Phase-based Dynamic Instruction Window Optimization for Embedded Systems

Tosiron Adegbija
Department of Electrical and Computer Engineering
University of Arizona, Tucson, AZ, USA
tosiron@email.arizona.edu

Ann Gordon-Ross
Department of Electrical and Computer Engineering
University of Florida, Gainesville, FL, USA
ann@ece.ufl.edu
Also affiliated with the NSF Center for High-Performance Reconfigurable Computing (CHREC) at UF

***Abstract*—Even though much previous work explores adapting instruction queue (IQ) and reorder buffer (ROB) sizes to application requirements, traditional IQ/ROB optimizations may be prohibitive for resource-constrained embedded systems, due to the hardware/execution time overheads. We propose low overhead, *phase-based instruction window optimization* to dynamically vary IQ and ROB sizes for different execution phases based on the applications' variable execution characteristics. Results show that our methodology reduces both the average execution time and energy consumption by 23%, compared to a base system with fixed IQ/ROB sizes.**

***Keywords*—dynamic optimizations, low-power embedded systems, instruction window optimization, phase-based tuning, out of order execution**

I. Introduction and Motivation

Due to consumer demands and technology advances, increasingly capable embedded systems (e.g., smartphones and tablets) have become ubiquitous and are expected to execute algorithmically complex and memory-intensive applications. Designing embedded system microprocessors that are equipped to execute these high-demand applications is challenging due to embedded systems' intrinsic design constraints, such as low power, small area, real-time deadlines, etc. These challenges are exacerbated by the adversarial nature of different design objectives (e.g., minimizing power, area, and maximizing performance/reducing execution time). For example, improving performance by incorporating a more complex cache hierarchy could increase power consumption and area.

Additionally, similar to general purpose processors, embedded systems are also impacted by the speed discrepancy between the processor and main memory—the well-known memory wall. An embedded system's performance is significantly impeded when there is a last level cache (LLC) miss, which imposes delays due to long memory access latencies. Thus, much research emphasis has been placed on system optimizations that ameliorate the performance degradations imposed by LLC misses.

Out-of-order (OoO) execution is an execution paradigm that enables the use of instruction cycles that would otherwise have been wasted due to execution delays, caused by long memory access latencies, execution of complex instructions, etc., by allowing an application's instructions to execute out of program order. Unlike in-order execution where an execution delay would cause the application to stall until the delay is resolved, OoO instruction window resources, such as the reorder buffer (ROB), instruction queue (IQ), and load-store queue (LSQ), allow independent instructions to be executed while the delay is being resolved. These hardware resources reduce the effective memory latency by enabling parallel memory accesses in memory-intensive applications—*memory-level parallelism (MLP)*—and facilitate parallel execution of independent instructions in compute-intensive applications—*instruction-level parallelism (ILP)*.

However, the instruction window resources can constitute significant energy overhead due to these resources' power consumption [7]. Since applications have varying execution requirements, these overheads can be minimized by specializing/optimizing the resources based on applications' unique requirements. In addition, such resource specialization allows the resources to more closely adhere to system design objectives.

To enable this specialization, *dynamic instruction window resizing* reduces the energy overhead imposed by instruction window resources by dynamically adapting the instruction window (IQ, ROB, and/or LSQ) sizes to varying application requirements. Since MLP and ILP have been shown [10] to be the most important application characteristics for dictating an application's required instruction window resources, most previous work used a fine-grained optimization approach that specialized the resources based on changes in the application's ILP and/or MLP. ILP-based (e.g., [7]) and MLP-based (e.g., [13]) techniques adapt the instruction window resources to an application's exhibited ILP and MLP, respectively. However, recent research reveals a new understanding of the applications' MLP and the relationship between MLP and ILP. Thus recent work focuses on MLP-based techniques, since these techniques have been shown to also exploit the application's ILP [10].

We present new analysis to show that, for embedded systems, a better alternative to application-based optimization and fine-grained optimization techniques, such as ILP and MLP-based techniques, is *phase-based optimization.* Application execution can be partitioned into execution intervals, and intervals with similar and stable characteristics (e.g., cache misses, instructions per cycle (IPC), branch mispredicts, etc.) can be grouped as *phases*. Since same-phased intervals tend to have similar resource requirements, phase-based optimization adapts the system resources to an application's distinct phases. To facilitate phase-based optimization, *phase classification* clusters instruction intervals with similar characteristics using methods such as K-means clustering [14].

Since MLP-based techniques impose significant overheads that make these techniques infeasible for embedded systems, we propose *phase-based instruction window optimization* as a viable alternative to MLP-based dynamic instruction window optimization techniques for embedded systems. First, the resizing frequency in traditional MLP-based techniques can impose significant execution time and energy overheads on embedded systems. For example, our analysis of various SPEC2006 [15][1] applications showed that different application phases can have similar MLP, such that resizing the instruction window resources based on MLP changes, rather than

[1]SPEC2006 benchmarks exhibit greater runtime execution variability than typical embedded systems benchmarks (Section V) and provide a more rigorous evaluation of our methodology.

phase changes, can significantly increase execution time and energy. Second, previous MLP-based techniques [10][13] used additional hardware to maintain an application's MLP information, which contain how much MLP could be exploited in the application. While this hardware overhead may be acceptable in general purpose processors, the overhead is significant in resource-constrained embedded systems. For example, based on the reported area of the proposed hardware structure used to store the MLP information in [10], this hardware structure would constitute about 16% hardware overhead if used in the ARM Cortex A9 processor [5].

Our phase-based instruction window optimization methodology obviates the need for the MLP-based technique's additional hardware by significantly reducing how much information is collected at runtime. Rather than resizing the instruction window based on MLP changes, our methodology profiles applications online to determine application phases. Our methodology then determines the instruction window size for each distinct phase using a simple optimization heuristic. The instruction window size is then used for subsequent occurrences of the phase. Since previous work [7] has shown that the ROB and IQ are the most impactful instruction window resources for performance and energy consumption, we focus on these resources in our work. Our phase-based instruction window optimization methodology achieves finer optimization granularity than application-based optimization, but also eliminates the execution time and energy overhead accrued by MLP-based techniques. Additionally, our methodology allows instruction window optimization to be performed concurrently with other hardware optimizations (e.g., cache tuning) at runtime, since the phases classified in our work can be used as the basis for parallel optimization of other hardware resources.

In this paper, we show that the instruction window resources can be optimized on the basis of phase changes, rather than MLP changes in order to minimize the attendant overheads of MLP-based optimizations, while achieving similar optimization benefits. We investigate the impact of the ROB and IQ sizes to give insight into which resource has higher optimization impact, and thus should have more optimization effort. Using insights from our studies, we formulate our phase-based instruction window optimization methodology, which requires minimal designer effort, requires no a priori knowledge of the executing applications, and maintains MLP awareness. Our experimental results reveal that our phase-based instruction window optimization methodology reduces both the average execution time and overall system energy consumption by 23%, compared to fixed instruction window sizes.

II. Background and Related Work

Phase-based optimization specializes system resources to an application's execution phases in order to adhere to design objectives. To enable phase-based optimization, phase classification partitions application execution into instruction intervals, measured by the number of instructions executed, and intervals with similar characteristics are clustered into phases. The relationship between phases and execution characteristics (e.g., IPC, cache miss rates) is well known, however, in this work, we establish for the first time, to the best of our knowledge, the relationship between phases and the MLP, and leverage this insight for our phase-based instruction window resizing methodology.

We broadly categorize previous instruction window resizing techniques as ILP-based and MLP-based techniques. In general, ILP-based techniques monitor an application's ILP during execution and resize the IQ and/or ROB when a change in the ILP is detected. Folegnani et al. [7] proposed an IQ resizing technique that logically resized the IQ based on the contributions of the most recent instructions in the IQ to the ILP. Kucuk et al. [11] proposed a methodology that resized the IQ based on the number of instructions in the IQ. This methodology periodically measured the IPC and increased the size of the IQ whenever there was an IPC reduction. Similarly, Buyuktosunoglu et al. [3] used the number of ready-to-issue entries in the IQ to determine the IQ size. However, these techniques had the potential for performance overheads due to the periodic monitoring of the ILP and the frequency of IQ resizing, since the IQ was resized whenever the ILP changed. Also, these works did not consider the performance impacts of resizing the ROB. Khan et al.'s work [9] is the most similar to ours. The authors proposed a resizing technique based on application phases, where application phases were classified based on the ILP and instruction mix. Even though the authors did not explicitly exploit the MLP, which more recent work has shown to be more important for optimizing instruction window resources, the proposed technique can be complementary to ours, and we intend to explore this synergy in future work.

MLP-based techniques resize the IQ and/or ROB when a change in the MLP is detected. Kora et al. [10] showed the relationship between MLP and ILP, and showed that focusing on exploiting an application's MLP was sufficient for optimizing the IQ and ROB size. Petoumenos et al. [13] proposed an MLP-based IQ resizing technique that resized the IQ based on the number of dispatched instructions between LLC misses. Even though these works exploited the MLP and achieved performance improvements compared to a system with static IQ and ROB sizes, these works were targeted towards general purpose computers, and required significant hardware overhead to facilitate the resizing decisions. Our work significantly reduces these overheads by leveraging the application's execution phases, which also exhibit stable MLP, and dynamically determines the best IQ and ROB sizes for different phases for optimal execution.

III. Application Phase MLP Analysis

Recent studies [10][13] have shown the importance of MLP in instruction window resource performance, especially in the presence of cache misses. MLP allows sequential and independent long-latency memory accesses to be performed in parallel, thus effectively halving the memory access time. To simply illustrate the impact of MLP on performance, consider two long-latency main memory accesses, MM_1 and MM_2, resulting from level two (L2) cache misses. Assuming a main memory access latency of T cycles, without exploiting MLP, the memory accesses occur sequentially and require $2T$ cycles. Alternatively, with MLP, the memory accesses occur in parallel and last T cycles (assuming there are sufficient hardware resources to fully exploit MLP). Thus, for instruction window optimization to be efficient, the optimization methodology must exploit MLP where available (i.e., where there exists independent main memory accesses).

Our phase-based instruction window optimization methodology is based on the premise that the MLP for various application phases remains relatively stable for the duration of each phase. We detected MLP in each application phase using the occurrence of LLC misses [10], which are easily obtainable at runtime using hardware performance counters. Since LLC misses are typically clustered with respect to time, the occurrence of one miss indicates that more misses are likely to occur soon afterwards. We assumed the L2 cache to be the LLC and quantified the MLP in each phase using the average number of L2 cache misses per thousand instructions (*L2MPK*) in each phase. Thus, to analyze the MLP in application phases, we first classified the applications into phases using a traditional phase classification method (Simpoint [8]) and measured the L2MPK changes during each phase's execution.

Fig. 1 depicts the distinct phases' MLPs in L2MPK for SPEC2006 *mcf* and *bzip2*, to represent memory and compute intensive applications, respectively (the trends are similar for other

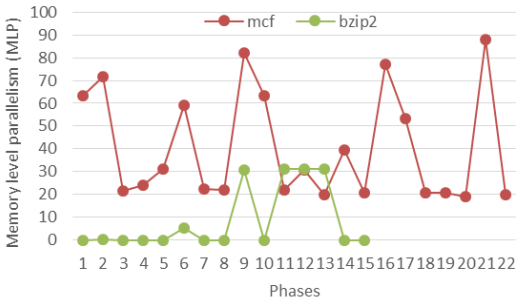


Fig. 1. Distinct phases' MLPs in L2 cache misses per thousand instructions (L2MPK)

applications). As expected, *mcf* exhibits much higher MLP than *bzip2*. In addition, detailed analysis of the applications' phases revealed that phases exhibited stable MLP during their executions. We observed that the MLP did not change as frequently as other execution characteristics (e.g., cache miss rates, branch mispredicts, etc.) across different phases. For example, even though *mcf*'s phases 7 and 8 were classified as different phases due to the phases' different execution characteristics, the MLP was relatively stable across both phases. Similarly, *bzip*'s phases 1 to 5 had relatively stable MLP, even though their execution characteristics were different. Thus, we leveraged these observations while developing our instruction window resizing approach to minimize the frequency of resizing, and minimize the amount of MLP information gathered during runtime.

IV. Phase-based Instruction Window Optimization

We developed our phase-based instruction window optimization methodology using the insights from our application phase MLP analysis (Section III). Since phases exhibit similar MLP levels, and MLP is the most important characteristic for instruction window optimization, our methodology optimizes the IQ and ROB sizes at phase granularity, rather than MLP granularity. Fig. 2 depicts an overview of our phase-based instruction window optimization methodology, which dynamically determines the best IQ and ROB sizes for the application's phases during profiling, and stores these sizes in the *phase history table (PHT)*. The PHT, which can be stored in the SRAM for quick access, is a small data structure that stores previously determined IQ and ROB sizes for the phases' subsequent executions, thus eliminating redundant optimization efforts and minimizing optimization overhead. When a new phase P_i is executed, our methodology uses the optimization algorithm (Section IV.B) to determine P_i's IQ and ROB sizes. The IQ and ROB sizes are stored in the PHT for P_i's subsequent executions, and P_i continues execution using these IQ and ROB sizes. For previously executed phases with similar MLP levels, the IQ and ROB are configured to previously determined sizes as stored in the PHT.

To orchestrate our optimization methodology, we designed a simple hardware *tuner* that uses a hierarchical state machine to

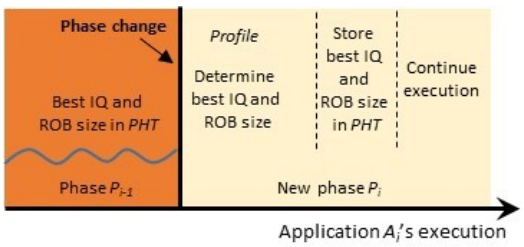


Fig. 2. Overview of our phase-based instruction window optimization methodology

implement our algorithm and control the datapath that performs the energy calculations (details of the state machine and datapath are omitted for brevity). The tuner gathers an application's phases' execution statistics (e.g., IPC, L2 cache miss rates) from the microprocessor's hardware counters [5]. The tuner then calculates each phase's execution time and energy when using different IQ and ROB sizes, as determined by our phase-based instruction window optimization algorithm (Section IV.B), and stores the sizes that achieve the lowest execution time and/or energy in the PHT. The PHT also stores a *phase_benefit* flag to indicate whether or not an application benefits from phase-based optimization, to prevent redundant information from being stored in the PHT (details in Section IV.B). The remainder of this section describes the IQ and ROB resizing techniques we have adopted for our work, our phase-based instruction window optimization algorithm, and our algorithm's computational complexity.

A. IQ and ROB Resizing

To enable low-overhead and efficient IQ and ROB resizing, we assume multi-banked instruction window resources as described in [1][4], where the IQ and ROB's sizes can be configured by gating individual banks. These multi-banked resources are easy to implement and afford low area, power, and access time overheads, which makes the resources suitable for constrained embedded systems. Additionally, to further reduce power consumption, we assume a non-compacting design, such that entries from an instruction issue are not immediately filled, until a new instruction is dispatched into the queue, thus reducing the number of shifts occurring in the queue. We use a 64-entry IQ with eight 8-entry banks and an 80-entry ROB with two 40-entry banks. Thus, our instruction window resources offer 8-, 16-, 32-, and 64-entry IQ sizes, and 40- and 80-entry ROB sizes. Our analysis showed that additional banking for the ROB (i.e., 20-entry ROB) was unnecessary, since the 20-entry ROB did not offer any execution time or energy savings compared to the 40- and 80-entry ROBs. We direct the reader to [1] and [4] for additional details on the circuitry of the resizable IQ and ROB.

B. Algorithm

Algorithm 1 depicts the pseudocode for our phase-based instruction window resizing algorithm that is implemented by the tuner. The algorithm takes as input the array of IQ and ROB sizes, and the base IQ and ROB sizes (line 1), and outputs a phase P_i's best IQ and ROB sizes (line 2). The initial IQ and ROB sizes default to the base IQ and ROB sizes at system startup (line 3-4). For each ROB size, our algorithm increases the IQ size as long as increasing the IQ size decreases the execution time and/or energy. For each configuration, our algorithm executes phase P_i for one tuning interval, and calculates the execution time and energy for the tuning interval to determine the next IQ and ROB sizes to explore (lines 5-16). We used a tuning interval of 1 million instructions, which we empirically determined to be sufficient to obtain stable execution statistics for each phase. Additionally, this tuning interval allows our algorithm to determine each phase's best IQ and ROB sizes within a single execution of that phase due to the execution length of the phases, however, this interval could be increased/decreased with future application requirements with no impact to our methodology. When the ROB size is changed, our algorithm begins with the most recently used IQ and cycles through the array of IQ sizes as long as changing the IQ size decreases the execution time and/or energy (lines 9-15). The best (lowest execution time) IQ and ROB sizes are then stored in the PHT for subsequent executions of that phase (line 17). Since our algorithm targets embedded systems, any IQ/ROB sizes that increase the energy consumption are also discarded. In general, we observed that configurations that reduced the execution time also reduced the energy consumption.

```
1   Inputs: Array of IQ sizes, ROB sizes; base IQ,
    ROB
2   Outputs: Best IQ size, Best ROB size
3   Initial IQ size ← base IQ;
4   Initial ROB size ← base ROB
5   i = 0
6   foreach ROB size
7      if i > 0
8         IQ = IQMRU //IQMRU: most recently used IQ size
9      foreach IQ //cycle starts with current IQ size
10        Execute for one tuning interval
11        Calculate execution time and energy
12        if current_time > previous_time or
13           Current_energy > previous energy
14           IQMRU = IQi-1 //IQi-1: previous IQ
15           break
16        i = i + 1
17  [store IQ size, ROB size in PHT]
```

Algorithm 1. Phase-based instruction window optimization pseudocode

At the end of an application's complete execution, the tuner clears the *phase_benefit* flag if the algorithm selected the same IQ and ROB sizes for all the phases in the application (i.e., no change in these sizes resulted in decreases in execution time savings) and consolidates that application's phases' PHT entries to one entry (i.e., these phases have the same MLP level). *Phase_benefit* = 0 implies that the complete application requires only one set of IQ and ROB sizes, while *phase_benefit* = 1 implies that an application requires different IQ and ROB sizes for different phases. Thus, if *phase_benefit* = 0 for an application, the tuner sets the IQ and ROB sizes at the start of the application's execution and maintains the sizes throughout the application's execution.

C. Algorithm Computational Complexity

Our algorithm determines the best IQ and ROB sizes with worst-case time complexity $O(N)$, where N is the number of phases in the system, resulting in minimal computational overhead and scales well as the number of applications increases. We assume that the phases are previously classified since that is not the focus of our work, thus, this computational complexity does not include phase classification. The comparison of optimization performance to previous work is presented in Section V.C, and the area and power overheads are presented in Section V.D.

V. Experimental Results

A. Experimental Setup

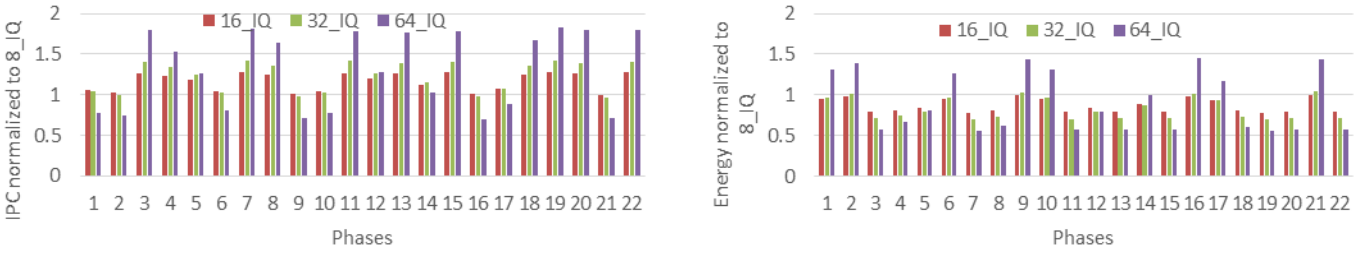
We evaluated our phase-based instruction window optimization's execution time and savings by comparing a system that resizes the IQ and ROB for different application phases (i.e., phase-based optimization) using our methodology to a base system with fixed IQ and ROB sizes. We used the largest IQ and ROB sizes as the base configurations for our evaluations, thus the base configuration was a 64-entry IQ and 80-entry ROB. We modeled an embedded system microprocessor with configurations similar to the ARM Cortex A9 processor in the GEM5 simulator [2] to generate execution statistics for systems with different IQ and ROB sizes. We calculated the energy consumption using McPAT [12], assuming a 32nm TSMC technology with a temperature of 350K. Our energy calculations comprised of the energy consumed by the whole system, including the caches, peripheral component interconnect (PCI) controllers, network interface units (NIUs), etc.

We used twelve benchmarks from the SPEC2006 benchmark suite, cross-compiled for the ARM instruction set architecture (ISA), to evaluate our methodology's execution time and energy efficiency. We used SPEC2006 benchmarks because these benchmarks exhibit greater runtime execution variability than embedded systems benchmarks, which are typically small kernels performing a specific task, and modern embedded systems (e.g., smart phones) execute applications that are similar to general purpose applications. Additionally, previous work [6] has shown SPEC2006 benchmarks to be suitable for evaluating embedded systems due to the current complexity of embedded systems applications. We omitted some of the benchmarks due to cross-compilation errors. However, the evaluated benchmarks comprise a diverse representation of the complete benchmark suite, and thus omission of some of the benchmarks does not affect the quality of the results or our conclusions. We executed each benchmark using the reference input sets for 2 billion instructions after fast-forwarding for 2 billion instructions. We used instruction intervals of 1 million instructions for phase classification, such that intervals with similar characteristics (e.g., cache miss rates, branch mispredicts, IPC, etc.) were clustered to form phases.

A. Impact of IQ and ROB Sizes on Phases

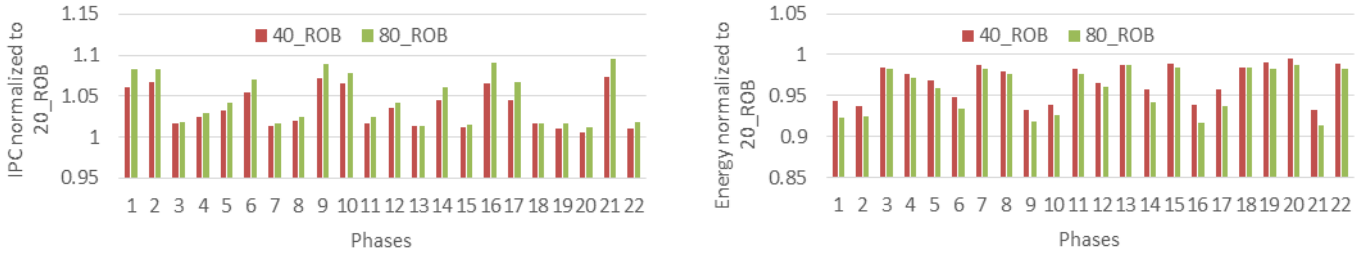
To illustrate the potential impact of IQ and ROB resizing on different phases, we examined the IPC and energy consumed while executing different phases with different IQ and ROB sizes. For brevity, we only show details for *mcf* so that the impacts of IQ and ROB sizes can be clearly evaluated, but note that similar trends exist across all benchmarks.

Fig. 3 depicts the IPC and energy when executing *mcf*'s phases with the 16-, 32-, and 64-entry IQs—denoted as 16_IQ, 32_IQ, and 64_IQ, respectively—normalized to the 8-entry IQ (8_IQ) with a constant 40-entry ROB (the choice of the constant ROB size does not affect the analysis). Fig. 3 (a) shows significant differences in IPC across different phases and different degrees of impact when the IQ size is varied. For example, even though 16_IQ achieved the highest IPC for phase 2, 64_IQ improved phase 3's IPC over 16_IQ by 38%. Similarly, phase 7 showed significant IPC variations with different IQ sizes. Compared to 8_IQ, 16_IQ, 32_IQ, and 64_IQ improved phase 7's IPC by 27%, 42%, and 81%, respectively. Fig. 3 (b) also shows that different IQ sizes have different degrees of impact on different phases' energy consumption. For example, for



(a) *(b)*

Fig. 3. (a) IPC and (b) energy of 16-, 32-, and 64-entry IQ normalized to 8-entry IQ (baseline of one) when executing *mcf*'s phases



(a) *(b)*

Fig. 4. (a) IPC and (b) energy of 40 and 80-entry ROB normalized to 20-entry ROB (baseline of one) when executing *mcf*'s phases

phase 5, compared to 8_IQ, 16_IQ, 32_IQ, and 64_IQ reduced the energy by 16%, 20%, and 19%, respectively. However, for phase 6, 16_IQ and 32_IQ only reduced the energy by 4% and 3%, respectively, while 64_IQ *increased* the energy by 27% compared to 8_IQ. We observed that for phase 6, there was no performance bottleneck resulting from the fetch throughput beyond 16_IQ, since phase 6 had fewer branch mispredicts than the other phases on average. Thus, 64_IQ increased the power consumption without a commensurate reduction in the execution time. These results show that the IQ sizes have significant impact on the phases' executions. While smaller IQ sizes consume less power, smaller IQ sizes could result in significantly more energy consumption due to the significant increase in execution time that could occur when there is a performance bottleneck due to a small IQ size.

Fig. 4 depicts the IPC and energy consumed when executing *mcf*'s phases with the 40- and 80-entry ROBs—denoted as 40_ROB and 80_ROB, respectively—normalized to the 20-entry ROB (20_ROB) with a constant 8-entry IQ (the choice of the constant IQ size does not affect the analysis). Fig. 4 (a) shows that while the IPC varies significantly across different phases, variable ROBs have considerably smaller impact on the individual phases than the IQ. For example, the maximum IPC improvement among all the phases occurred in phase 21, where 80_ROB increased the IPC by 9% compared to 20_ROB. Fig. 4 (b) shows a similar energy trend as the IPC. The maximum energy reduction was in phase 21, where 80_ROB reduced the energy by 9%, compared to 20_ROB. These results reveal that when the designer must prioritize IQ or ROB optimization, methodologies that prioritize the IQ over the ROB will achieve better performance and energy efficiency. For all the phases, 40_ROB and 80_ROB both improved the IPC and energy. 20_ROB did not provide any optimization benefit and could be eliminated in order to reduce the design space, thus reducing optimization overhead.

B. *Execution Time and Energy Savings of Phase-based Optimization*

Fig. 5 depicts the execution time and energy consumption of our phase-based instruction window optimization methodology as compared to the base system with a fixed 64-entry IQ and 80-entry ROB. On average over all the benchmarks, our methodology reduced both the execution time and energy by 23%, compared to the base IQ and ROB sizes. Our methodology outperformed the base system for all the benchmarks, and determined optimal IQ and ROB sizes for the majority of the application phases. The average execution time and energy consumptions were within 1% of the optimal. Our methodology achieved execution time and energy savings as high as 48% and 47%, respectively, for *namd*. For this benchmark, our methodology determined the best IQ and ROB sizes to be the same for all the phases and cleared the *phase_benefit* flag, such that subsequent executions of *namd* were executed with the same IQ and ROB sizes. We observed that even though *namd* had seven distinct phases, the MLP was relatively stable across all the phases. Thus, our methodology determined a single IQ and ROB size for executing all of *namd*'s phases.

To illustrate the ability of our methodology to automatically determine the amenability of executing applications to phase-based optimization, we evaluated the percentage of different applications that were executed using the different possible configurations. Fig. 6 depicts the percentage of the application executions that required different IQ and ROB sizes. The IQ and ROB sizes are depicted as *x-y*, where x and y represent the ROB and IQ sizes, respectively. The graph does not show 40-8, 40-16, and 80-8, since our methodology did not select these configurations for any of the applications' phases. The figure shows that our methodology selected 40-64 most frequently among all the possible configurations, and some applications required just 40-64 for all the phases. However, some applications (e.g., *mcf*) required a variety of configurations for different phases in the applications, due to the variability in MLP levels in the applications' phases. Our methodology was able to detect these variations and determine the appropriate IQ and ROB sizes.

In general, we observed that compared to compute intensive applications (e.g., *bzip2, calculix*), memory intensive applications (e.g., *mcf, libquantum*) had a much higher variation in the exhibited MLP across different phases. This observation can be leveraged in augmenting the optimization algorithm to predict future phases' configurations based on previously executed phases. For example,

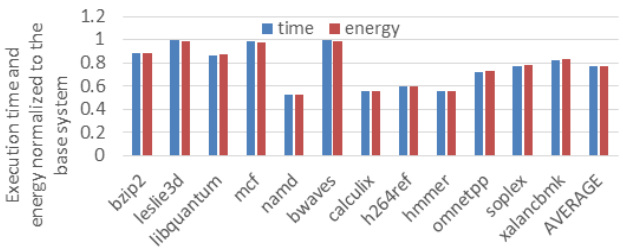


Fig. 5. Execution time and energy consumption compared to the base configuration (80-64)

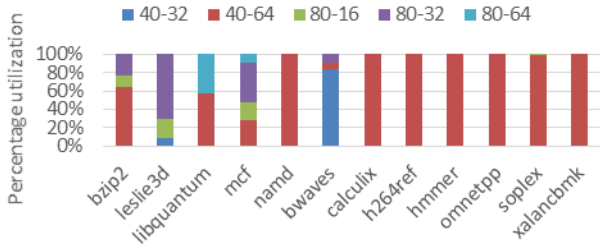


Fig. 6. Percentage of application executions that required different IQ and ROB sizes (sizes denoted as x-y, where x is the ROB size and y is the IQ size)

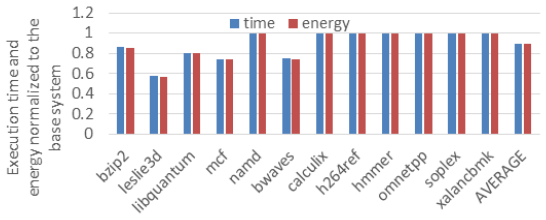


Fig. 7. Execution time and energy consumption compared to 40-64

when a compute intensive application's first phase is executed, the algorithm could predict that the next phase will exhibit similar MLP to the current phase, and determine those phases IQ and ROB sizes to be the current sizes. Those sizes can then be adjusted if necessary after the phases have been executed. While this augmented algorithm may not necessarily improve the energy savings, it will likely reduce the frequency of resizing, thus reducing the optimization overhead.

Since our methodology selected 40-64 for a majority of the phases, we also evaluated our methodology in comparison to a system with a fixed 40-entry ROB and 64-entry IQ. Fig. 7 depicts the execution time and energy consumption of our methodology as compared to a system with 40-64 as the base configurations. On average over all the benchmarks, our methodology reduced both the execution time and energy consumption by 11%, with savings as high as 42% and 43%, respectively, for *leslie3d*. Among all of the benchmarks that required more than one configuration, our methodology achieved average execution time and energy savings of 20%. In summary, our phase-based instruction window optimization methodology successfully identified the best IQ and ROB sizes for efficient execution time and energy optimization.

C. Comparison to Prior Work

We evaluated our work with respect to prior work by comparing our method to the MLP-based approach [10]. Results showed that our phase-based instruction window resizing methodology achieved similar energy optimization results to MLP-based optimization, since the MLP-awareness was also maintained by our technique (graphs omitted for brevity). However, our work significantly reduced the runtime optimization overhead as compared to MLP-based techniques. We measured the runtime optimization overhead by the number of execution cycles required for optimization. Similar to previous work [10], we assumed a resizing penalty of 30 cycles. The resizing penalty is the number of cycles required to change the IQ and ROB sizes by shutting down or switching on individual banks (Section IV.A). We calculated the runtime optimization overhead as *(number of configurations explored – 1) * resizing penalty*. We assumed the worst-case scenarios for our phase-based optimization methodology, where the IQ and ROB sizes are changed for all distinct phases. Our phase-based methodology achieved a 94X runtime optimization speedup on average over all the applications, compared to the MLP-based technique. This speedup was achieved because our approach significantly reduced the resizing frequency by only resizing the IQ and ROB on phase changes rather than on MLP changes.

D. Area and Power Overheads

We have designed the hardware tuner using synthesizable VHDL and Synopsys Design Compiler to quantify the area and power overheads imposed by our work. We evaluated the overheads relative to an ARM Cortex A9 processor, to represent state-of-the-art embedded systems processors. Our tuner imposes 1.2% and 1% area and power overheads, respectively, which represents a significant overhead reduction from previous work. Thus, our phase-based instruction window optimization methodology imposes minimal area and power overheads, and is practical for resource-constrained embedded systems.

VI. Conclusions

In this paper, we presented a low overhead phase-based instruction window optimization methodology that leverages applications' execution phases and dynamically determines the best IQ and ROB sizes for different phases for optimal execution. Our methodology significantly reduces the optimization overheads imposed by MLP-based techniques and involves low computational and implementation complexity, thus making our methodology practical for resource-constrained embedded systems. Experimental results show that our phase-based instruction window optimization methodology reduces both the execution time and energy consumption by 23%, compared to a system with static IQ and ROB sizes. Future work involves extending our methodology to multicore systems with data dependencies among the cores.

Acknowledgements

This work was supported in part by the National Science Foundation (CNS-0953447). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Nation Science Foundation.

References

[1] J. Abella and A. Gonzalez, "On reducing register pressure and energy in multiple-banked register files," International Conference on Computer Design, 2003.

[2] N. Binkert, et al., "The gem5 simulator," Computer Architecture News, May 2011.

[3] A. Buyuktosunoglu, D. Albonesi, S. Schuster, D. Brooks, P. Bose, and P. Cook, "A circuit level implementation of an adaptive issue queue for power-aware microprocessors," ACM Great Lakes Symposium on VLSI, 2001.

[4] A. Buyuktosunoglu, D. Albonesi, P. Bose, P. Cook, and S. Schuster, "Tradeoffs in power-efficient issue queue design," International Symposium on Low Power Electronics and Design, 2002.

[5] Cortex A-9 Processor, http://www.arm.com/products/processors/cortex-a/cortex-a9.php

[6] M. Domeika, "Software development for embedded multi-core systems: a practical guide using embedded Intel architecture," Edition 1, April 2008.

[7] D. Folegnani and A. Gonzalez, "Energy-efficient issue logic," International symposium on Computer Architecture, 2001.

[8] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "SimPoint 3.0: faster and more flexible program phase analysis," Journal of Instruction-Level Parallelism, 2005.

[9] O. Khan and S. Kundu, "A model to exploit power-performance efficiency in superscalar processors via structure resizing," ACM Great Lakes Symposium on VLSI, 2010.

[10] Y. Kora, K. Yamaguchi, H. Ando, "MLP-aware dynamic instruction window resizing for adaptively exploiting both ILP and MLP," International Symposium on Microarchitecture, 2013.

[11] G. Kucuk, K. Ghose, D. Ponomarev, and P. Kogge, "Energy-efficient instruction dispatch buffer design for superscalar processors," International Symposium on Low Power Electronics and Design, 2001.

[12] S. Li, et al, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," International Symposium on Microarchitecture, 2009.

[13] P. Petoumenos, G. Psychou, S. Kaxiras, J. Gonzalez, and J. Aragon, "MLP-aware instruction queue resizing: the key to power-efficient performance," International Conference on Architecture of Computing systems, 2010.

[14] T. Sherwood, E. Perelman, G. Hamerly, B. Calder, "Discovering and exploiting program phases," International Symposium on Microarchitecture, 2003.

[15] SPEC CPU2006. http://www.spec.org/cpu2006.