle$
7:  else if $execution\_mode$ is OoO then
8:      $Deactivate\ IO\ instruction\ scheduling$
9:      $Activate\ OoO\ mode\ instruction\ scheduling$
10: end if
11: if $is\_in\_transition$ then
12:     if current_cycle $<$ $(forced\_flush\_cycle\_tolerance + transition\_start\_time)$ then
13:         if number of instructions in ROB == 0 then // is pipeline drained?
14:             $Activate\ IO\ mode\ instruction\ scheduling$
15:             $is\_in\_transition \leftarrow false$
16:         end if
17:     else
18:         $is\_in\_transition \leftarrow false$
19:         $Flush\ all\ the\ pipeline$
20:     end if
21: end if
```

3.3. OTHER HARDWARE COMPONENTS

Applications run on execution phases which are suitable for IO execution in return of tolerable performance loss, in such a phase, iMODE aims to save power by switching the execution mode to IO. As mentioned in the previous section, the instruction scheduling mechanism could be modified with the intent to save power. Indeed, the major power saving achieved by shutting down the wake-up and selection logic runs in OoO mode with the much more simpler IO instruction scheduling which dispatches the first n instructions from the head of the IQ.

Actually, there are other datapath structures that can either be modified to run on a limited way or completely shut down when running in IO mode. For instance, following the methodology of Khubabib et al., Load Queue (LQ) could be completely shut down since executing the load instructions speculatively could not be done in IO mode of iMODE processor [22]. Moreover, Re-Order Buffer (ROB) could be completely turned off when switching to IO mode. Since the execution in IO mode flows in program order, the IO mode of iMODE does not need a re-ordering mechanism. Also, note that turning off ROB is safe considering the OoO to IO transition stage. In iMODE before the switch operation to IO, all of the pipeline is waited to be drained which means there will be no instruction in the ROB before iMODE execute in OoO mode. In this point, ROB circuitry could be turn off and head and tail pointers could be reseted. Until the execution switches back to OoO mode, ROB could stay turned off and ready to be opened and reused again. Last but not least, the same strategies could be applied to register file and register renaming mechanisms.

As mentioned before, threshold pairs can be adjusted by the OS according to the performance goal of the system. For the general purpose systems, the energy-delay product is the most acceptable criteria. Hence, we use energy-delay product to find out which thresholds are the best in saving power efficiently.

4. EXPERIMENTAL METHODOLOGY

We used 16 benchmarks from spec2006 to evaluate our proposed design. All of the experiments are simulated in Gem5 using the x86 ISA [7, 23]. Each benchmark is simulated for 100M instructions. The configuration parameters used in experiments are given in Table 4.1. Empirically obtained PT values are given in Table 4.2 and PE values are given in Table 4.3.

Table 4.1. Specifications of the simulated processor

CPU Components	**Specifications**
L1 I- and D-Caches	16 Kb, 4-Way, 64-byte line size, 2 cycle latency
L2 Caches	128Kb, 8-way, 64-byte line size, 20 cycle latency
CPU Frequency	2 GHz
Pipeline	4-way issue, 192 entry ROB, 64 entry IQ, 32 entry LQ, 32 entry SQ, 256 int and 256 FP registers

Spec2006 benchmarks are divided into two groups according to their utilization of OoO execution mode. For the classification, we first collect the base OoO and IO performances of the benchmarks one by one. If a benchmark results 40 percent more throughput compared to its IO execution mode performance, marked as a high-utility benchmark, otherwise, it is marked as low-utility. As anticipated, high-utility benchmarks are more OoO dependent, as a result, they do not tolerate executing in IO mode, whereas, the low-utility ones embrace the IO mode since they already cannot utilize the OoO execution mode. Classification aims to show the significance of the power savings of the low-utility benchmarks compared to high-utility ones.

In our simulation regions, 16 benchmarks are classified as shown in Table 4.4. To evaluate the iMODE, three metrics are used. Throughput, power saving, and energy efficiency. For energy efficiency Equation (4.1) and Equation (4.2) is used. In the equation, energy and delay parameters are calculated as percentage values found by accepting the full-OoO simulations are the baseline values for iMODE. Hence, the desired efficiency value should be less than one, meaning that the energy cost per IPC is less than the traditional OoO execution. In here Equation (4.3) are also provided. This is because in literature, all three of these equations are used. The difference is that the effect of the delay parameter. In general purpose processors energy-delay is the most common criteria to evaluate the efficiency of the system. The second equation, energy-delay square is used as a criteria for evaluation of workloads and the third equation delay-cube is used from servers and data centers. Compared to energy-delay equation, other two equations increase the penalty of the delay by squaring and cubing the effect of delay. Hence, when the performance drop tolerance is decreased, delay square and delay cube are used.

Last but not least, we assumed the power saving of iMODE as 2x which is an extremely pessimistic value for iMODE compared to 4x and 6x power savings from the literature [1, 22]. As a result, we assume that iMODE dissipates half of the energy dissipated by traditional OoO core. Equation (4.4) shows the calculation for power consumptions. By using ratio of the power consumptions for two desired experiment, the power savings are calculated as percentage values.

$$Efficiency = Energy \cdot Delay \tag{4.1}$$

$$Efficiency = Energy \cdot Delay^2 \tag{4.2}$$

$$Efficiency = Energy \cdot Delay^3 \tag{4.3}$$

$$Power\ Consumption = \#\ of\ cycles\ spent\ in\ IO\ mode + 2 \cdot (\#\ of\ cycles\ spent\ in\ OoO\ mode) \tag{4.4}$$

Table 4.2. Parameters for postpone threshold

PT_{high}	80%
PT_{med}	50%
PT_{low}	30%

Table 4.3. Parameters for postponed number of epochs

PE_{high}	80%
PE_{med}	50%
PE_{low}	30%

Table 4.4. Classification of benchmarks

CPU Components	**Specifications**
L1 I- and D-Caches	16 Kb, 4-Way, 64-byte line size, 2 cycle latency
L2 Caches	128Kb, 8-way, 64-byte line size, 20 cycle latency
CPU Frequency	2 GHz
Pipeline	4-way issue, 192 entry ROB, 64 entry IQ, 32 entry LQ, 32 entry SQ, 256 int and 256 FP registers

5. RESULTS AND DISCUSSION

In this chapter, iMODE is compared with the traditional OoO processor. Each section evaluates the iMODE for different aspects which are the effects of *perf-loss-thresh* and perfgain- thresh, the effect of the length of the trial duration, and lastly, the effect of the decision postponing algorithm.

Since the proposed design of iMODE is explained in previous chapters, from this point, we can compare iMODE with the proposed design of Lukefahr et al., titled Composite Cores [1], in the perspective of disadvantages and design problems:

- For their proposed design they need much more space. Despite they use a mutual front-end, big and little cores have very different back-end architectures. Hence, they have to have separate two cores within their design. On the other hand, iMODE needs only one OoO core with slightly modified structures.
- In iMODE, during IO mode of execution, we shut off all of the unnecessary structures to save both dynamic and static power, however, in Composite Cores, after they migrate from one core to another, the old core probably waits in idle mode (in the paper they do not mention what happens to the old core). As a result, by waiting idle, they save from only dynamic power since there is no transistor activity in the idle core.
- Another design problem is to assume a very optimistic 80-cycle latency for the main memory.
- They also present 1000 cycles for their decision point epoch. However, this duration is very short for the mood change of an application. From our experiments, we see that the transition from one mode to another may not be instantaneous. On top of that, their proposed architecture has a register transfer overhead in each migration.
- They try to predict the performance of the other core by using some indirect indicators. One of the indicators is the ILP calculated using the ready instructions in the IQ. However, during our experiments, we noticed that calculating ILP solely using the instructions from IQ yields spurious ILP score since IQ holds all instructions including speculative ones, as well. Unfortunately, the instructions in

the mispredicted path are squashed and they have no effect on the commit IPC. To obtain a more accurate ILP estimate, we suggest that ILP among committed instructions can be collected. In fact, again in our experiments, we inserted an extra buffer to the end of the ROB to hold n number of instructions to calculate their ILP. Yet, we may not obtain perfectly healthy results since the micro ops of the x86 architecture hide the true ILP between the committed instructions.

The results presented in Chapter 5.1 and 5.2 show the effects of changing threshold values and trial duration lengths, whereas the results presented in Chapter 5.3 demonstrate the benefits of applying decision postponing. All results are evaluated in terms of throughput, power consumption, and energy efficiency (via energy-delay product [27]). While obtaining the energy consumption for different configurations, a similar but more pessimistic approach [22].

5.1. PERFORMANCE THRESHOLDS

In Chapter 3, in Figure 3.4, we mentioned that iMODE is using two different thresholds to be able to decide switch back or stay in an execution mode. These thresholds, namely *perf-loss-thresh* and *perf-gain-thresh* have critical effect on the performance of the proposed design. Note that, *perf-loss-thresh* is used to set a threshold for when to switch back after IO is tested, and *perf-gain-thresh* is used to measure how much the application needs for OoO, performance mode, after OoO mode is tested. Hence, decreasing the *perf-loss-thresh* value leads iMODE to switch more easily and frequently to the OoO mode since tolerance for the performance loss is decreased. Increasing the *perf-gain-thresh* leads iMODE to stay in IO mode more frequently since the *perf-gain-thresh* is high and switching back to OoO mode becomes harder. Thresholds can be adjusted by the OS if there are certain QoS concerns. Moreover, system can have certain performance goals or power saving goals to meet which iMODE can help to achieve.

IPC lost caused by iMODE under different thresholds pairs are shown in Figure 5.1. As can be guessed, IPC loss in high-utility benchmarks is higher than low-utility benchmarks since they are more performance demanding applications. As a result, the IPC to loss due to trial durations is more striking to high-utility benchmarks. However, low-utility benchmarks are much more resistant to IPC loss since they already do not need high performance and they utilize the iMODE according to the power saving aspect.

The results also show that IPC loss decreases when the *perf-loss-thresh* increases, see the difference between the first two configurations. We can understand why it is happening using a small example, iMODE tries the IO mode and calculates a performance-loss value.

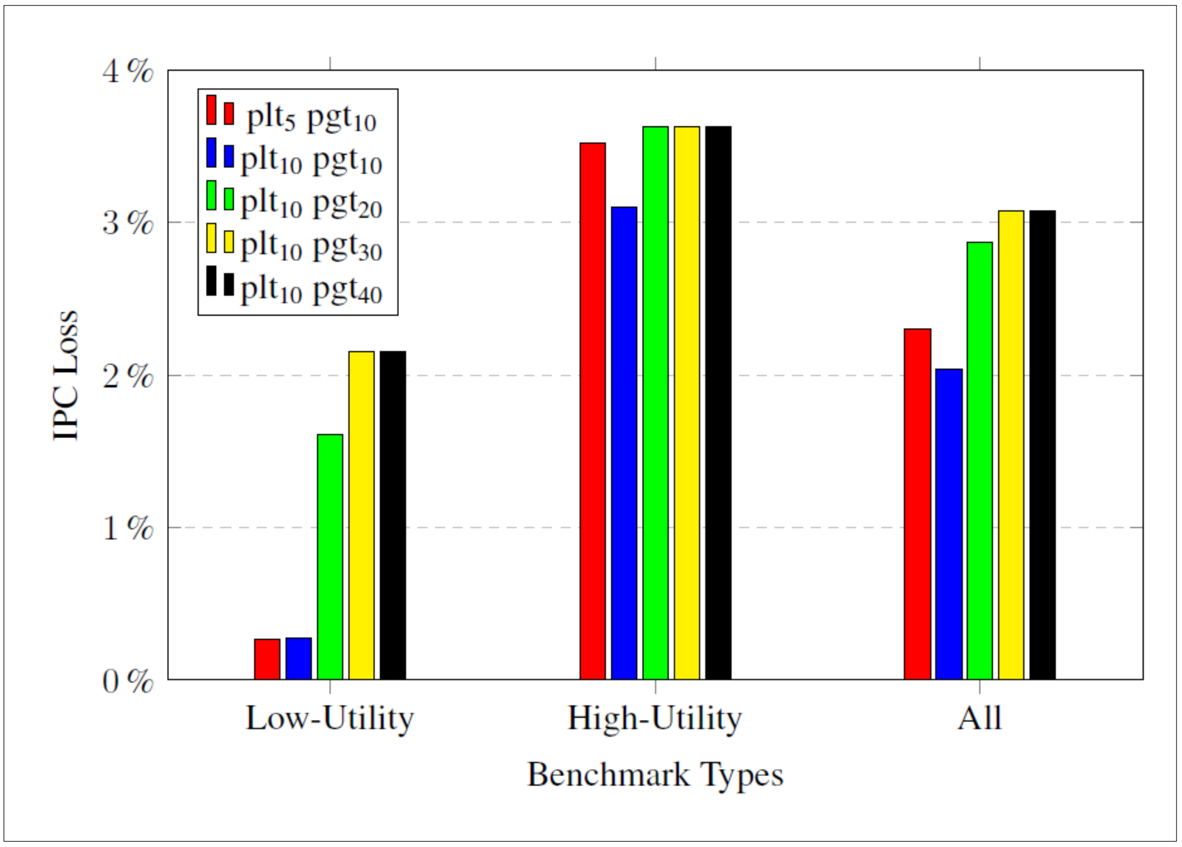

Figure 5.1. IPC loss of iMODE compared to traditional OoO core with various threshold pairs. Numbers in the legends represent perf-loss-thresh (plt) and perf-gain-thresh (pgt) values, respectively.

If this value is greater than 5 percent, it switches back to OoO mode, in the second configuration, *perf-loss-thresh* is increased to 10 percent which makes harder to switch back to OoO mode. This situation results in lower IPC loss due to decreasing IO mode execution. When we look at to 2., 3., 4., and 5. configuration, we can see that *perf-gain* thresh is increasing and IPC loss is increasing. We can also understand this situation using a small example. iMODE calculates a performance-gain value when tries to OoO mode in a trial duration. And then, if this performance-gain value is higher than *perf-gain-thresh* we understand that application is in need of higher performance, hence, we decide to stay in OoO mode. If this performance-gain value is less than 10 percent, iMODE decides to switch back to IO mode. When *perf-gain-thresh* is increased, switching back to IO mode becomes easier. In other words, iMODE's power-saving motivation is increased, thus IPC loss is increased.

Figure 5.2 shows power savings in percent according to different threshold pairs. We can see that threshold pairs which result in high IPC losses usually gives high power savings. This is because the iMODE utilizes the IO execution mode more when the *perf-gain-thresh* is higher. Figure 5.2 also show that in low-utility benchmarks, power savings are much more compared to high-utility benchmarks. The reason behind these results is simple, the more iMODE utilizes IO execution, the more it will save power while losing performance. The exceptional case occurs in high-utility benchmarks, iMODE processor spends even more power to run these benchmarks. For these benchmarks, the IPC difference is so high that switching to IO mode hurts the performance so much that in the end, iMODE spends even much more power compared to baseline OoO execution. Figure 5.3 shows the comparison for the efficiency gain values of the different threshold pairs.

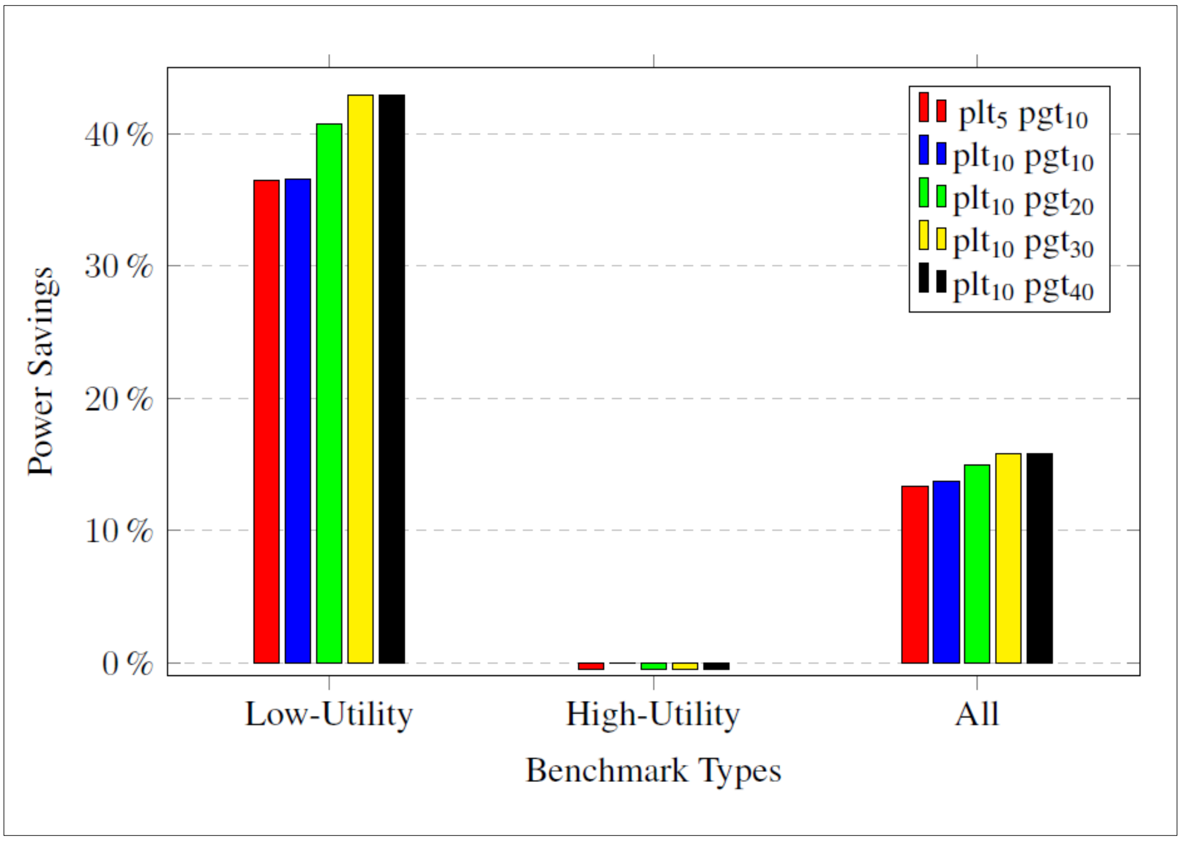

Figure 5.2. Power savings of iMODE compared to traditional OoO core with various threshold pairs. Numbers in the legends represent perf-loss-thresh (plt) and perf-gain-thresh (pgt) values, respectively.

iMODE processor has the highest efficiency gain in low-utility benchmarks since it can utilize the IO mode by not losing too much IPC. However, iMODE processor has negative efficiency gain for high-utility benchmarks. This is because iMODE either loses an unacceptable amount of IPC or leads an unacceptable slow-down while losing some IPC. As a result, iMODE uses energy approximately 13 percent percent more efficiently than traditional OoO processor.

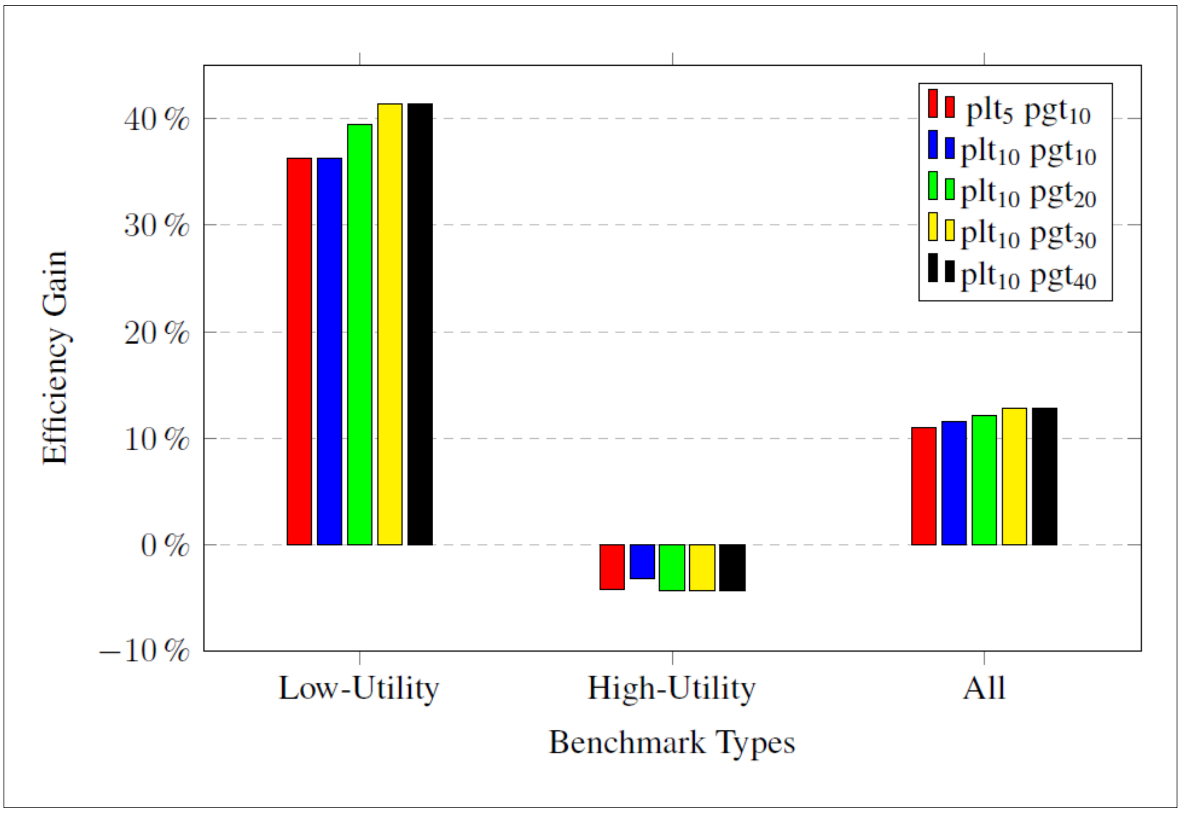

Figure 5.3. Efficiency gain of iMODE compared to traditional OoO core with various threshold pairs according to the energy-delay product. Numbers in the legends represent perf-loss-thresh (plt) and perf-gain-thresh (pgt) values, respectively.

When we use energy-delay-square product (shown in Equation (4.2)), we immediately see that efficiency values are decreased, as seen in Figure 5.4. This is not a surprise since energy-delay-square product formula gives the delay, in other words IPC loss, much more significance.

However, despite using much more harsh efficiency formula in terms of IPC loss tolerance, we can see that iMODE is still efficient compared to traditional OoO processors on the average. Similar to energy-delay product results, most efficient threshold pair is the 10 percent and 30 percent for *perf-loss-thresh* and *perf-gain-thresh* respectively.

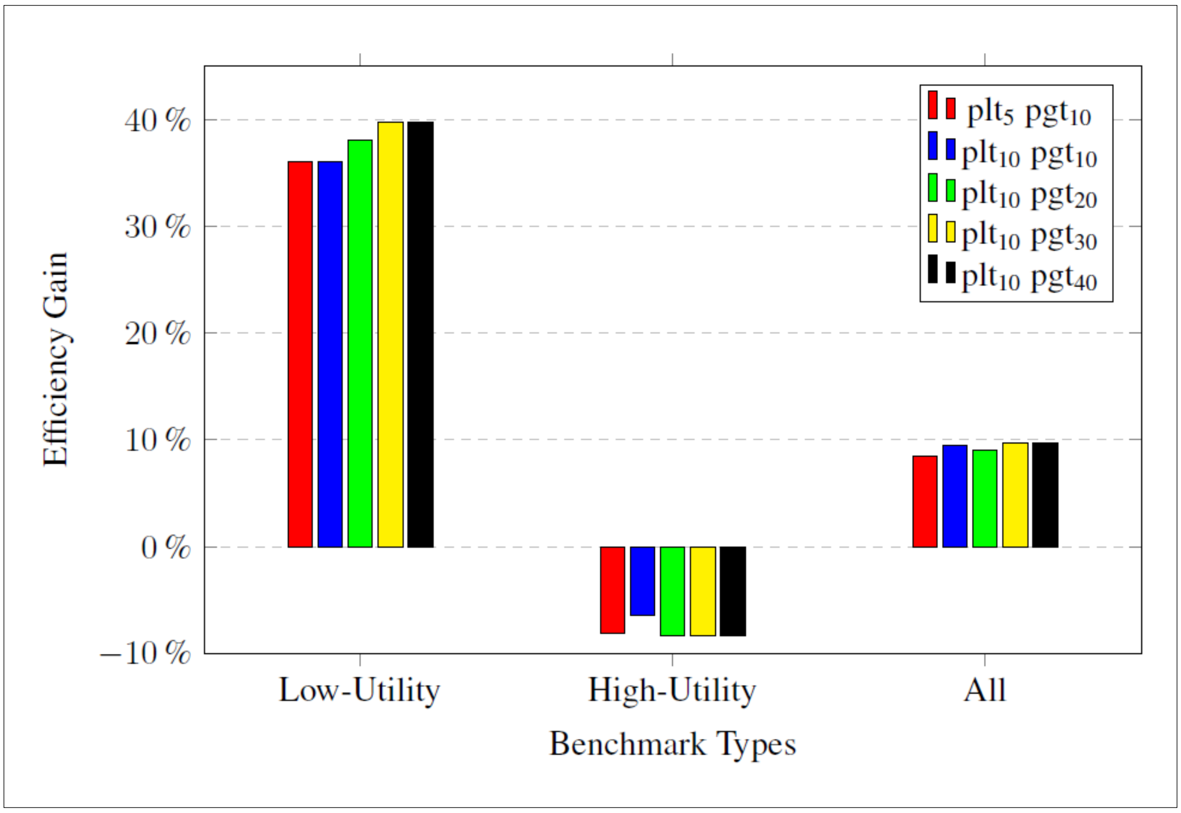

Figure 5.4. Efficiency gain of iMODE compared to traditional OoO core with various threshold pairs according to the energy-delay-square product. Numbers in the legends represent perf-loss-thresh (plt) and perf-gain-thresh (pgt) values, respectively.

Increasing the *perf-gain-thresh* value from 30 to 40 has no difference in any metric. The reason behind that is the benchmarks did not respond to the *perf-gain-thresh* value from 30 percent to 40 percent, calculated performance-gain value is either smaller than 30 or greater than 40 in the simulated interval. From now on, we evaluate the iMODE processor using the 10 percent *perf-loss-thresh* and 30 percent for the *perf-gain-thresh*.

The individual throughput loss and power savings for all benchmarks when the selected threshold pair is utilized (*perf-loss-thresh* is 10 percent and *perf-gain-thresh* is 30 percent) are presented in Figures 5.5 and 5.6, respectively.

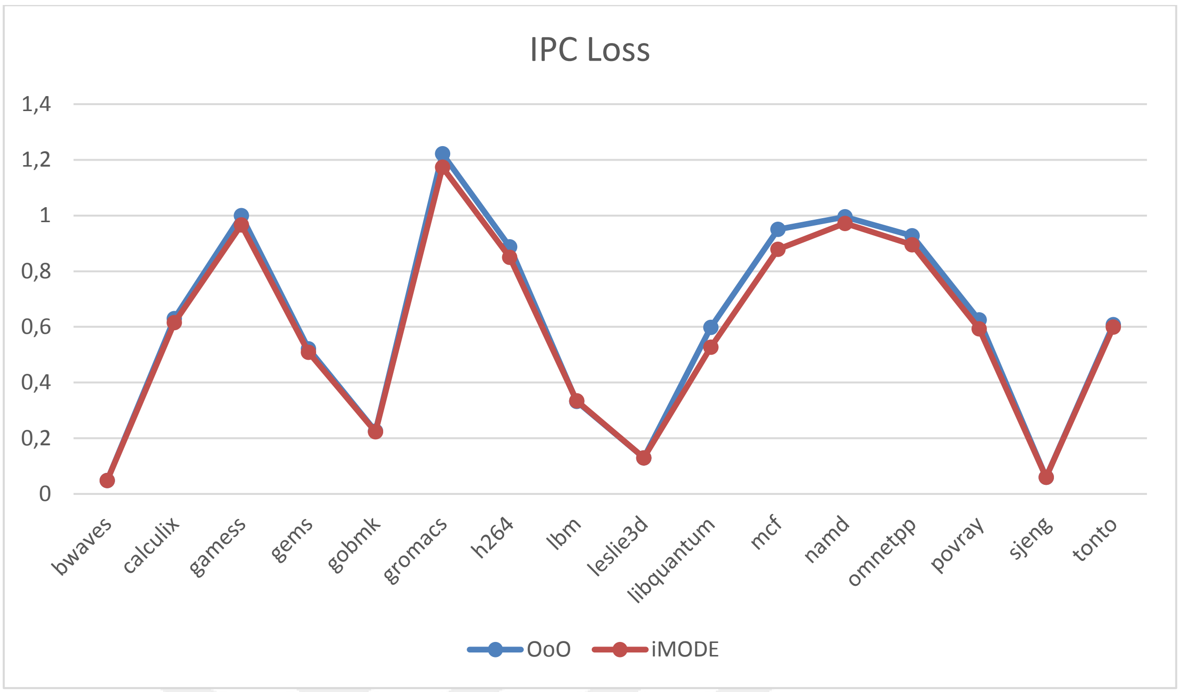

Figure 5.5. IPC Loss for all of the benchmarks when plt = 10 and pgt = 30

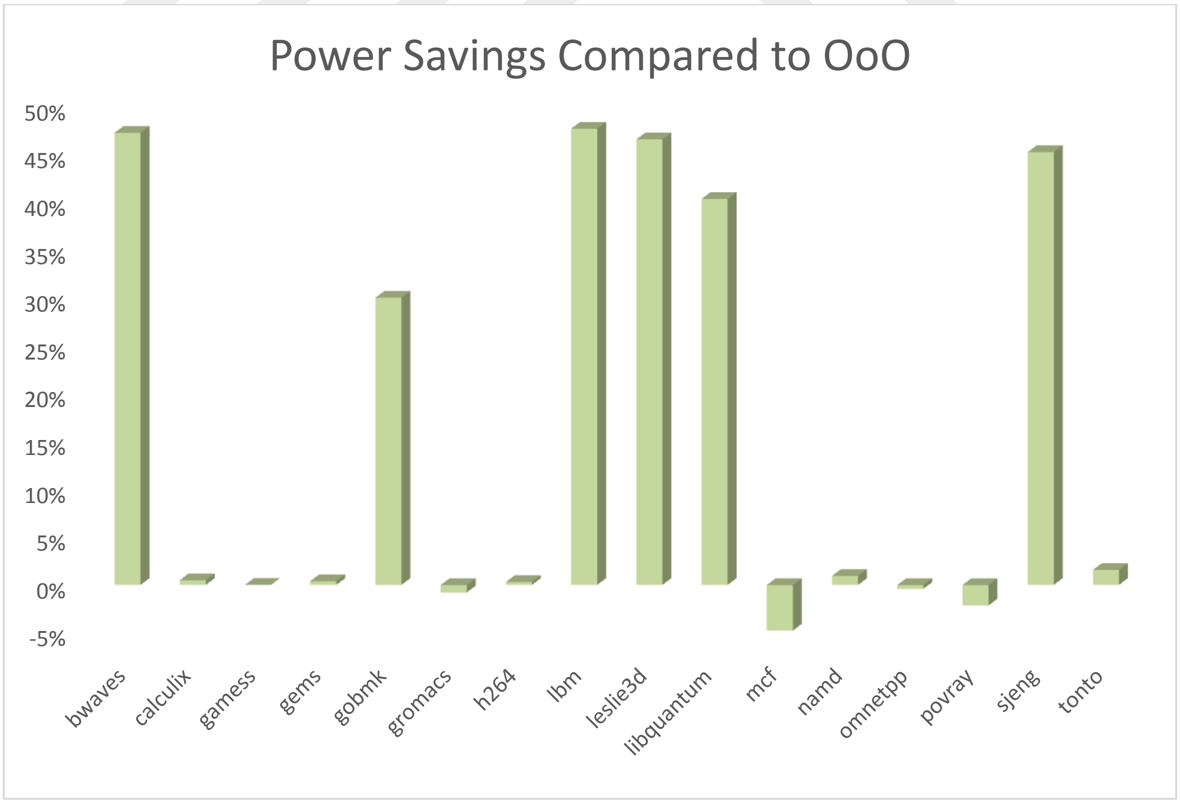

Figure 5.6. Power savings for all of the benchmarks when plt = 10 and pgt = 30

5.2. TRIAL DURATION LENGTH

The IPC loss in high-utility benchmarks caused by trial durations which mainly executed in IO mode. Since high utility benchmarks are utilizing OoO mode greatly, trying the IO mode for a trial duration length hurts performance. To be able to observe its effect, we evaluate iMODE for three different trial duration length.

In Figure 5.7, we can see the iMODE results in terms of IPC loss, power savings, and energy efficiency. In these experiments, we accept 10K cycles length trial duration as a basis and compared it with 50K and 200K cycles length of trial durations. Our preliminary experiments showed us using a shorter trial duration than 10K cannot provide sufficient information to determine the current mood of the application accurately.

The results show that extending the trial duration leads to an increase in both IPC loss and power savings. This is an expected result especially for the high-utility benchmarks since their main durations are mostly OoO, and their trial durations are mostly IO. They were already suffering trial duration IPC losses, hence, extending the trial duration length caused more than 10 percent IPC loss for high utility benchmarks. For this reason, 200K cycles trial duration is not acceptable for iMODE since it causes a harsh decline in IPC loss. To consider 50K cycles length, IPC loss for baseline 10K is acceptable, however, both the littleness of power saving and loss in energy efficiency 50K is also not better than 10K cycles length of the trial duration.

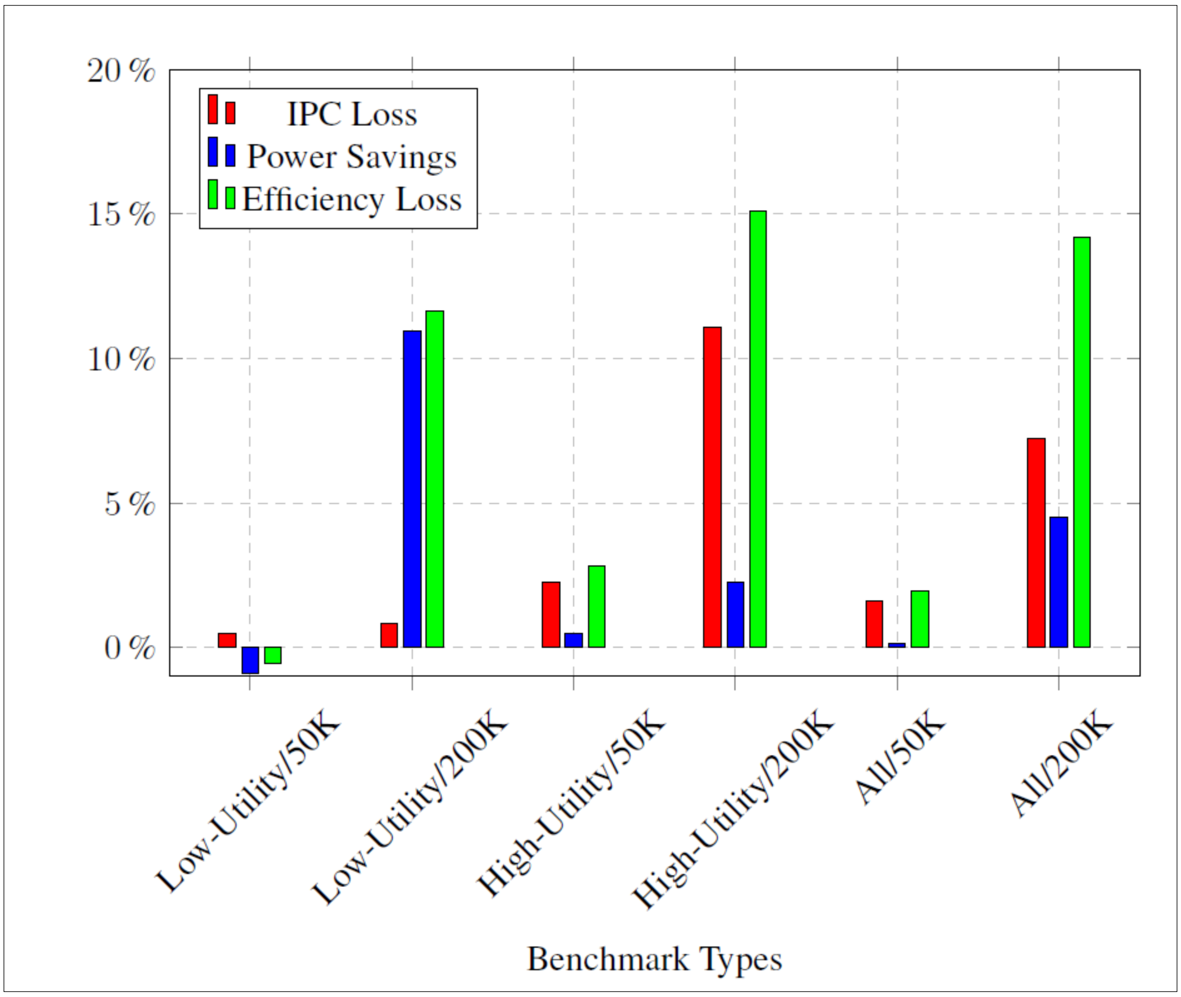

Figure 5.7. Comparison between different trial durations in terms of throughput, energy cost, and efficiency. Baseline duration length is 10K cycles.

5.3. DECISION POSTPONING

Delaying the decision points is a way to reduce the ratio of trial durations to the main duration. If the iMODE is confident selecting any mode, either OoO or IO, delaying the decision points save us from trying the opposition mode unconditionally. Thus increases either the energy efficiency or decreases the IPC loss according to the type of the application. When OoO mode selected in decision point depending on a confidence level we postpone the next decision point by a number of epochs to stay in the OoO mode. This strategy is applicable to IO mode as well but currently wiMODE applies it to the OoO mode only. This is because our main goal is to hold IPC losses at a certain level rather than increase the efficiency level. Although it is a future research direction, in the presented results iMODE makes postponing only if the main duration is OoO. Applying it to IO mode also have a side effect that if a bad decision is made high IPC losses are potential risk for iMODE. On the other hand, bad decisions do not comprise a risk to OoO mode since it only leads to missing potential power savings, but not losing IPC losses.

In the section which performance thresholds are discussed, five suitable threshold pairs are used in our experiments and after this point in evaluating iMODE we use 10 percent for perfloss- thresh and 30 percent is used for *perf-gain-thresh*. In decision postponing algorithm, we use Postpone Threshold (PT) and Postponed number of Epochs (PE) to delay the decision points. We can say that performance-loss value after tried IO execution could not be 100 percent, because it means iMODE is not committing any instructions, or in other words, it is stalling. For the purpose of getting preliminary results and have an insight whether the decision algorithm has a positive effect on energy efficiency, we divide the minimum 30 percent and maximum 100 percent performance-loss interval into three regions as seen in Table 4.2. After that, we choose 3 different PE values to evaluate. These values are shown in Table 5.1. To understand and observe the effects of PT and PE values, we choose traditional OoO execution as the baseline results for the iMODE and experiment with three different PE values. Results are shown in Figure 5.8, we can see that for the low-utility benchmarks there is no difference in terms of energy efficiency, except for 0.05 and 0.2 decreases on second and third experiments, respectively. On the other hand, as for the high-utility benchmarks, we can see that there was no efficiency gain in experiment one and two but for experiment three there is a clear increase in efficiency gain. This is because our postpone algorithm is for designed OoO execution only. After these results, we choose experiment three’s PE values for the evaluation of decision postponing.

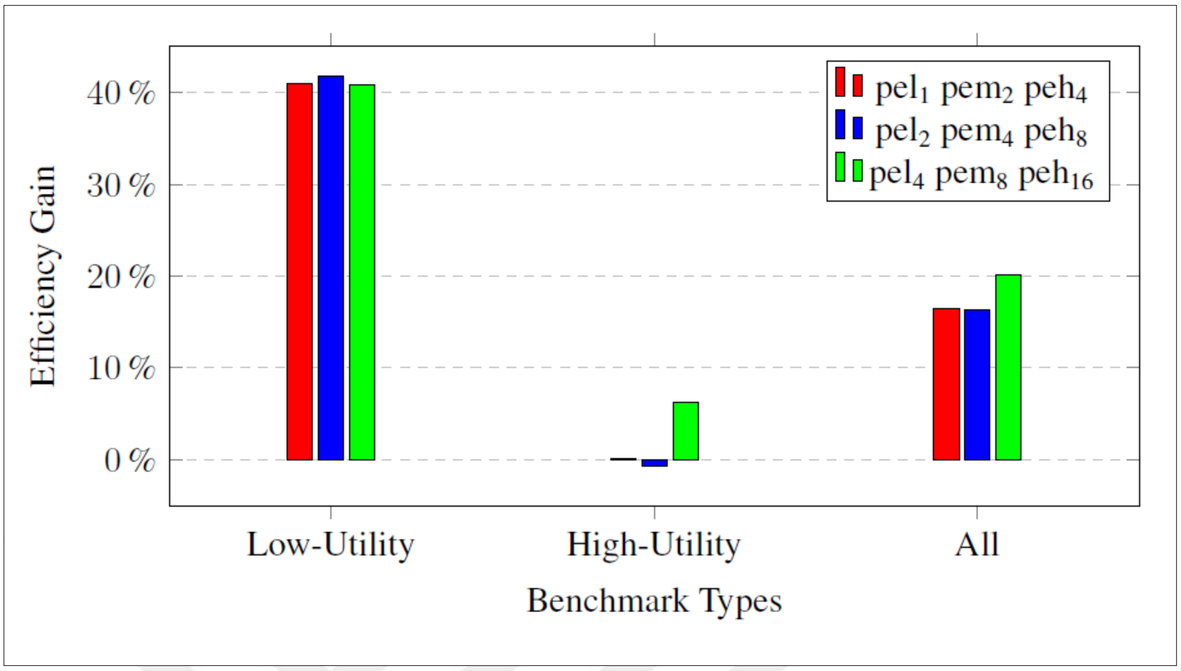

Figure 5.8. Efficiency gain of iMODE with different PE values compared to baseline traditional OoO results. Numbers in the legends represent pel (PE_{low}), pem (PE_{medium}), and peh (PE_{high}) values, respectively.

Table 5.1. Parameters for postponed number of epochs for different experiments

	Exp1	**Exp2**	**Exp2**
PE_{high}	4	8	16
PE_{medium}	2	4	8
PE_{low}	1	2	4

In Figure 5.9, we can see the performance of iMODE in three evaluation criteria when the decision postponing algorithm is applied. The results show that delaying the decision points increased the performance of iMODE in all fronts, except a very little IPC loss in low-utility benchmarks (0.02 percent) and power saving in high-utility benchmarks (-0.3 percent). However, efficiency is improved both low and high-utility benchmarks. When considered all of the benchmarks, postponing the decision point reduces the IPC loss by 0.6 percent, increases the power saving by 1.3 percent, and increases the efficiency gain by 2.1 percent. As a result, the iMODE processor reduces power consumption by 17 percent on average for the cost of a 0.8 percent drop in throughput. It also increases efficiency by 16.3 percent compared to a traditional OoO core with a similar datapath configuration.

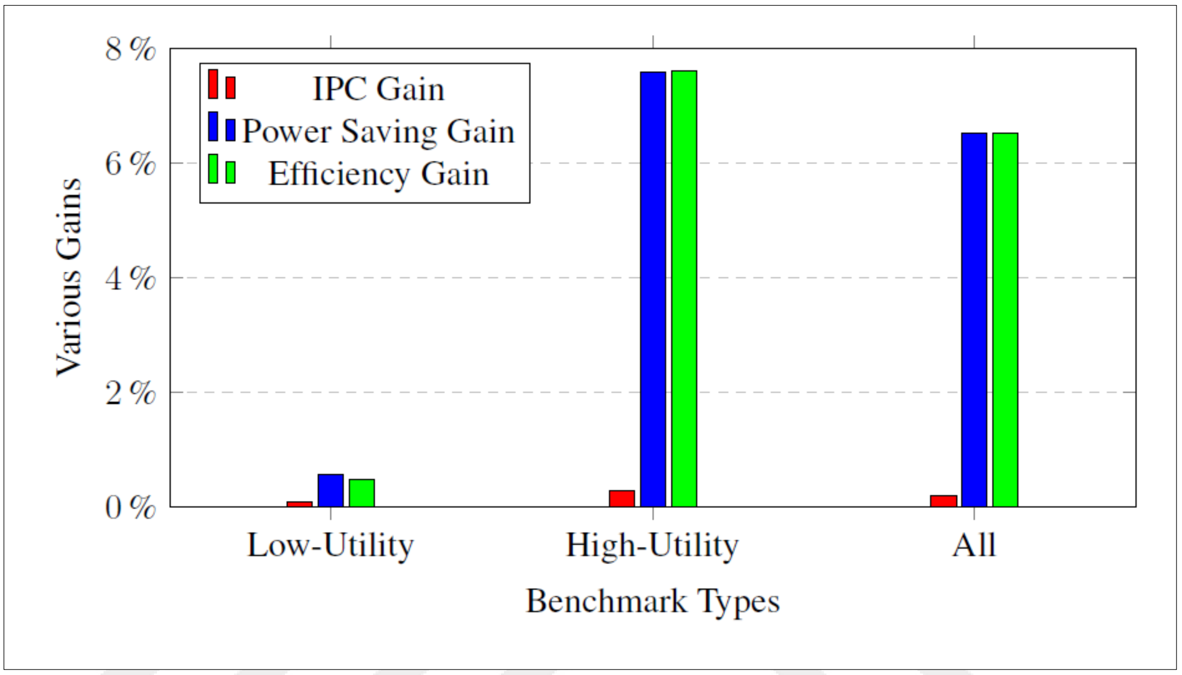

Figure 5.9. Performance of iMODE when decision postponing is utilized, compared to the baseline configuration. Trial duration is 10K cycles.

6. CONCLUSION

In this study, we focus on the design and implementation of iMODE processor that can alternate its execution mode seamlessly between OoO and IO modes. Mode switching decisions are made after running applications and collecting performance statistics in both modes. The decision algorithm is run periodically, and at the end of each epoch, an enforcement mechanism accomplishes the mode-switch decision. We allow the pipeline to drain after a mode switch decision, but to alleviate large performance penalties, which can occur at the end of these mode switching decisions, we flush the processor pipeline after five thousand cycles. We also integrate a confidence mechanism for postponing trial modes when the OoO mode has a considerable performance advantage over the IO mode. In our experiments, we show that the iMODE processor can really track the mood changes of applications, in an accurate manner. iMODE achieves 17 percent power savings with only less than 1 percent of performance drop on the average across all simulated benchmarks, while achieving peak power savings and efficiency gain figures of 47 percent and 48 percent, respectively.

REFERENCES

1. Lukefahr A, Padmanabha S, Das R, Sleiman FM, Dreslinski R, Wenisch TF, Mahlke S. Composite cores: pushing heterogeneity into a core. *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on;* 2012: Microarchitecture.

2. Kumar R, Farkas KI, Jouppi NP, Ranganathan P, Tullsen DM. Single-ISA heterogeneous multi-Core architectures:the potential for processor power reduction. *Proceedings 36th Annual IEEE/ACM International Symposium on;* 2003:Microarchitecture.

3. Annavaram M, Grochowski E, Shen J. Mitigating Amdahls Law through EPI throttling. *Proceedings of the 32nd Annual International Symposium on;* 2005:Computer Architecture.

4. Afram F, Ghose K. FlexCore: A reconfigurable processor supporting flexible, dynamic morphing. *Proceedings of the 2015 IEEE 22nd International Conference on;* 2015:High Performance Computing (HiPC).

5. Greenhalgh P. Big.LITTLE processing with ARM Cortex-A15 and Cortex-A7: Improving energy efficiency in high-performance mobile platforms. White Paper, ARM; 2001.

6. Kumar R, Tullsen DM, Ranganathan P, Jouppi NP, Farkas KI. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. *Proceedings 31st Annual International Symposium on;* 2004:Computer Architecture.

7. Binkert N, Beckman B, Black G, Reinhardt SK, Saidi A, Basu A, Hestness J, Hower DR, Krishna T, Sardashti S, Sen R, Sewell K, Shoaib M, Vaish N, Hill MD, Wood DA. The gem5 simulator. *SIGARCH Computer Architecture News*. 2011;39:1-7.

8. Homayoun H, Kontorinis V, Shayan A, Lin TW, Tullsen DM. Dynamically heterogeneous cores through 3D resource pooling. *IEEE International Symposium on High-Performance Computer Architecture;* 2012:IEEE.

9. Kumar R, Jouppi NP, Tullsen DM. Conjoined-core chip multiprocessing. *Proceedings of the 37th Annual IEEE/ACM International Symposium on;* 2014:Microarchitecture.

10. Kim C, Sethumadhavan S, Govindan MS, Ranganathan N, Gulati D, Burger D, Keckler SW. Composable lightweight processors. *40th Annual IEEE/ACM International Symposium on;* 2007:Microarchitecture.

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

12. Annavaram M, Grochowski E, Shen J. Mitigating Amdahl's Law through EPI throttling. *SIGARCH Computer Architecture News*. 2005;33:298-309.

13. Bahar RI, Manne S. Power and energy reduction via pipeline balancing. *Proceedings of the 28th Annual International Symposium on;* 2001:Computer Architecture.

14. 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;* 2000.

15. Albonesi DH, Balasubramonian R, Dropsbo SG, Dwarkadas S, Friedman EG, Huang MC, Kursun V, MAgklis G, Scott ML, Semeraro G, Bose P, Buyuktosunoglu A, Cook PW, Schuster SE. Dynamically tuning processor resources with adaptive processing. *Computer*. 2003;36:49-58.

16. Manne S, Klauser A, Grunwald D. Pipeline gating: speculation control for energy reduction. *Proceedings of the 25th Annual International Symposium on;* 1998:Computer Architecture.

17. Srinivasan ST et al, Rajwar R, Akkary H, Gandhi A, Upton M. Continual flow pipelines. *Proceedings of the 11th International Conference on;* 2004:Architectural Support for Programming Languages and Operating Systems.

18. Lebeck AR, Koppanalil J, Li T, Patwardhan J, Rotenber E. A large, fast instruction window for tolerating cache misses. *Proceedings 29th Annual International Symposium on;* 2002:Computer Architecture.