Enhancing the energy efficiency and performance of a multicore system

Seminar Report

Submitted in partial fulfillment of the requirements

for the course

EE-694

by

Siddhant Singh Tomar

(Roll No. 213079010)

Under the guidance of

Prof. Virendra Singh

Department of Electrical Engineering
Indian Institute of Technology Bombay
September 11,2022

Acknowledgement

I express my gratitude to my guide Prof. Virendra Singh for providing me the opportunity to work on this topic.

Siddhant Singh Tomar
Electrical Engineering
IIT Bombay

Abstract

This report presents a comprehensive literature review focused on enhancing the energy efficiency of a multi-core system (based on ARM's big.LITTLE technology). The report investigates the optimization of single-thread performance through the analysis of "Run-Ahead" execution. The benefits of Run-Ahead execution lie in its ability to capitalize on processor stalls, efficiently generating precise prefetches by pre-executing instructions along the program path.

Contents

List of Figures

Chapter 1

Introduction

1.1 Heterogeneous Core

Unsustainable power consumption and ever-increasing design and verification complexity has driven the micro- processor industry to integrate multiple cores on a single die, or multi-core, as an architectural solution sustaining Moore's law. Beyond the performance, another grand challenge is the energy efficiency of multi-core systems. Heterogeneous multi-core systems—comprised of multiple cores with varying capabilities, performance, and energy characteristics have emerged as a promising approach to increasing energy efficiency[2], big.LITTLE is an example of heterogeneous computing architecture developed by ARM Holdings, coupling relatively battery-saving and slower processor cores (LITTLE) with relatively more powerful and power-hungry ones (big). Such systems reduce energy consumption by identifying phase changes in an application and migrating execution to the most efficient core that meets its current performance requirements. Migration to an efficient core happens at coarser granularity (after millions of instructions). Challenges and opportunities to exploit fine-grained application phases for energy efficiency, by mapping the phase to the most efficient core are presented in chapter 3 of this report.

1.2 Run-Ahead Execution.

The performance of individual cores is equally important and comes with its own set of challenges, one of which is tolerating the latency of the memory subsystem. Conventional latency tolerance techniques are caching, pre-fetching, multi-threading, etc. Today's high-performance, big (OoO) cores tolerate long latency operations using out-of-order execution. However, as latencies increase, the size of the instruction window must increase even faster. The size of an instruction window that can handle these latencies is prohibitively large, in terms of both design complexity and power consumption.[3] Towards this end, Run-ahead execution is an effective way to increase memory latency tolerance in an out-of-order processor, without requiring an unreasonably large instruction window. Run-ahead execution unblocks the instruction window blocked by long latency operations allowing the processor to execute far ahead in the program path. This results in data being pre-fetched into caches long before it is needed, leading to significant performance. Run ahead execution improves memory latency tolerance without significantly increasing processor complexity. Unfortunately, a run ahead execution processor executes significantly more instructions than a conventional processor, sometimes without providing any performance benefit, which makes it inefficient[4]. Run-Ahead execution and techniques for making it more efficient are reviewed in chapter 3 of this report.

Chapter 2

Literature Survey

2.1 Composite Cores.

In big.LITTLE architectures, migration between the cores incurs high overhead, which limits the migration opportunities to coarse-grained phases (hundreds of millions of instructions). Due to this, the opportunities to run the LITTLE core (energy efficient cores at finer granularity (several thousands instructions) where performance difference between big and LITTLE core is comparable, is missed. Composite Cores, an architecture that reduces switching overheads by tightly coupling big and little compute uEngines, is proposed. By sharing much of the architectural state between the u-Engines, the switching overhead can be reduced to near zero, enabling fine-grained switching and increasing the opportunities to utilize the little u-Engine without sacrificing performance.

2.2 Run-Ahead Execution.

With the growing disparity between processor and memory speeds, operations that cause cache misses out to main memory take hundreds of processor cycles to complete execution. Tolerating these latencies solely with out-of-order execution has become difficult, as it requires ever-larger instruction windows, which increases design complexity and power consumption. For this reason, computer architects developed software and hardware prefetching methods to tolerate these long memory latencies. The processor is unable to make progress while the instruction window is blocked waiting for main memory. Runahead execution removes the blocking instruction from the window, fetches the instructions that follow it, and executes those that are independent of it. Runahead's performance benefit comes from fetching instructions into the fetch engine's caches and executing the independent loads and stores that miss the first or second level caches. All these cache misses are serviced in parallel with the miss to main memory that initiated runahead mode, and provide useful prefetch requests. The premise is that this non-blocking mechanism lets the processor fetch and execute many more useful instructions than the instruction window normally permits.

A runahead processor executes significantly more instructions than a traditional out-of-order processor, sometimes without providing any performance benefit. This makes runahead execution inefficient and results in higher dynamic energy consumption than a traditional processor. To reduce the energy consumed by a runahead processor, it is desirable to reduce the number of instructions executed during runahead mode. Unfortunately, reducing the number of instructions executed during runahead mode may significantly reduce the performance improvement of runahead execution, since runahead

execution relies on the execution of instructions during runahead mode to discover useful prefetches. Main goal is to increase the efficiency of a runahead processor without significantly decreasing its IPC performance improvement. Three major causes of inefficiency in a runahead processor: short, overlapping, and useless runahead periods.

Chapter 3

Review

3.1 Composite Cores.

3.1.1 Summary

In this paper, the big (O3)and the LITTLE (InO) share much of the architectural state , to reduce switching overhead and exploit fine grained application phases to save power. Fetch stage(L1 I-cache,branch predictors and I-TLBs) , L1 D-cache and D-TLBs is shared between OoOe and iOe in the composite core. The register file is the only state that must be explicitly transferred during migration. Speculative transfer of registers during "switch". Reactive online controller: Performance estimator which uses simple regression model + performance metrics (L2 miss,L2 hit,Branch mispredictions,active u-Engine cycles ,ILP and MLP) to estimate performance of inactive u-Engine while computing the decision for the upcoming quanta of instructions.

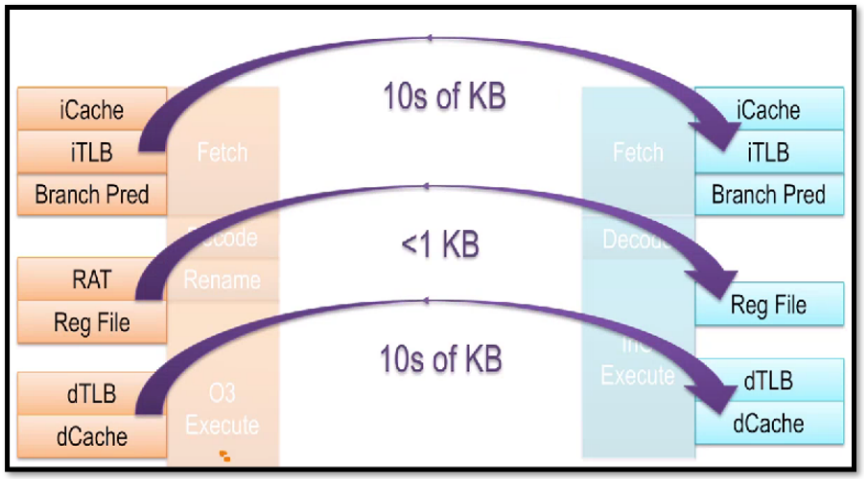

Figure 3.1: Migration Overhead between big and LITTLE engine.

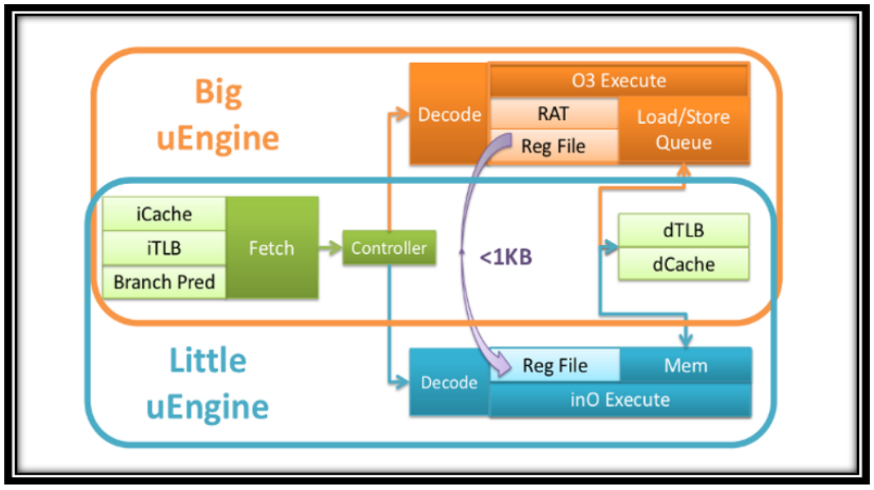

Figure 3.2: Migration Overhead in Composite core.

3.1.2 Strengths

1. To achieve near zero u-Engine transfer overhead, a core micro architecture (Composite core) which shares necessary hardware resources is proposed.

2. low-overhead switching techniques to transfer context/architectural state between the u-Engines

3. Intelligent switching decision logic that facilitates fine- grain switching via predictive rather than sampling- based performance estimation.

4. Switching Between the core is done transparent to OS, reducing kernel overhead.

3.1.3 Weaknesses

1. 20% area overhead incurred by little u-Engine.

2. limits the optimizations for scheduler in kernel, since all the cores are not exposed to scheduler, instead only a single composite core is exposed. Due to which the system should have equal no. of big and little cores instead of Non symmetric configuration for e.g. 2 big cores 4 little cores.

3. All the cores are not exposed to scheduler, therefore all cores can't be exploited for peak performance is required.

3.2 DynaMOS

3.2.1 Summary

The out-of-order core dynamically creates a data flow graph to issue independent instructions and get more ILP. This data flow graph or issue schedule tends to repeat for sequences of instruction with predictable control and data flow. In this work, offloading of recurring issue schedules from the big(OoO) core to the little(InO) core is proposed, by recording and replaying memoizable traces, we can offload more execution from the big core to the energy-efficient little core, without compromising performance. DynaMOS builds on the work on the Composite core. In DynaMOS the little core is provisioned

to work in two modes namely, "InO" and "OinO" mode. In "InO" mode little works like an in-order core where it executes instructions in program order. In "OinO" mode reordered instructions are executed as per big's schedule which was recorded in a schedule trace cache (4KB) by big core by identifying a memoizable trace using a "trace selection table"(0.3KB). "OinO" mode in little core is capable of detecting and resolving false dependencies, and speculatively issuing memory operations. This allows the OinO mode to achieve nearly the same performance as an OoO core, at a fraction of the energy cost.

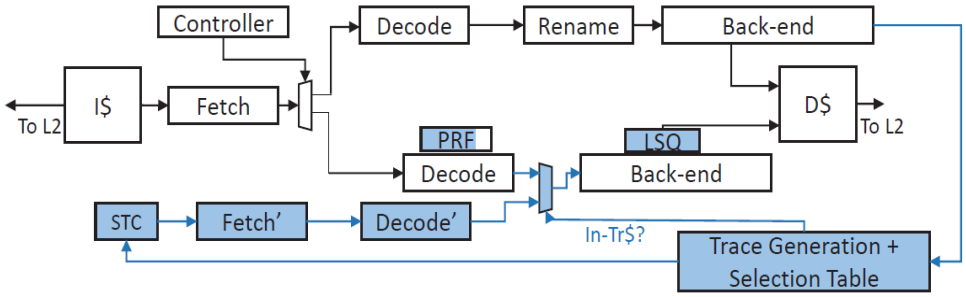

Figure 3.3: little core in DynaMOS [1]. (The shaded portions represent additional resources needed to enable "OinO" mode on little.)

3.2.2 Strengths

1. DynaMOS conserves upto 30% of big's energy, a 2.1x increase from previous work (Composite core).

2. little with "OinO" gains a speedup of 30% over "InO" mode, achieving 82% of big's performance on average.

3.2.3 Weaknesses

1. The entire trace needs to be aborted and re-executed in "InO" mode in case of misspeculation, precise interrupt, and store-to-load aliasing.

2. The constraint that an AR can be mapped to at most a fixed number of PRs force the trace schedule generation algorithm on big to discard many schedules, limiting achievable energy savings.

3. 7.8% of the traces that are finally picked to go on OinO abort, adding a time penalty of 0.3% of total execution.

3.3 MorphCore

3.3.1 Summary

Traditional ACMPs inhibits adaptability to varying degrees of software thread level parallelism. To overcome this limitation, a heterogeneous core is proposed, which acts as traditional 2-way SMT OoO core for single threaded workloads and dynamically "morphs" into

highly-threaded 8-way SMT InO for utilizing TLP in multi-threaded workloads, hence the name "MorphCore", it provides two modes of execution: OoO and InO and is built on two key insights: First, a highly-threaded(i.e., 6-8 way SMT) in-order core can achieve the same or better performance as an out-of-order core. Second, a highly-threaded in-order SMT core can be built using a subset of the hardware required to build an aggressive out-of-order core. In highly-threaded SMT InO mode,common micro-architectural structures like reservation station,ROB,SQ and physical register file are shared on per thread basis. Additional hardware required to support InO mode include changes in front-end for fetching in 8-way SMT and Inorder wakeup and select logic as depicted in Fig 3.6.

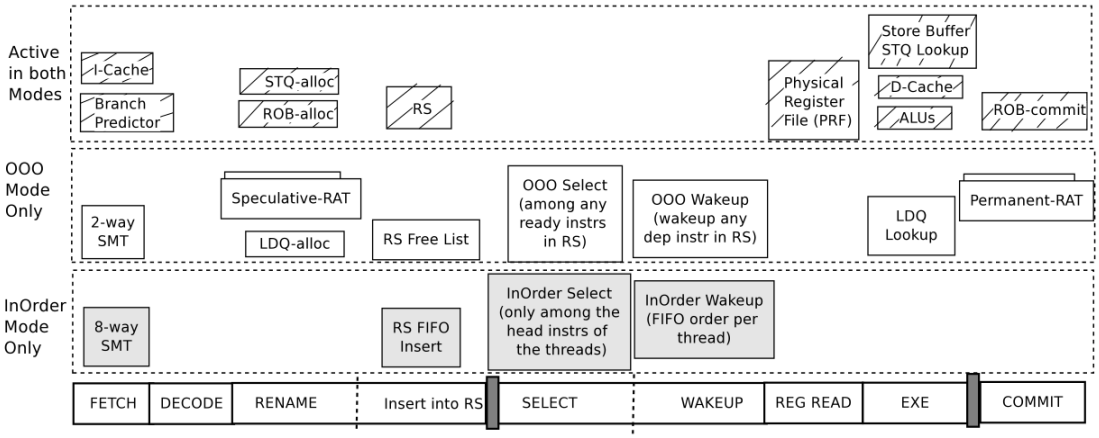

Figure 3.4: MorphCore micro-architecture.

3.3.2 Strengths

1. Morphcore gives high single thread performance and high throughput for multi-threaded code, two competing requirements.

2. MorphCore reduces energy consumption for two reasons: a) MorphCore reduces execution time, thus keeping the core's structures active for shorter period of time, and b) even when MorphCore is active, some of the structures that will be active in traditional out-of-order cores will be inactive in MorphCore's InOrder mode. These structures include the Rename logic, part of the instruction Scheduler, and the Load Queue.

3. Area overhead of morphcore is only 5% as compared to 20% in composite core.

3.3.3 Weaknesses

1. L1 cache miss rate and branch misprediction rate increases in 8-way SMT InO mode.

3.4 Run-Ahead Execution

3.4.1 Summary

The processor enters runahead execution mode when a memory operation misses in the second-level cache or LLC and that memory operation reaches the head of the instruction window. The address of the instruction that causes entry into runahead mode is recorded. To correctly recover the architectural state on exit from runahead mode, the processor checkpoints the state of the architectural register file. For performance reasons, the processor also checkpoints the state of the branch history register and the return address stack. All instructions in the instruction window are marked as "runahead operations" and will be treated differently by the micro-architecture. Any instruction that is fetched in runahead mode is also marked as a "runahead operations".

Each physical register has an invalid (INV) bit associated with it to indicate whether or not it has a bogus value. The first instruction that introduces an INV value is the instruction that causes the processor to enter runahead mode. If this instruction is a load, it marks its physical destination register as INV. Any instruction that sources a register whose invalid bit is set is an invalid instruction. INV bits are used to prevent bogus prefetches and resolution of branches using bogus data.

If both the store and its dependent load are in the instruction window, this forwarding is accomplished through the store buffer that already exists in current out-of-order processors. However, if a runahead load depends on a runahead store that has already pseudo-retired (which means that the store is no longer in the store buffer), it should get the result of the store from some other location. A small runahead cache for this propose. Using the runahead cache to hold the results and INV status of the pseudo-retired runahead stores. The runahead cache is addressed just like the data cache, but it can be much smaller in size, because only a small number of store instructions pseudo-retire during runahead mode. Its purpose is to provide communication of data and INV status between instructions. The evicted cache lines are not stored back in any other larger storage, they are simply dropped. The runahead cache is only accessed by runahead loads and stores. In normal mode, no instruction accesses it.

Branches are predicted and resolved in runahead mode exactly the same way they are in normal mode except for one difference: A branch with an INV source, like all branches, is predicted and updates the global branch history register speculatively, but, unlike other branches, it can never be resolved.

If the branch is mispredicted, the processor will always be on the wrong path after the fetch of this branch until it hits a control-flow independent point. The point in the program where a mispredicted INV branch is fetched the "divergence point." Existence of divergence points is not necessarily bad for performance, the later they occur in runahead mode, the better the performance improvement.

Exit from runahead mode is initiated when the data of the blocking load returns from memory, exit from runahead mode the same way a branch misprediction is handled.

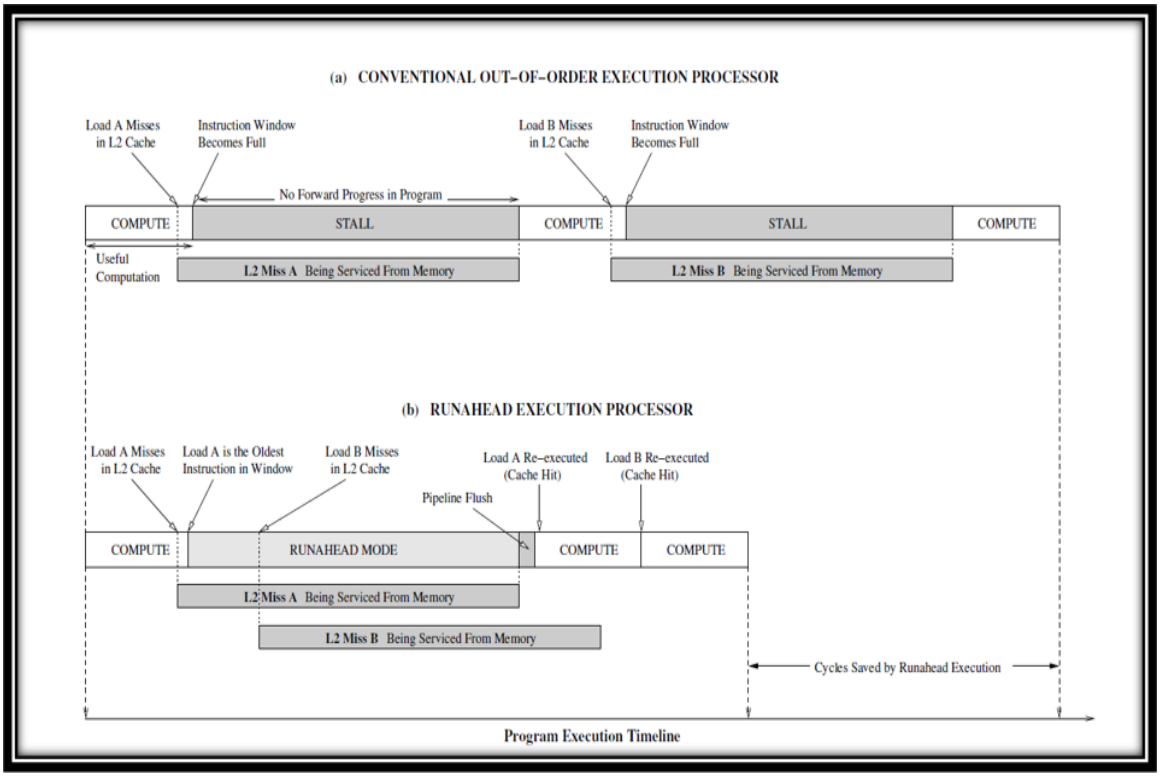

Figure 3.5: Conventional OoOe engine vs Run-Ahead Execution Engine

3.4.2 Strengths

1. Very accurate prefetches for data/instructions (all cache levels) as the program path is followed.
2. Simple to implement, most of the hardware is already built in.
3. Uses the same thread context as main thread, no waste of context.
4. No need to construct a pre-execution thread.

3.4.3 Weaknesses

1. Limited by branch prediction accuracy.
2. Cannot pre-fetch data for load dependent load instructions.
3. Effectiveness limited by available "memory-level parallelism".
4. Prefetch distance (how far ahead to prefetch) limited by memory latency.

3.5 Energy Efficient Run-Ahead Execution.

3.5.1 Summary

Three major causes of inefficiency in a runahead processor: short, overlapping, and useless runahead periods. The following subsections will describe their causes and techniques proposed to eliminate them.

Eliminating Short Runahead Periods

A short runahead period can occur because the processor may enter runahead mode due to an L2 miss that was already prefetched by the prefetcher, a wrong-path instruction, or a previous runahead period, but that has not completed yet. Short runahead periods are not desirable, because the processor may not be able to pre-execute enough instructions far ahead into the instruction stream and hence may not be able to generate any useful prefetches during runahead mode. As exit from runahead execution is costly (it requires a full pipeline flush), short runahead periods can actually be detrimental to performance.
solution:
The processor keeps track of the number of cycles each L2 miss has spent after missing in the L2 cache. Each L2 Miss Status Holding Register (MSHR) [10] contains a counter to accomplish this. When the request for a cache line misses in the L2 cache, the counter in the MSHR associated with the cache line is reset to zero. This counter is incremented periodically until the L2 miss for the cache line is complete. When a load instruction at the head of the instruction window is an L2 miss, the counter value in the associated L2 MSHR is compared to a threshold value T. If the counter value in the MSHR is greater than T, the processor does not initiate entry into runaheadmode, predicting that the L2 miss will return back soon from main memory.

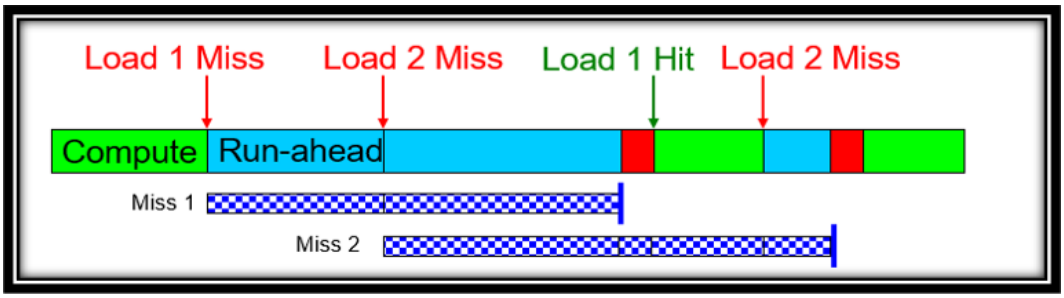

Figure 3.6: Short RunAhead Periods.

Eliminating Overlapping Runahead Periods

Two runahead periods are defined to be overlapping if some of the instructions the processor executes in both periods are the same dynamic instructions. Overlapping periods occur due to two reasons:

1. Dependent L2 misses.

2. Independent L2 misses with different latencies.

solution:
Reducing the inefficiency due to overlapping periods involves not entering a runahead period if the processor predicts it to be overlapping with a previous runahead period. During a runahead period, the processor counts the number of pseudo-retired instructions. During normal mode, the processor counts the number of instructions fetched since the exit from the last runahead period. When an L2 miss load at the head of the reorder buffer is encountered during normal mode, the processor compares these two counts. If the number of instructions fetched after the exit from runahead mode is less than the number of instructions pseudo-retired in the previous runahead period, the processor does not enter runahead mode

Eliminating Useless Run-ahead Periods

Useless run-ahead periods that do not result in prefetches for normal mode instructions are another cause of inefficiency in a run-ahead processor. These periods exist due to the lack of memory-level parallelism in the application program, i.e. due to the lack of independent cache misses under the shadow of an L2 miss. Useless periods are inefficient because they increase the number of executed instructions without providing any performance benefit.

solution:

1. using RCST (runahead cause status table having a 2 bit saturating counter) for maintaing history of a static load instruction, to see if it iniates a useful runahead period.
2. INV load count technique.(count no. of INV Load instruction in a runahead period)
3. Coarse-Grain Uselessness Prediction Via Sampling.(Sample past N consecutive rinahead periods, to make a decision whether to enter into runhead mode in future.)

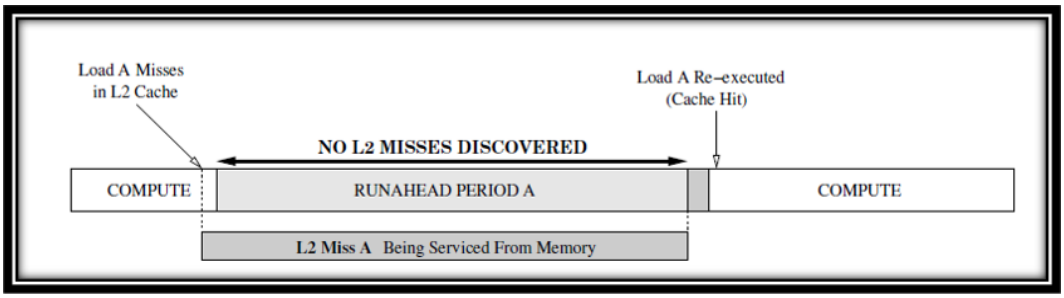

Figure 3.7: Useless RunAhead Periods.

3.5.2 Strengths

1. "Efficient run-ahead execution" has two major advantages:
   - It does not require large, complex, and power- hungry structures to be implemented in the processor core. Instead, it utilizes the already existing processing structures to improve memory latency tolerance.
   - The simple efficiency techniques described in this paper, it requires only a small number of extra instructions to be speculatively executed in order to provide significant performance improvements.
2. The techniques proposed reduce the increase in the number of instructions executed due to runahead execution from 26.5% to 6.2%, on average, without significantly affecting the performance improvement provided by runahead execution.

3.5.3 Weaknesses

1. Entire processor state is released during run-ahead execution, this incurs pipeline refill latency.
2. Instructions not resulting in useful pre-fetches still executed.

References

[1] S. Padmanabha, A. Lukefahr, R. Das, and S. Mahlke, "Dynamos: Dynamic schedule migration for heterogeneous cores," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 322–333, 2015.

[2] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke, "Composite cores: Pushing heterogeneity into a core," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 317–328, IEEE Computer Society, 2012.

[3] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt, "Runahead execution: an alternative to very large instruction windows for out-of-order processors," in *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, pp. 129–140, 2003.

[4] O. Mutlu, H. Kim, and Y. Patt, "Techniques for efficient processing in runahead execution engines," in *32nd International Symposium on Computer Architecture (ISCA'05)*, pp. 370–381, 2005.