ENERGY EFFICIENCY AND PERFORMANCE IN MULTIPROCESSORS SYSTEMS ON CHIP

A Dissertation

by

DAVID KADJO

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Paul V. Gratz
Committee Members,	Jiang Hu
	Rusty H. Harris
	Vivek Sarin
Head of Department,	Miroslav M. Begovic

May 2015

Major Subject: Computer Engineering

ABSTRACT

As process technology shrinks, the transistor count on CPUs has increased. The breakdown of Dennard scaling has led to diminishing returns in terms of performance per power. A trend which promises to impact future CPU designs. This breakdown is due in part to the increase in transistor leakage driven static power. We, now, have constrained energy and power budgets. Thus, energy consumption has to be justified by an increased in performance. Simultaneously, architects have shifted to chip multiprocessors(CMPs) designs with large shared last level cache(LLC) to mitigate the cost of long latency off-chip memory access. A primary reason for that shift is the power efficiency of CMPs. Additionally, technology scaling has allowed the integration of platform components on the chip; a design referred to as multiprocessors system on chip(MpSoC). This integration improves the system performance as the communication latency between the components is reduced.

Memory subsystems are essential to CPUs performance. Larger caches provide the CPU faster access to a larger data set. Consequently, the size of last level caches have increased to become a significant leakage power dissipation source. We propose a technique to facilitate power gating a partition of the LLC by migrating the high temporal blocks to a live partition; Thus reducing the performance impact. Given the high latency of memory subsystems, prefetching improves CPU performance by speculating future memory accesses and requesting the data ahead of the demand. In the context of CMPs running multiple concurrent processes, prefetching accuracy is critical to prevent cache pollution effects. Furthermore, given the current constraint energy environment, there is a need for lightweight prefetchers with high accuracy. To this end, we present *BFetch* a lightweight and accurate prefetcher driven by control

flow predictions and effective address speculation.

MpSoCs have mostly been used in mobile devices. The energy constraint is more pronounced in MpSoCs-based, battery powered mobile devices. The need to address the energy consumption in MpSoCs is further accentuated by the proliferation of mobile devices. This dissertation presents a framework to optimize the energy in MpSoCs. The proposed framework minimizes the energy consumption while meeting performance and power budgets constraints. We first apply this framework to the CPU then extend it to accommodate the GPU.

ACKNOWLEDGEMENTS

I would like to thank Dr. Paul V. Gratz for the opportunity to participate in research with the Computer Architecture, Memory Systems and Interconnection Networks(CAMSIN) research group. His advice and counsel have guided me through my academic journey at Texas A&M University.

I would also like to express my gratitude to my advisory committee members: Dr. Jiang Hu, Dr. Vivek Sarin and Dr. Rusty Harris for their invaluable feedback. Also, thanks to Dr. Daniel A. Jimenez for his constructive assessments during part of this work.

Special thanks to my colleagues of the CAMSIN and TACO(Texas Architecture and Compiler Optimization) research groups for the good and fun times. The enjoyable hangouts helped alleviate the journey through the PhD program.

Thanks to Intel Corporation, the College of Engineering at Texas A&M University and the National GEM Consortium for their financial support in the form of a Doctoral Fellowship.

I would also like to thank my family for their support throughout all these years. Millions of thanks to my fiancee Brianna Ford. Her support through the past few years has been incommensurable.

Finally, I would like to thank God for his blessings and guidance. I, certainly, will not be at this stage of my life without his support.

DAVID KADJO

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Power management has become a critical issue for current and future chip-multiprocessors(CMPs), as Moore's law continues providing increasing transistor count. Esmaeilzadeh *et al*, show that a major driver of unusable or dark silicon in future many-core CMPs is increasing leakage power consumption [21]. They also argue that energy consumption must be justified by increased performance to be practical as VLSI scaling continues. Thus, computer architects have focused on building efficient systems. An example of such an efficiency has been the design of better memory subsystems to provide the CPU faster access to larger data sets. To this end, architects have relied on large a memory hierarchy and data prefetching. Given the end of the Dennard scaling, growing cache size comes at an increasing high cost in terms of power/energy consumption. This constraint also impacts the hardware budgets for prefetchers. It becomes necessary to design lightweight prefetchers with high accuracy.

Another example of the drive for more efficient utilization of transistor in computer designs has been the increased integration of platforms components on chip. This integration results in performance improvement due to a reduced communication latencies between components. It also yields better efficiency in terms of power consumption because of the shorter interconnects [39]. Modern MpSoCs consist of many different processing elements (PE) including multiple CPU cores, graphical processing units (GPU), DSP cores and video accelerators. While this rich set of resources enables mobile devices to be used for a wide range of applications, it also elevates the platform power and need for energy efficiency [9, 60, 8, 24].

This dissertation addresses efficiency in MpSoCs through architectural and sys-

tem level approaches. Techniques to increase performance without impacting energy and to reduce energy without impacting performance are proposed.

1.1 Microarchitectural Techniques

1.1.1 Last Level Cache Static Power Reduction

The leakage power, to a first order, is linearly proportional to the number of transistors in CMOS circuits. In recent years, architects have dramatically increased the size of the last level cache (LLC), in an attempt to mitigate the large off-chip memory access latencies and bandwidth constraints. Since performance degrades significantly with off-chip memory accesses, the goal in LLC design is to provide enough cache to contain the *worst case* application memory footprint. Thus, the LLC has grown to occupy a large portion of chip die area, as much as 30% in recent Intel CMPs [42]. As such, the leakage power within the LLC has become a significant factor in the overall chip power budget. It is therefore imperative to minimize the leakage consumption within the LLC, without impacting performance. Ideally, one would like the LLC to be minimally sized to application footprint, however, as application footprints vary with application and within an application with program phase, it is impossible to decide at design time the optimum LLC size for all applications at all times. While prior work exists in resizing caches via power gating, it is either overly conservative, leaving energy savings on the table, or imposes a heavy performance burden. We propose a novel LLC power-gating scheme that dramatically reduces energy in the LLC, while leveraging temporal locality aware block migration to mitigate any significant performance impact.

Figure 1.1 illustrates how application footprints vary between applications, it shows the miss rate curve for two applications (*bzip2*, *sjeng*) from the SPEC2006 benchmark suite. In the case of *bzip2*, the miss rate drastically decreases as the

LLC size increases to 8MB, staying constant beyond that point. Thus, its memory footprint can be said to be 8MB. Alternately, *sjeng*'s miss rate is nearly oblivious to the cache size; a 32 MB LLC is not large enough to hold the application memory footprint. A cache of size bigger than an application memory footprint provides little or no performance benefit. Conversely, it dissipates additional energy that affects the overall platform energy as cooling mechanisms are activated to keep the temperature down [4]. When running *bzip2* on a CPU with a LLC size of 32MB, 75% of the cache can be disabled. Because *sjeng*'s miss rate is largely unaffected by the LLC size, nearly all the cache can be disabled.

Previous research ([35, 78, 1]) has proposed methods for reducing the leakage power in caches, however, these methods are not well applicable to LLCs. They require time stamp bookkeeping hardware per cache line, leading to a high hardware overhead due to the low access rate in LLCs. Other methods [58, 2] proposed to shutdown partitions of the cache. These techniques, either do little to reduce the leakage power or have a big performance impact. In general, cache blocks are not ordered by their temporal locality within the sets. Simply shutting down a portion of the cache often leads to performance degradation. The performance impact results from increased LLC misses due to locality lost in the partition being shutdown, exacerbated by the bursts of traffic on the memory bus as the dirty blocks from the partition being shutdown are written back to the main memory.

1.1.2 Branch Prediction Directed Prefetching

The current energy constrained environment also impacts hardware budgets for prefetchers. Moreover, in the context of CMPs running multiple concurrent processes, prefetching accuracy is critical to prevent cache pollution effects [20, 73]. Thus, there is a clear need for a light-weight prefetcher with very high accuracy.

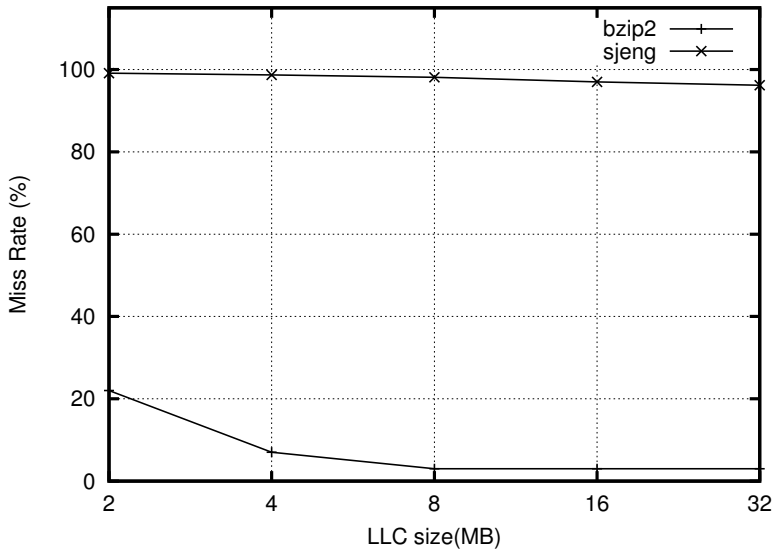


Figure 1.1: Last level cache footprints for sjeng and bzip2.

In this work we present a novel, high-accuracy, low-overhead prefetcher for use in chip multiprocessor designs with shared, last-level caches (LLCs). This prefetcher leverages control-flow speculation, together with effective address value speculation to efficiently provide an accurately predicted stream of future memory references.

Prefetching is a well known and deeply studied technique in which a hardware mechanism attempts to fill the cache with useful data ahead of the actual demand load request stream coming from the processor. In effect, a perfect prefetcher could make all memory accesses complete as if they were first level cache hits. Figure 1.2 shows the speedup that might be achieved under such a *Perfect* L1 D-cache prefetcher normalized against the same system without prefetching for a set of SPEC CPU2006 benchmarks. The *Perfect* prefetcher achieves a geometric mean speedup of $\sim$2X versus no prefetching. For comparison, we also show two other current, light-weight prefetchers, *Stride* [14] and *SMS* [63]. We see that while these prefetchers can provide a significant performance gain, there is room for improvement. We note that in

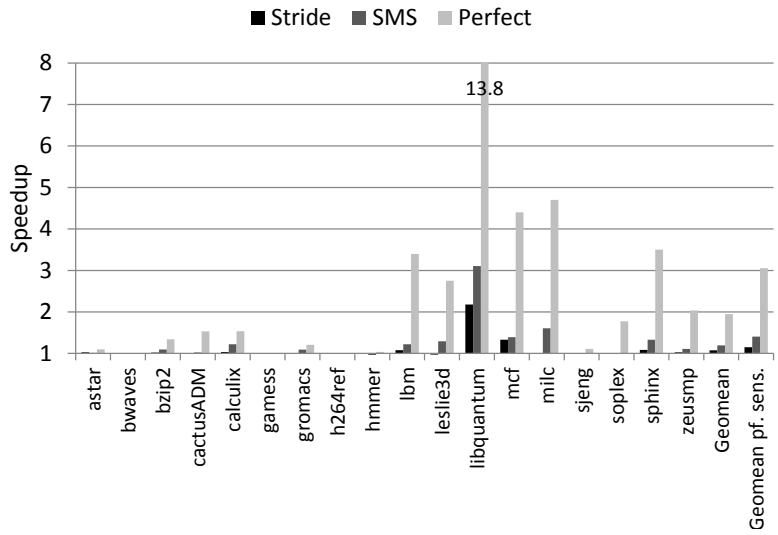


Figure 1.2: Speedup comparison between the *Stride*, *SMS*, and *Perfect* Prefetchers.

the figure, several benchmarks see no gain from the perfect prefetcher, these benchmarks are largely L1 cache resident. To focus on the benefit that can be provided with prefetching, we also show the mean across these *prefetch sensitive* benchmarks, denoted as *geomean pf. sens.*.

Many data prefetching techniques have been proposed over the years, however, most existing prefetchers predict future accesses based on current cache misses. For example, sequential prefetchers prefetch the lines sequentially following the current miss [61], stride prefetchers prefetch lines that exhibit a strided pattern with respect to the current miss [14], and region-based prefetchers prefetch a set of blocks around the miss [63]. These techniques are light-weight, energy-efficient and work well for regular memory accesses, however, they tend to be inaccurate for applications with irregular access patterns. More recent prefetchers attempt to address prefetch accuracy for irregular access patterns [62, 31]. While these methods show significantly improved accuracy, they come at a very high cost in storage overhead, either requiring huge structures to record the memory access patterns or the reservation of

large amount of off-chip memory for meta-data storage (and the associated, energy consuming shuttling of large meta-data information on and off chip). Under modern constraints on energy and power consumption, it is critical to design efficient, low-overhead prefetching techniques which can address irregular access patterns.

To improve both efficiency and accuracy, we propose to use control flow speculation to feed a prefetch engine. Control flow, *i.e.*, which basic block of instructions is executed and in what sequence, is determined by the direction of branch instructions contained in the path. Each basic block contains a particular set of loads and stores; so, branch instructions directly determine the access patterns of data and can be used to inform a suitable prefetcher. A challenge with this approach is to determine the effective address of a speculative future load given that the register values it is based on are likely to change before the corresponding memory instruction is executed. Our approach builds on the fact register content varies in a predictable manner across a set of basic blocks, even in the case of irregular control flow. Thus it is possible to stitch together the expected transformations of a register across a sequence of predicted basic blocks leading to a given future load instruction. The effective address of that load can then be predicted accurately based on the current architectural state of the register and those predicted transformations. Unlike prior light-weight prefetchers based on current cache misses, this approach has the added advantage that prefetches can be issued for future loads without waiting for an actual miss to occur. Unlike techniques which speculatively continue execution beyond long latency loads [19, 51], our approach is extremely light-weight, with only a small prefetch engine active during operation, rather than the entire core.

1.2 Platform Level Energy Optimization

As mobile platforms have become mainstream, the computational demands on these devices have reached unprecedented levels. Besides a powerful multiprocessor system-on-chip (MpSoC), mobile platforms host major hardware components such as touch screen displays, modems, cameras and GPS modules. The diversity of hardware resources and applications running on mobile devices results in a large variation in the power consumption profile across different usage scenarios, as illustrated in Figure 1.3.

1.2.1 A CPU Approach

Recent empirical data indicates that the CPU no longer dominates the platform power consumption [8, 60]. The major contributors to the power consumption under a typical use case are the CPU, the display, and the power management IC (PMIC) with 30%, 33% and 25% share, respectively [8]. Hence, platform level resource management is necessary to optimize the overall energy consumption. Furthermore, accurate modeling and optimization of platform components is needed due to the non-trivial relation between performance and energy consumption [74, 5, 72, 6, 17, 76].

Traditionally, CPU cores have been the major focus of power management research. Idle power management [49, 3] and dynamic voltage and frequency scaling (DVFS) has been extensively used to minimize power [44, 5, 17, 36, 18]. However, current DVFS policies used in mobile platforms are based on those designed for CPU centric desktops which are not efficient for system level optimization [52]. As a motivational example, we ran a memory intensive workload on a smartphone with Intel© Atom™ Clovertrail SoC. Reducing the CPU frequency from 2GHz to 1.2GHz results in 12% reduction in total platform power while merely increasing the execution time from 44.10 sec to 45.78 sec. Despite this increase in run-time, our

measurements show a platform energy consumption reduction of 9%, which obviously translates to a longer battery life. However, when executing the same workload, the default Android on-demand frequency governor [52], which sets the frequency based on the CPU utilization, maintains the CPU at 2GHz. In other words, this policy fails to detect the poor CPU usage due to the frequent cache misses.

True platform-level power management is possible only when all the platform resources and PEs in the MpSoCs provide hardware support for different sleep and performance states, and these states are exposed to the OS [30]. To date, the CPU cores provide by far the richest set of power states and control knobs exposed at the OS-level. On the other hand, there are very limited number of options to change power states of the other resources such as, GPU and memory at the OS-level. Therefore, we present a general architecture for platform-level power management, but practically illustrate it by focusing on CPU, display and PMIC. We specifically model the impact of PMIC which supplies power to all the components in the platform [27]. Accounting for PMIC efficiency is important since the power dissipated in the PMIC due to DC to DC converters and voltage regulators affect the total platform power.

Multiprocessors systems on chip (MpSoCs) for mobile platforms typically contain integrated CPU-GPU cores to provide a better 3D graphic experience. On these platforms, 3D graphic applications are among the most commonly used applications. Therefore, energy optimization for such applications is important. We extend our framework to optimize the energy savings in 3D graphics applications while meetings performance requirements.

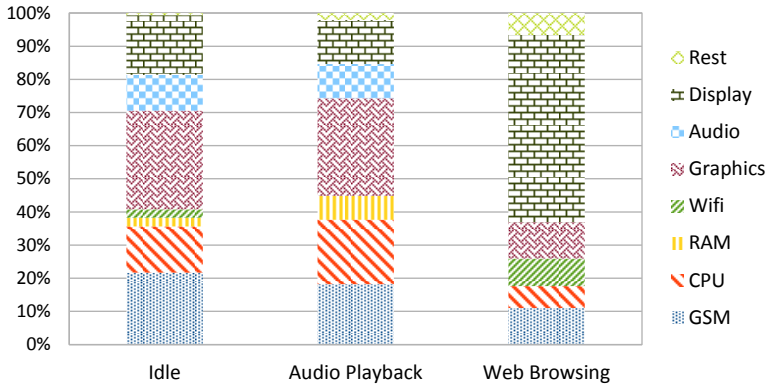


Figure 1.3: Power consumption breakdown for a mobile platform under different usage scenarios [8].

1.2.2 A CPU-GPU Approach

In typical graphic applications, the GPU works in tandem with the CPU; with the CPU generating tasks for the GPU to produce frames for the display. The target Frame Per Second (FPS) is usually constrained (e.g. 60 FPS). At the same time, applications usually vary in terms of their need for the CPU and GPU computational resources to meet the target FPS. This indicates that running the CPU and GPU at their maximum frequency always may unnecessarily burn significant energy for workloads that do not require aggressive resources to achieve the target FPS. On the other hand, running the CPU and GPU at their minimum frequency may not be sufficient to meet the target FPS. Hence, any energy optimization work geared towards graphic applications needs to correctly understand and model the interaction between the CPU and GPU and set their frequency intelligently to meet the desired Quality of Service target (QoS) while burning minimal energy.

Prior techniques for power management on CPU-GPU heterogeneous systems have mostly focused on discrete GPU cards for high performance computing (HPC) [67,

41, 15]. These techniques exploit the partitioning of workloads into tasks and subsequently mapping them onto either the CPU or the GPU. Pathania *et al.*, recently, proposed a technique to reduce the energy in mobile 3D gaming [53]. Their method, however, relies on an heuristic technique based upon an offline analysis, that determines the CPU-GPU frequencies combinations that generate the desired QoS.

We propose a technique that models the interaction between the CPU and GPU as queuing systems. The CPU generates requests for work and insert them into a request queue. The GPU, then, ejects the requests from the queue and executes the commands that build up the frames. A request, generally, contains all the rendering commands for the GPU. The frames are, then, inserted into a frame queue and sampled by the display controller. The CPU frequency determines the injection rate into the request queue. While, the GPU frequency controls the ejection rate from the request queue and the injection rate into the frame queue. Thus, achieving a target FPS requires the synchronized control of both the GPU and CPU frequencies. This synchronization maintains a certain occupancy in the queues. A detailed illustration of the CPU-GPU interaction is presented in Section 4.2.3. Furthermore, we build run-time power and performance models of both the CPU and GPU based upon the information collected by the hardware performance counters. These counters provide insight into the compute engines' behavior. The power models allow the runtime estimation of the power consumption for the purpose of energy optimization, while the performance models allow estimation of the frequency scalability.

1.3 Thesis Statement

This dissertation explores techniques to reduce the efficiency of modern MpSoCs. These techniques can be broken into two basic components. First, we examined micro-architectural technique; We then look at techniques in platform power man-

agement to improve efficiency and performance.

1.4 Dissertation Contributions

This dissertation makes the following contributions:

Last Level Cache Static Power Reduction: We propose a novel method that migrates the high temporal locality blocks, data and tag, to the future live partition; thus mitigating the performance impact of oblivious cache bank shutdown via power gating. Our contribution is as follows:

- We migrate, at random, the high temporal locality blocks from the partition to be shutdown to the future live partition, the size of which is estimated based on the application reuse behavior. The migration is done during a transitional period and the blocks locality are computed through a predictor.

- We, further, develop a cache management policy to increase the cache efficiency. This management policy pseudo-partitions the cache with the high locality blocks residing in the future live partition and the low locality blocks in the partition being shutdown. Thus facilitating the cache resizing via power gating.

Our proposed method is independent of the underlying LLC replacement policy. We show that our approach of intelligent block migration provides significant performance improvement when compared to no blocks migration. We found the LLC leakage power to be of orders of magnitude greater than that of the on-die interconnect therefore the LLC leakage is the focus of our work. We evaluate our technique on sets of PARSEC and SPEC2006 benchmarks.

Branch Directed Prefetching: We propose *B-Fetch*, a combined control-flow and effective address speculating prefetching scheme. *B-Fetch* leverages the high prediction accuracy of current-generation branch predictors, combined with a novel effective address speculation technique to identify prefetch candidates.

- We demonstrate that future memory instruction effective addresses are predictable based on a speculative control flow path from a simple branch predictor. This speculative control flow path is used to feed our *B-Fetch* prefetch engine.

- We propose an effective address value speculation technique based on the current architectural state with learned, per-basic-block variations, to generate effective addresses for the *B-Fetch* prefetch engine.

- We introduce a per-load filtering mechanism to reduce the cache pollution. This technique builds up a confidence estimation for load instructions to determine whether a prefetch candidate is useful or not.

- We show that *B-Fetch* outperforms the best-in-class light-weight prefetcher, Spatial Memory Streaming (*SMS*) [63] by 3.5% for single-threaded workloads (8.5% among prefetch sensitive), and up to 8.9% for multi-application workloads, with 65% less storage overhead than *SMS*.

We evaluate our technique on a set of SPEC CPU2006 benchmarks for both single threaded and multiprogrammed workloads and show performance improvement versus baseline.

Platform Level Power Management: We present a general platform power management framework that minimizes the energy consumption while meeting performance and power budgets constraints. We illustrates it using two concrete DVFS algorithms. The first one minimizes the CPU and platform energy while achieving performance guarantees. The performance target is set by a user level application, and expressed as a fraction of the maximum achievable performance which is measured in the number of instructions executed per sampled period. The second one

dynamically, adjusts the CPU and GPU frequencies to maintain queue reference occupancy and achieve a QoS target while optimizing for the energy. Both techniques account for the PMIC's energy inefficiency and provide a closed loop control by comparing the achieved power and performance against a reference target. The contributions are as follows:

- We develop closed loop controllers that maximize performance with a power constraint and minimize power with a performance constraint, respectively.
- We develop analytical models to predict power consumption and performance at run-time and use them in closed loop controllers.
- We present a queuing model to represent the interaction between the CPU, GPU, and the display.

We implemented the proposed approaches on an Android mobile tablet based on a Atom processor and show platform energy savings over a series of workloads.

1.5 Dissertation Organization

This dissertation is organized as follows. Chapter 2 presents our technique to reduce the power consumption of the last level cache in CMPs. A data prefetcher to increase the performance of CMPs is proposed in Chapter 3. Chapter 4 presents our approach to increase the platform energy efficiency and performance. the background in memory subsystems and their effect on CPU performance. Chapter 5 reviews previous techniques in cache power reduction and prefetching. It, also, reviews prior work dedicated to energy optimization in SoCs. Finally, chapter 6 concludes our dissertation.

2. LAST LEVEL CACHE POWER REDUCTION*

This chapter presents our method to reduce the leakage power in last level cache for CMPs.*We first introduce a background of cache memories. Then, we outline the design of our proposed design. The evaluation along with the analysis of our results are also presented.

2.1 Background

Current memory access latencies are of the order of hundreds of processor cycles. To be effective at masking such high latencies, computer architects have leverage larger caches and data prefetchers.

Caches are SRAM memory structures designed to exploit the locality of reference exhibited in programs. They are the temporal and spatial locality. Temporal locality speculates that a referenced memory address will likely be referenced again in the near future. While the spatial locality states that if a memory address is referenced, nearby memory addresses will likely be referenced in the near future. Caches are designed to contain the application working set. In general, an application miss rate decreases with increasing cache capacity; until a point at which the working set fits within the cache. This is illustrated in Figure 1.1. *Bzip2* miss rate decreases until the cache size is 8MB; then remains constant. At that point, increasing the cache size doesn't affect the application miss rate, hence the performance. Of course this trend is dependent on the application; as we observe that *Sjeng* miss rate is nearly oblivious to the cache size. Ideally, one would want the cache size to fit the

*Part of this chapter is reprinted with permission from *Power Gating with Block Migration in Chip-Multiprocessor Last-Level Caches* by David Kadjo, Hyungjun Kim, Paul V. Gratz, Jiang Hu and Raid Ayoub, 2013, The 31st IEEE International Conference on Computer Design, IEEE Computer Society Washington DC, Copyright [2013] by IEEE Computer Society.

application working set. Caches store lines of data or blocks. A typical size is 64 bytes. The blocks are organized into sets. The number of blocks per set is referred to as the cache associativity. A memory request into the cache is split into *tag*, *index* and *byte offset* (shown in Figure 2.1). The *index* selects the set, the *tag* is checked for match against the selected set and the *block offset* specifies the position of the requested data. A cache hit occurs when a valid block matches the tag of the memory request otherwise it is a cache miss. A cache block also contains addition flags bits to indicate whether the block is "valid" and "dirty". Caches are organized as direct mapped, set associative or fully associative. A direct mapped cache has one block per set (1-way associative); that is a memory address occupies one and only one block. A k-way set associative cache is organized in set of k blocks; a memory address can occupy any of the blocks within the set. A fully associative cache a only one set. As a result, a memory address can occupy any of the blocks.

Given the limited size of the cache, a management policy is used to insure the cache its optimum utilization. A cache management policy can be broken into three policies that are: an insertion policy, a promotion policy and a replacement policy. The insertion policy indicates whether an incoming block is placed within the cache. If so, where it is placed. The promotion policy indicates the status of the hit block. The replacement policy indicates the victim block when an incoming block is placed in the cache. The Least Recently Used (LRU) policy is a common policy used for cache management. The LRU policy maintains a "time stamp" of blocks references within a set. Using the LRU policy, an incoming block is placed within the cache, replacing the block that was accessed the furthest in time. When a hit occurs, the block accessed is promoted to the Most Recently Used (MRU). The least recently used block is selected as a victim block when an incoming block is inserted in the cache.

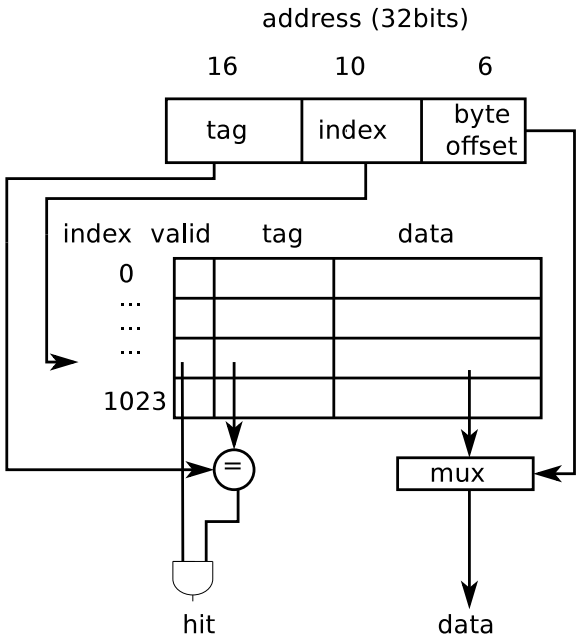


Figure 2.1: Accessing a direct mapped cache.

Cache misses can be classified into four groups that are compulsory, conflict capacity and coherence misses. Compulsory misses occur due to the cache being empty. Every first reference to a memory location results in a miss. Conflict misses occur when multiple memory locations map to the same blocks. Capacity misses occur due the small capacity of the cache; i.e the application working set is larger than the cache size. Coherence misses occur, mainly in CMP contexts, due to the invalidation based coherence protocol when multiple processors write to the same memory location.

2.2 Design

Our proposed technique monitors the application cache footprint and periodically resizes the LLC accordingly. We outline in the following sections the cache size estimation technique and the cache resizing procedures.

2.2.1 Cache Size Estimation

A key problem with dynamically resizing the LLC is quickly identifying the effective cache size for the running application memory footprint. In general, the existence of temporal locality leads to a non-uniform access pattern for the cache's blocks. The LRU (Least Recently Used) algorithm takes advantage of this property to reduce the miss rate. If we construct a histogram that corresponds to the frequency of accessing the different elements in the LRU stack, we observe a monotonic decrease in the frequency distribution where the highest frequency maps to the recently accessed elements. When the cache is underutilized, there is a large variation in the frequency distribution. Hence, the frequency distribution can be used as an indication of the cache utilization by the application [48]. We implement a sampling-based reuse behavior analysis, which decouples from the underlying LLC replacement policy. A set sampler to the LLC is implemented and managed using the LRU replacement policy. This sampler is co-located with the LLC and monitors the LLC traffic to update its sampled tags. The sampler is augmented with counters to keep track of per way hits and total misses as shown in Figure 2.2. These counters allow the construction of the frequency access distribution. A hit to way i causes the corresponding lru^{th} ranked counter, C_{lru}, to be incremented, while a miss to the sampler increments the counter C_{32}. For example, if a block in way 1 is hit and that block is ranked 7^{th} on the LRU stack within its set, the counter C_7 is incremented. Its LRU stack rank is then updated to 0 (Most Recently Used). Equation 2.1 shows the miss rate of a k-way associative partition of the sampler. It is the ratio of the misses to the k-way partition over the total numbers of accesses to the sampler.

$$miss_k = \frac{\sum_{i=k}^{32} C_i}{\sum_{j=0}^{32} C_j} \qquad 1 \leq k \leq 32 \tag{2.1}$$

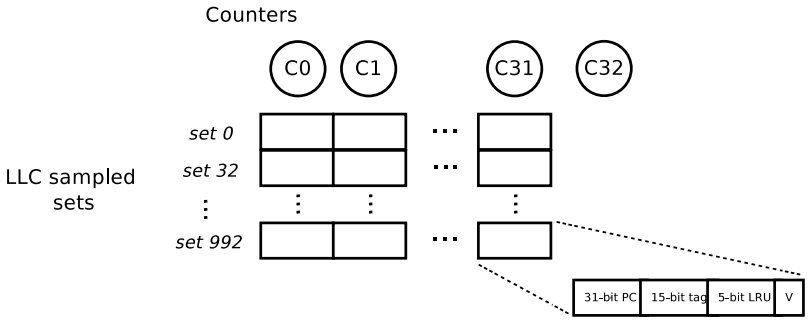


Figure 2.2: High-level block diagram of the set sampler.

We empirically determined the set sampling rate to be $1/32^{th}$ of the LLC as the lowest sampling rate that captures the reuse behavior of the running application with an error less than 10%. As such, the effective cache size can be estimated using the sampler. The sampler associativity is identical to the LLC's. As a result, we estimate the effective cache size for the memory footprint to be the k-way with less than ε impact on the miss rate as shown in Equation 2.2.

$$miss_k \leq (1 + \varepsilon) \times miss_{32} \qquad 0 \leq \varepsilon \leq 1 \tag{2.2}$$

When multiple programs are running, the sampler evaluates the super set of the applications footprints. Since its primary purpose is to estimate runtime cache footprint, it is unnecessary for the sampler to remain coherent with the actual cache set contents. Furthermore, the sampler has a dual purpose serving as a temporal locality predictor.

2.2.2 Cache Resizing

The application execution time is divided into epochs, the anatomy of which is shown in Figure 2.3. Between time t and $t + T_c$, the LLC operates using its default replacement policy. At $t + T_c$, the effective cache is estimated as per Equation 2.2

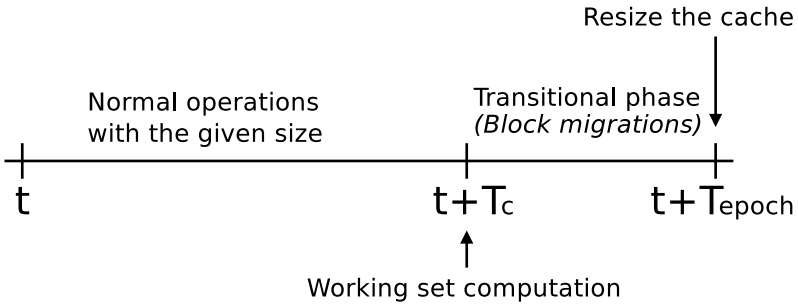


Figure 2.3: Anatomy of an epoch.

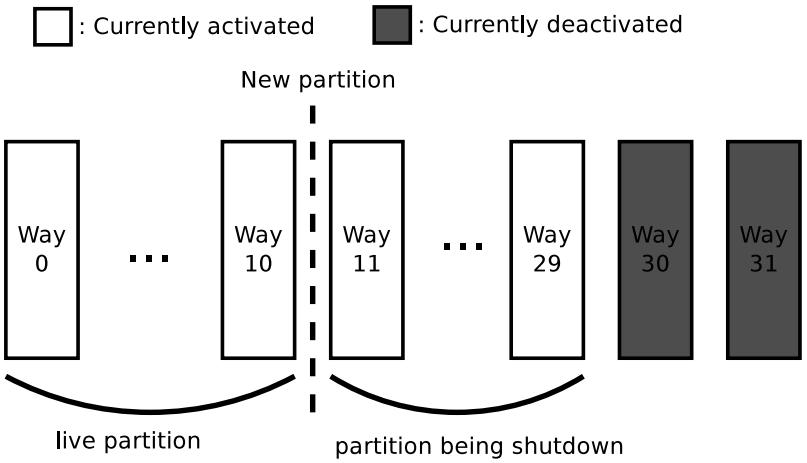


Figure 2.4: Shrinking the LLC size.

using the information gathered by the sampler counters. At time $t + T_{epoch}$, the LLC is resized via power-gating. A transitional phase ($t+T_c$, $t+T_{epoch}$) is used to mitigate the performance impact of shutting down the LLC partition by migrating the useful data from the partition to be shutdown to the partition which will remain live when the LLC shrinks.

2.2.2.1 Temporal Locality Prediction

During a cache block's lifetime, it typically sees one or more bursts of accesses, then it is idle until evicted [71]. We implement a trace based predictor, inspired by

the work of Khan *et al* [37], to determine the blocks temporal locality. The intuition behind this prediction is that if a sequence of instructions leads to the last access of a cache block, the same sequence on another block will likely have a similar effect on that block. The block diagram of the temporal locality prediction is shown in Figure 2.5. The prediction consists of the sampler and a set of 3 saturating counter arrays or locality counters (f, g and h). Each array consists of 4096 2-bit counters; the arrays are indexed with a different hash function($f(PC)$, $g(PC)$, $h(PC)$). A sampler entry encodes the instruction signature (PC) of the instruction that touches a block. When a block is being evicted from the sampler, the indexed counters within the arrays using the corresponding PC are accessed and incremented. Otherwise, the counters are decremented. This phase is referred to as the *update* process. When a request for a block in the LLC is made, the prediction is evaluated and the decision is stored (*query* process). As such, each block in the LLC is augmented with a bit ($PredictionBit$). This bit is set if the block is predicted not useful and cleared otherwise. The predictor is access in parallel with the LLC, thus it does not incur any performance penalty. We define the *usefulness* of the block as the sum of the indexed counters:

$$U(blk) = f(PC_{blk}) + g(PC_{blk}) + h(PC_{blk}) \tag{2.3}$$

The usefulness is compared to a threshold τ to predict the block temporal locality.

$$\text{blk} = \begin{cases} \text{useful} & \text{if } U(blk) \geq \tau, \\ \text{not useful} & \text{if } U(blk) < \tau \end{cases} \tag{2.4}$$

A block is predicted useful if its temporal locality is greater or equal to the threshold; otherwise it is predicted not useful. The blocks predicted not useful are

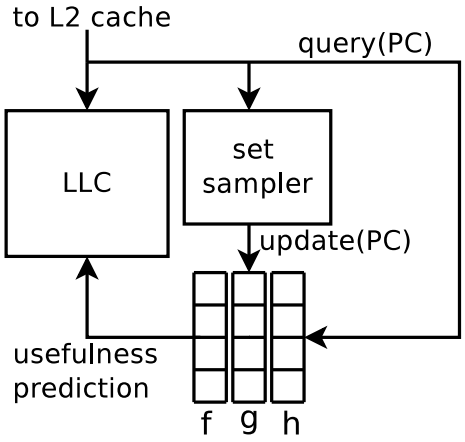


Figure 2.5: Block diagram of temporal locality prediction.

used as victims during the block migration in the transitional phase.

2.2.2.2 Procedure and Cache Management

To increase cache efficiency, the amount of time the blocks contain useful data, we develop a cache management policy to be used during the transitional phase. When the LLC size is being reduced (Figure 2.4), the useful blocks from the partition to be shutdown are migrated to the live partition as follows: Upon a hit in the partition to be shutdown; the data is accessed. At the same time, a prediction for the block is made; if the block is predicted to be useful, it is migrated to the live partition, thereby replacing a not useful block (with its *PredictionBit* set). If no such block is found, the victim block is selected using the default cache replacement policy. If the hit block is predicted not useful, its prediction bit is set. On a hit in the live partition, the data is accessed; if the block is predicted not to be useful, the prediction bit is set. On a miss, if the incoming block is predicted to be useful it is placed in the live block; otherwise it is placed in the partition to be shutdown, this done as remediation in the case of false negatives. At the end of the transitional phase, the cache is pseudo-partitioned with the high temporal locality blocks mostly

in the live partition and the blocks with low temporal locality in the partition to be shutdown. The cache is then resized at the end of the transitional phase. When the LLC is computed to be expanded, the blocks with the *PredictionBit* set are used as victim blocks to increase the cache efficiency. If no such blocks are found, the default LLC replacement policy is used. The LLC shrinking procedure is outlined in Algorithm 1.

Algorithm 1: LLC management during the transitional phase when the cache is being shrunk.

```
On access to the LLC;
if access is a hit then
    Access data;
    if hit in live partition then
        if blk predicted to be not useful then
            PredictionBit ⟵ 1;
        end
    else
        if blk predicted to be useful then
            Migrate blk to live partition, replacing victim blk;
        else
            PredictionBit ⟵ 1;
        end
    end
else
    if incoming blk is predicted useful then
        placed in live part;
    else
        placed in part being shutdown;
    end
end
```

When writing back the dirty blocks from the partition to be shutdown, we avoid bursts of writebacks at the end of the epoch by utilizing a writeback policy similar

to that proposed by Lee *et al* [43]. We spread the writebacks during the transitional phase, speculatively writing back dirty blocks when the memory bus is idle. No invalidation is sent to higher level caches because our memory hierarchy is non inclusive. The switching latency of the power gate transistor is 2 cycles. It does not impact the application performance. When an additional cache partition is powered up, it is accessed at the end of the transitional period, thousands of cycles later; which is much greater than the switching latency. At the end of the epoch, the sampler counters C_i are halved to keep track of the program phase.

2.2.3 Hardware Implementation

Our design consists of three main hardware components:

- A sampler, co-located with the LLC, encodes the PC of the last instruction that touches a block. A set of counters that captures the application temporal reuse behavior.
- An array of counters to determine the temporal locality of the cache blocks.
- The power gating, *gated-*V_{dd}, circuitry used to turn on/off the cache partitions.

The hardware storage overhead cost relative to the LLC is shown in Table 2.1. We assume that the hardware storage is proportional to the area. As such, the area overhead is estimated to be 3.64% of the LLC.

2.3 Evaluation

In this section, we discuss the methodology used to evaluate our technique, the energy cost associated and our experimental results.

Structure	Per Entry	Per Set	Total	Overhead
Sampler	52 bits	1664 bits	6.5KB	0.3071%
Counters	32 bits	1024 bits	1024 bits	0.0059%
Locality counters	2 bits	6 bits	3KB	0.1417%
Prediction bits	1 bit	32 bits	4KB	0.1890%
Gated-Vdd circuitry				3%
Total overhead percentage				3.6437%

Table 2.1: Hardware storage overhead.

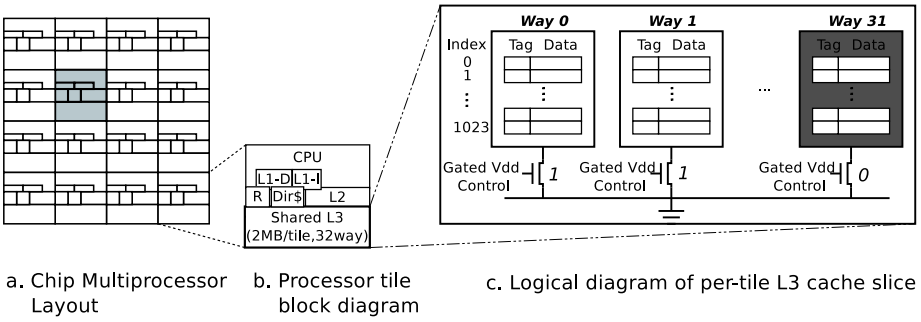


Figure 2.6: Conceptual system overview.

2.3.1 Methodology

We modified gem5 [7], a cycle accurate simulator, to accommodate the additional hardware. Our CPU baseline model is a 16-core (4x4) CMP. Each tile consists of a processor, private L1/L2 caches. The LLC is shared among all the cores, with each one having a 2MB slice as shown in Figure 2.6. Our LLC baseline mirrors the Intel Core 7 with 2MB of last level cache per core. We scale up the LLC configuration with cores count. Addresses are interleaved among the slices. Our simulator has a non-inclusive cache hierarchy; its configuration is shown in Table 2.2. We explored a range of configuration parameters and the ones shown in Table 2.2 represent the best setting. These parameters are, $T_{epoch} = 10$ millions cycles, $T_c = 0.7 \times T_{epoch}$, ε

= 0.1 and the usefulness threshold $\tau = 8$. The default replacement policy for the LLC is LRU.

Core	8-issue out of order, 2GHZ
L1 I-cache	32KB, 64B line, 8-way, 2 cycles latency
L1 D-cache	32KB, 64B line, 8-way, 2 cycles latency
L2 Private cache	256KB, 64B line, 8-way, 10 cycles latency
Last level cache	2MB/core, 64B size, 32-way, 40 cycles latency
Cache Coherence	MOESI
Main Memory	200 cycles
T_{epoch}	10 million cycles
T_c	7 million cycles
ε	0.1
τ	8

Table 2.2: System configuration.

We use a set of SPEC CPU2006 benchmarks for both single-threaded and multi-programmed workloads and the PARSEC benchmarks for multithreaded workloads. For the SPEC benchmarks, we fast-forward 10 billion instructions, warm up for an additional 1 billion instructions then run in detailed mode for the next 2 billion instructions. For fair comparison, when running a single-threaded benchmark, we assume only one slice (2MB) of the LLC is available for the workload use. Thus, the results for a single-threaded workload are compared to a CPU baseline of one core and a 2MB LLC. For multiprogrammed workloads, we use the frequency of access (FOA) inter-threads contention model outlined by Chandra *et al* [10] to select the 20 mixes (*mix1* - *mix20*) of workloads with the highest cache contention. The statistics

are gathered when all of the benchmarks have executed 2 billion instructions. In the case of PARSEC, we run the benchmarks for 2 billion instructions starting from the Region of Interest (ROI), warming up the cache for 1/3 of the ROI. The simulations are performed using the "simlarge" input set with the LLC size of 32MB. We present the performance as the normalized IPC against the IPC achieved with the baseline configuration. In the case of multiprogrammed workloads, we use the normalized system throughput ($\sum IPC_i / \sum IPC_{\text{iconv}}$). In the case of multithreaded workloads, the runtime speedup over the baseline is computed.

Structure	Energy Cost (nJ per access)
Main memory	11.99
Sampler	0.009
Locality counters	0.00293
2MB slice cache	0.408

Table 2.3: Dynamic energy cost of the hardware structures.

2.3.2 Energy Computation

The proposed technique reduces the leakage energy while incurring minimal extra energy consumption as follows:

- Leakage and dynamic energy due to the sampler and array of counters.
- Leakage energy due to the *gated-V_{dd}* circuitry.
- Dynamic energy due to the migration of blocks, data and tag, from the partition being shutdown to the live partition.
- Dynamic energy due to main memory accesses resulting from the writebacks

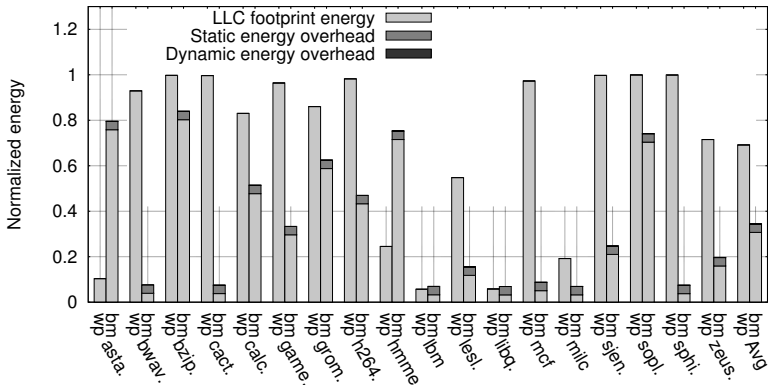


Figure 2.7: Energy dissipation for way partitioning "wp" and blocks migration "bm" normalized to the baseline.

of dirty blocks from the partition being shutdown and the extra misses to the LLC.

In our evaluation, the total energy consumption is computed to be the sum of the leakage energy of the live LLC fraction and the sum of the extra energies listed above. The energy is normalized to the total leakage energy of the LLC in the baseline configuration. The extra dynamic energy due to functional units is ignored because the additional workload execution time is negligible, it less than 2% on average in our experiments. The dynamic energy of the overhead hardware structure is computed as the product of the number of accesses, found through the simulation, and the per-access energy cost, found using CACTI 6.0 [50] configured for 32nm process technology (Table 2.3). The leakage energy of the baseline LLC is estimated to 8.26nJ per cycle per 2MB slice. The leakage energy of the additional hardware is estimated to be 3.64% of the conventional cache as per the discussion in Section 2.2.3.

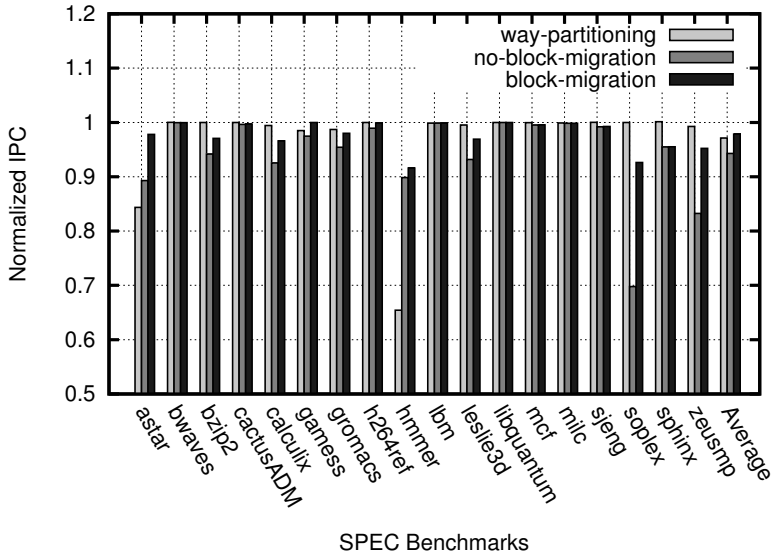


Figure 2.8: IPC comparison for blocks migration and way partitioning normalized to the baseline.

2.3.3 Results and Analysis

We present the power and performance results of our "block-migration" method (*bm*). For comparison, we also evaluate the technique presented by Sato *et al* [58], referred to as *way-partitioning* (*wp*).

2.3.3.1 Single-threaded Workloads

The normalized energy consumption is presented in Figure 2.7. It is broken down into "LLC footprint energy" (the static energy of the active LLC fraction), "static energy overhead" (the leakage overhead due to the additional hardware), and "dynamic energy overhead" (the sum of the dynamic energy due to the extra accesses to the main memory and accesses to the additional hardware and blocks migration). Because Sato *et al* do not estimate the hardware overhead of their "way-partitioning" method, we only present its "LLC footprint energy" thus the comparison is somewhat adverse to "block-migration". The normalized "static energy overhead" is

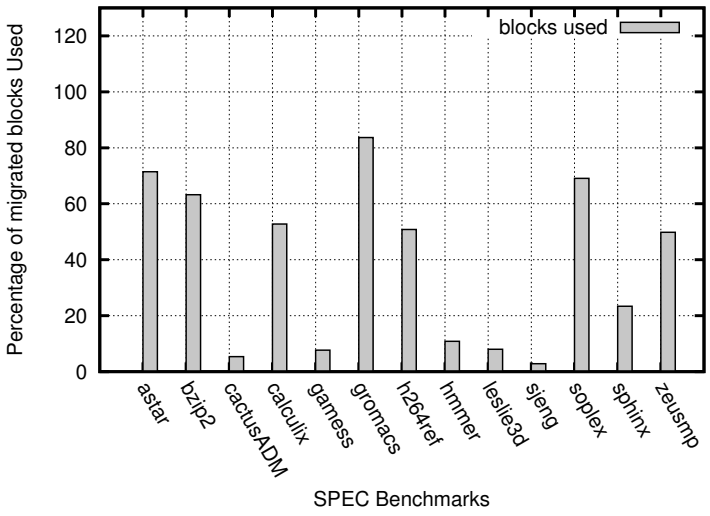


Figure 2.9: Percentage of migrated blocks accessed in the next epoch when the cache is shrunk.

constant across benchmarks at 3.64% while the normalized "dynamic energy overhead" is negligible at 0.032% on average, almost not visible on the charts. Our "block-migration" method achieves an average energy reduction of 66%, versus "way-partitioning" which achieves 30%. *bwaves, cactusADM, lbm, libquantum, milc* either have their memory footprint small enough to fit into the L2 cache or too big to fit into the LLC. As such, our proposed technique shuts down a significant amount of the LLC, 95% for *bwaves*, without performance impact. Not much block migration is performed in those cases; consequently, the performance impact is on par with when no migration is performed (Figure 2.8)."way-partitioning" fails to identify these cases, it allocates 93% of the LLC for *bwaves*. "way-partitioning" locality computation fails to take into account the application overall reuse behavior. Our method shuts down twice as much cache compare to "way-partitioning" with lower performance impact 2.16% versus 2.88% respectively. The energy savings must be examined in conjunction with the performance impact to determine the best trade-off. In cases, such

as *astar* and *hmmer*, "way-partitioning" saves more energy than "block-migration" but significantly overshoots our performance loss allowance. This is due to the fact that the "way-partitioning" fails to take into account the applications entire reuse behavior when estimating their locality. The performance impact without blocks migration ("no-block-migration") degrades significantly, by 5.69% on average. Some benchmarks such as *soplex* and *zeusmp* suffer a large performance degradation of 30% and 17% respectively without block migration. This is due to the lost of high temporal locality blocks when no migration is performed. The "way-partitioning" allocates a much bigger cache size and therefore has a lower performance performance impact. However, at the cost of a higher energy dissipation.

Benchmarks	Error (%)
astar	5.47
bwaves	0
bzip2	4.29
cactusADM	0
calculix	8.19
gromacs	8.46
h264ref	10.91
hmmer	5.11
lbm	0
leslie3d	5.7
libquatum	0
mcf	0
milc	0
sjeng	0
soplex	18.09
sphinx	10.03
zeusmp	8.2

Table 2.4: Cache size estimation error.

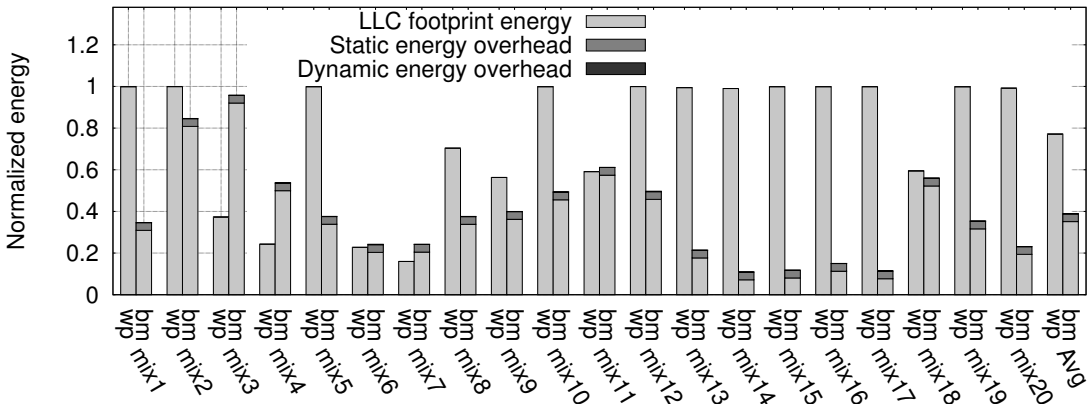


Figure 2.10: Energy dissipation for 2-mix benchmarks using way partitioning "wp" and blocks migrations "bm" normalized to the baseline.

2.3.3.2 Impact of Block Migration

We explore the impact of blocks migration for the "cache friendly" workloads, the ones which footprint occupy a significant amount of the LLC. Figure 2.9 shows the percentage of the migrated blocks which are accessed at least once within the next epoch when the cache is shrunk. We observe that indeed, useful data is migrated to the live partition. *astar, bzip2, gromacs* and *soplex* have a higher temporal locality and therefore have a higher percentage of migrated blocks used within the next epoch. These benchmarks show a higher performance degradation when no block migration is performed. *soplex* suffers a performance degradation of 30%.

2.3.3.3 Effective Cache Size Prediction

Table 2.4 shows the error of our sampling based cache size computation compared to the exact LLC footprint calculated using a stack distance profiling for each set in the LLC. The results show that our sampling methodology is a reasonably good fit for these benchmarks, though the higher error for *soplex* is also correlated with a higher than average IPC loss on that workload.

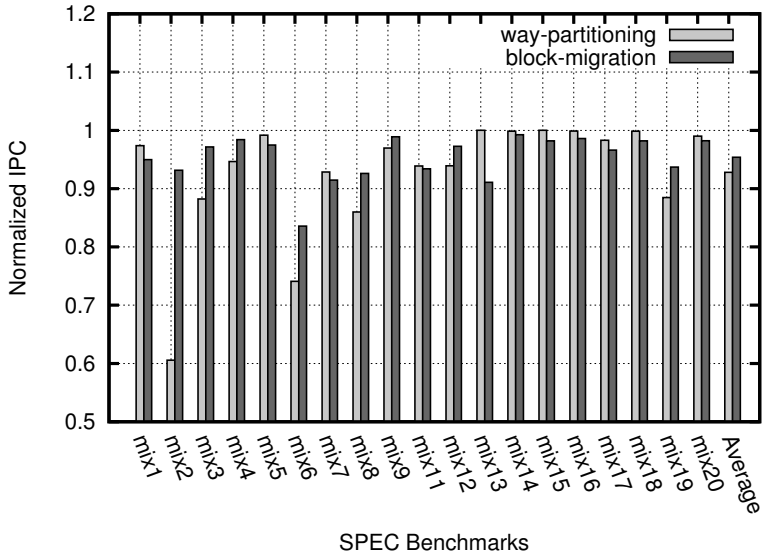


Figure 2.11: IPC for 2-mix benchmarks normalized to the baseline.

2.3.3.4 Multiprogrammed Workloads

Figure 2.10 shows the energy dissipation for with mixes of two single-threaded workloads (*2-mix*). To level the playing field, we scale hardware resources with workload count. Hence, we use 2 cores and 2 slices of 2MB LLC (4MB total). The "way-partitioning" shows inferior power savings compared to our technique (23% vs 65%). We observe comparable system throughput across the workload mixtures. On average, "block-migration" achieves 0.95 normalized throughput versus 0.92 for "way-partitioning"(Figure 2.11). We also experimented with 4- and 8-simultaneous single-threaded workloads, these results not included for brevity, and find the lead of "block-migration" increases versus "way-partitioning" with increasing numbers of simultaneous applications.

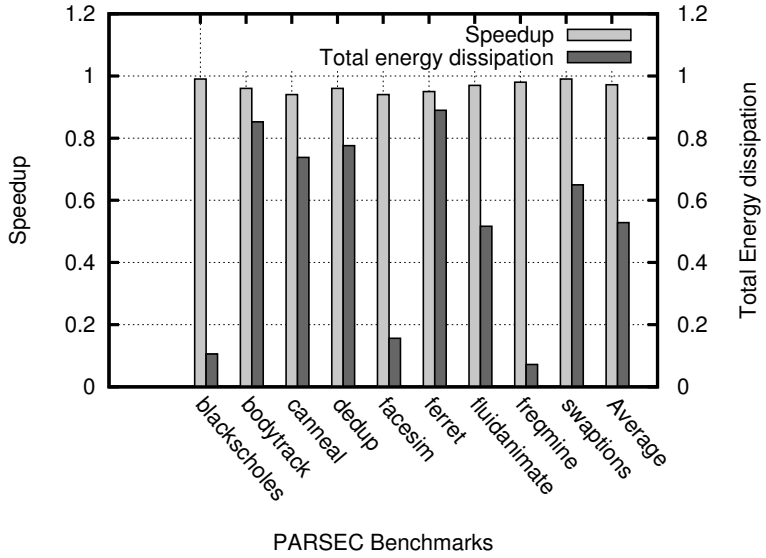


Figure 2.12: Speedup over baseline and energy dissipation for multithreaded benchmarks using blocks migrations.

2.3.3.5 Multithreaded Workloads

The evaluation of our proposed method on multithreaded workloads is illustrated by Figure 2.12. Each multithreaded workload is compiled to generate 16 threads on a 4×4 CMP. Our technique shows leakage energy savings of 50% on average at the cost of 2.79% performance impact. *Blackscholes* has a small input set (2MB) therefore a small portion of the LLC remains turned on (6.89%) with almost no performance impact. Alternately, *freqmine* and *facesim* have much larger working sets size (128MB) compared the LLC size (32MB), thus, our technique allocates a small fraction of the LLC (3.5% and 11.93% respectively). Our technique achieves substantial energy savings at yet affordable performance loss. The "way-partitioning" scheme, confining threads to cache partitions, is not applicable to multithreaded workloads.

2.4 Summary

This section presented a technique to reduce the static power consumption in last level cache for chip multiprocessor. The proposed technique facilitates power gating in cache by migration the blocks with high temporal locality from the partition to be turned off to the partition to be left powered on. We observed energy saving savings up to 66% with low performance impact of 2.16%. The hardware overhead is estimated to be 3.64%.

3. BFETCH FOR CHIP MULTIPROCESSORS*

This chapter presents a branch directed, lightweight data prefetcher to improve performance in CMPs.*A background on prefecthing is provided. The architecture of the prefetcher and details of the proposed design are also presented. In addition, we show evaluation results of our prefetcher compare to current state of the art prefetchers.

3.1 Background

A prefetcher must anticipate misses and issue prefetches far ahead of actual execution. This requires accurate prediction of a) the likely memory instructions to be executed, and b) the likely effective addresses of these instructions. The program execution path (*i.e.* which basic blocks are executed and in what sequence) is determined by the direction taken by the relevant control instructions. Memory access behavior can therefore be linked to prior control flow behavior. For example, consider the assembly code in Figure 3.1, consisting of a set of basic blocks and control flow instructions (branches). The basic block executed following each control instruction depends on the direction taken by that control instruction. Data requested in future execution phases and its access patterns are dependent on the branch outcomes encountered along the path and the per-block register transformations along that path. We therefore propose a lookahead mechanism that predicts the likely path of execution starting from the current non-speculative branch and issues prefetches for the memory references down that path.

*Part of this chapter is reprinted with permission from *B-Fetch: Branch Prediction Directed Prefetching for Chip-Multiprocessors* by David Kadjo, Jinchun Kim, Prabal Sharma, Reena Panda, Paul V. Gratz, Daniel A. Jimenez, 2014, The 47th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society Washington DC, Copyright [2014] by IEEE Computer Society

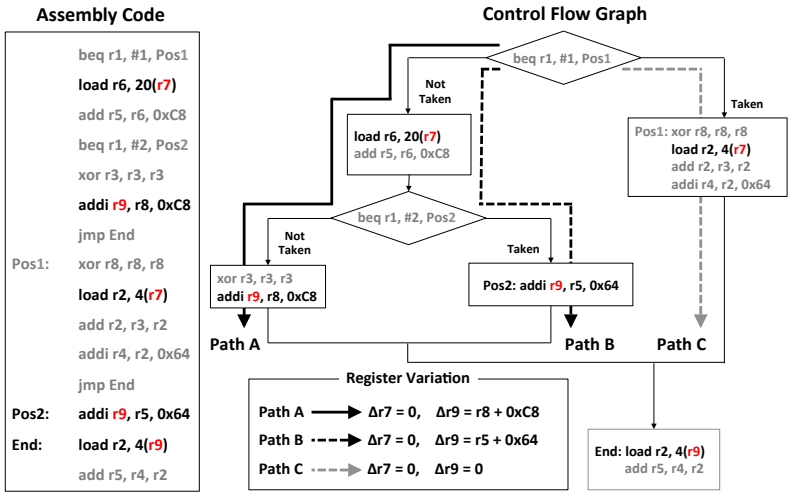
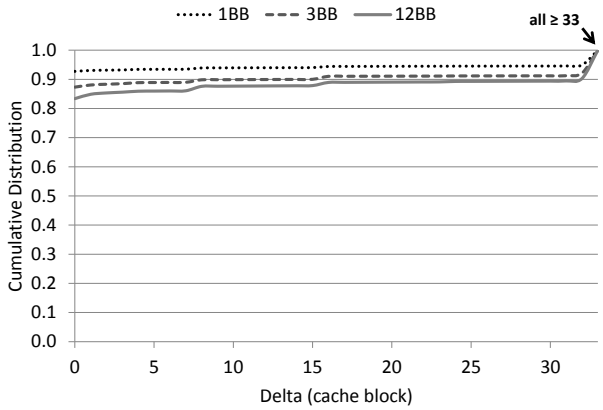
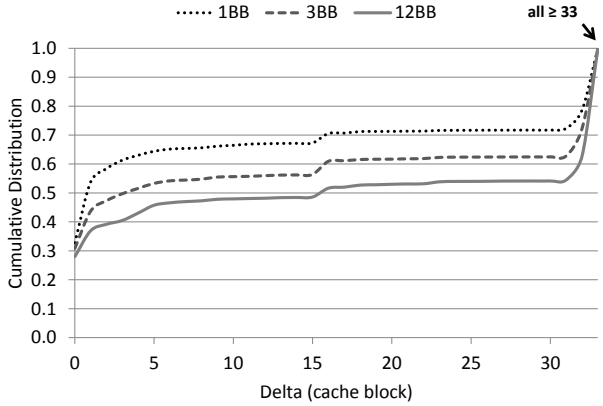


Figure 3.1: An assembly code fragment and its control flow graph equivalent.

To accurately predict effective addresses down the predicted path, we leverage the observation that each memory reference always uses a particular register for effective address computation. Unlike previously proposed prefetching approaches that use history based effective address computation techniques, we propose to associate register indices being used by the memory instructions with their preceding control instructions (the entry points of the basic block) and use this correlation to identify prefetch candidate addresses. For example, consider the **Path C** illustrated in Figure 3.1. Relevant memory instructions and their source registers are highlighted in the figure. In **Path C**, both register r7 and r9 do not change their contents. Thus, if we look ahead along the execution path, the effective memory addresses for load instructions can be predicted by adding up static offsets and register values. The lookahead process can be easily performed leveraging support from the branch predictor in the main pipeline. Meanwhile, if the branch is predicted to take **Path A**, the register content of r9 gets changed by the *addi r9, r8, 0xC8* instruction along this path. In this case, if we record the variation of r9 from preceding branch instructions,



(a) Variation of registers content across execution basic block (BB) expressed in granularity of cache block (64B)



(b) Variation of effective addresses across execution basic block (BB) expressed in granularity of cache block (64B)

Figure 3.2: Cumulative distribution of variation in registers content and effective addresses across execution basic blocks.

the effective address can be calculated by adding up current register value, register variation captured in the previous branches, and static offset value. The novelty of *B-Fetch* lies in exploiting branch prediction and predictable register variation to generate the effective memory address.

B-Fetch is based on the premise that register values at the time of effective address generation are correlated in a predictable way from a) their corresponding

values at a time when their preceding branch instructions were executed, and b) the transformations that occur to them over the course of the blocks to that point. Figure 3.2a shows a cumulative distribution of the register variation (delta) across execution basic blocks (BB), for 1 BB, 3 BB and 12 BB. The variation is expressed at the granularity of a cache block (64B). We observe that for a high percentage (92% in case of 1BB) of registers, the variation falls within 64B. Though that percentage decreases for high number of basic blocks, it remains high (89% for 3BB and 82% for 12BB). By contrast, Figure 3.2b shows the variation of the effective addresses across 1 to 12 BB. Unlike the register content, the effective address varies considerably, particularly as the depth increases to 12 BB. Hence prefetchers which rely upon stable or predictable changes in effective address are less likely to be accurate than those that can incorporate current register values into the prefetch effective address calculation.

3.1.1 Overview

B-Fetch is a data cache prefetcher that employs two speculative components. It speculates on a) the expected path through at future BBs, and b) the effective addresses of load instructions along that path. The first speculation is directed by a lookahead mechanism that relies on branch prediction to predict the future execution path. For the second, *B-Fetch* records the variation of register contents at earlier branch instructions and exploits this knowledge to predict the effective address. By making use of the *variation of register values* rather than the *effective address history*, *B-Fetch* can issue useful prefetches even for instructions that exhibit irregular control flow and data access patterns.

Figure 3.3 illustrates the overall system architecture of a *B-Fetch* enabled out-of-order processor. It shows the main CPU execution pipeline and the additional

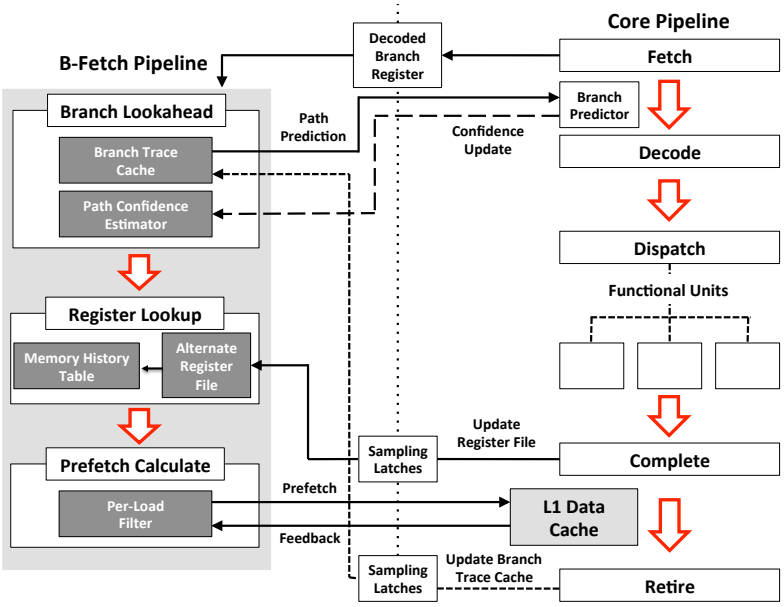


Figure 3.3: Overall *B-Fetch* microarchitecture.

hardware for the *B-Fetch* prefetcher. In our baseline design, the main processor has an out-of-order pipeline with a 4-wide issue width. The *B-Fetch* hardware forms a separate, 3-stage prefetch pipeline parallel to the main pipeline. The *B-Fetch* pipeline is connected to the core's Fetch stage via the Decoded Branch Register (DBR). As branch instructions are decoded in the main execution pipeline, the corresponding PCs are placed into the DBR to initiate the prefetching process. After branch PCs and target addresses are fed into the prefetch pipeline, the *B-Fetch* engine starts to predict future execution path, memory instructions, and their effective addresses. The *B-Fetch* pipeline consists of the following stages:

- **Branch Lookahead:** This stage is responsible for generating the predicted path of program execution starting from the currently decoded branch. This stage also estimates the confidence along that path, stopping when the confidence falls below a given threshold.

- **Register Lookup:** This stage is responsible for capturing and providing information about the registers used to generate effective addresses within a given block.

- **Prefetch Calculate:** This stage is responsible for generating the prefetch addresses that are issued to the prefetch queue, after suitable filtering by a per-load confidence estimator.

3.1.2 B-Fetch Microarchitecture

In this section, we outline the detailed description of *B-Fetch* architecture.

3.1.2.1 Branch Lookahead Stage

The first stage of the *B-Fetch* pipeline, the branch lookahead is responsible for accurately predicting the future execution path. Two primary components reside in the Branch Lookahead Stage. First, the predicted future branches and their target addresses are stored in a small cache called Branch Trace Cache (BrTC). Second, a confidence estimator is used to measure the reliability of the lookahead path and throttle the degree of lookahead when confidence falls below a given threshold.

Starting with a given branch in DBR, each cycle a branch prediction is made and used to look up the PC of the next branch in the BrTC. The confidence of the path to this point is calculated in parallel with the next branch look up. When the confidence falls below a given threshold, the stage will stop predicting further branches.

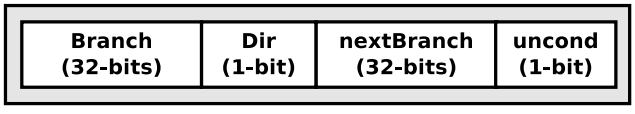


Figure 3.4: Single branch trace cache (BrTC) entry.

Branch Trace Cache (BrTC): The BrTC captures the dynamic control flow sequence of a program and constructs future lookahead paths across multiple BBs. The intent is to enable jumps from one BB to another, skipping all the non-control-flow changing instructions in between. Figure 3.4 shows a single BrTC entry, containing the branch address and direction to get to a given BB as well as the branch at the end of that BB. The BrTC is indexed using a hash of the current branch PC, predicted branch direction, and the target address. Thus, a given branch and direction are connected to the next branch in a direct sequential fashion. Each hash used to index the BrTC, containing branch and predicted direction, is also passed on to the Register Lookup stage. To save space, the lower 32 bits of the 43 bit address are used. We found PC branch aliasing in the BrTC to be highly unlikely.

To cover indirect branches, we include a target address to generate a hashed index. In doing so, we allow BrTC to have more flexibility for assigning entries with different targets. The *B-Fetch* pipeline simply borrows the signal from commit stage and uses it to update BrTC. Thus, the BrTC table is dynamically filled in during runtime and only commit-time updates of the entries are allowed.

Path Confidence Estimator: The path confidence estimator controls the depth of lookahead across BBs by keeping track of the cumulative confidence of the predicted paths. Whenever the cumulative path confidence falls below a threshold value, indicating a likelihood of wrong path prediction, the lookahead process is terminated. This technique avoids prefetching useless data by preventing lookahead down a likely mispredicted path. Note that the prefetch control mechanism based on the path confidence has not been explored in any similar prior work. *B-Fetch* calculates the path confidence using a mechanism similar to that proposed by Malik *et al.* [47]. Additionally, we adopt a composite confidence estimator by combining the JRS, up-down and self-counters proposed by Jiménez [32], to estimate individual branch confidence

values.

3.1.2.2 Register Lookup Stage

The Register Lookup stage tracks the memory instructions in a given BB and the per-BB transformations to their source register values. A pseudo-architectural state copy of the register file contents is maintained in the Alternate Register File (ARF). Register transformations over BBs are tracked in the Memory History Table (MHT), largest structure in the *B-Fetch* pipeline.

Alternate Register File (ARF): The ARF maintains a copy of the register file contents for use in generating predicted prefetch effective addresses. As illustrated in Figure 3.3, the ARF is updated with a sampling-latch delayed copy of execution stage generated register values. This approach insures that *B-Fetch* receives timely updates to its ARF, while not being on the critical path of the execution units in main pipeline. Given that the main pipeline is out-of-order, we maintained ARF consistency by only allowing a register to be updated by an instruction younger than the previous instruction that modified it. In the ARF, each register is augmented with an instruction sequence field to keep track of the modifying instruction order. Despite the possibility of speculative, wrong-path updates to the ARF, we found that this approach provided sufficient accuracy for the purpose of generating prefetch effective addresses, with significant improvement in performance versus a retire-stage, purely architectural-state, register file copy.

Memory History Table (MHT): The MHT maintains source register indices, current register values, and offset values to calculate effective addresses for prefetch candidates. Each entry in the table corresponds to a given BB, and is indexed by the hash of the current branch PC, predicted branch direction, and the target address generated in the Branch Lookahead Stage. Figure 3.5 shows a single MHT entry.

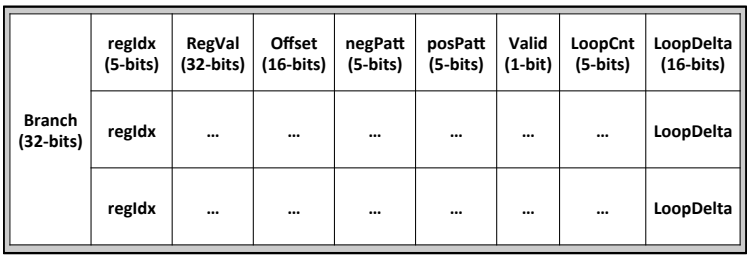


Figure 3.5: Single memory history table (MHT) entry.

Each MHT entry contains a field for the previous *Branch* PC, this is used as a tag to ensure that the hash used to index the entry corresponds to the correct branch. Following, the *Branch* field there are a set of three Register History Entries, each of which containing the following fields: *RegIdx*, *RegVal*, *Offset*, *negPatt*, *posPatt*, *Valid*, *LoopCnt* and *LoopDelta*. A single one of these sets is allocated for each unique register used in generating effective addresses within that basic block. We empirically determined that three Register History Entries was generally sufficient to cover the typical number of registers used in load address computation. More entries consumed more space without significantly improving performance. Each of these fields within these entries is discussed in the remainder of this section along with their function.

The *RegIdx* holds the source register index used for memory address generation in the corresponding BB, linking source registers to the branch that led to the BB. This linkage is learned as control instructions and memory instructions commit in program order in the main execution pipeline. Figure 3.2a indicates that the variation of effective addresses across multiple BBs fluctuates more than an offset of the register value. We observe that even if register values and effective addresses do not match exactly, in terms of variation, they still lie within a fixed offset from each other. The *Offset* field retains these fixed-offset relationships between source register values at a given prior branch and the actual effective address generated at the memory

```
Start: load r1, 24(r2)
       lda r2, r2, #128
       cmpeq r2, r3, r1
Br1:   beq r1, Start
```

Figure 3.6: An ALPHA assembly fragment with a loop.

instruction. The offset value is learned by monitoring the addresses generated by memory instructions in the main pipeline, compared against the stored *RegVal* field. *B-Fetch* generates the effective prefetching address based on the current value of the linked register (*RegVal*) added with the *Offset* value (see Equation 3.2). The *RegVal* is read into the MHT from the ARF.

Whenever a memory instruction executes in the main pipeline, the MHT is indexed using the prior branch PC and the *Offset* is updated. It is computed as the difference between the effective address and *RegVal*. Note that the *Offset* value includes not only the static offset from memory instruction but also the variation of register value itself over the course of that BB. The *Valid* bit indicates whether the entry is valid.

$$Offset = [\Delta RegisterValue] + StaticOffset \tag{3.1}$$

$$PrefetchAddress = [RegisterValue] + Offset \tag{3.2}$$

Loops: In order to efficiently and accurately prefetch for loops, our prefetching algorithm identifies loops and generates prefetch addresses for their future iterations. We make use of our ability to lookahead across future BBs to allow runtime identification of loops. As an example, consider the code fragment given in Figure 3.6.

Assuming that the estimated path confidence is high, the lookahead procedure should yield the following sequence of branch addresses: Br1(Taken) $\longrightarrow$ Br1(Taken) $\longrightarrow$ Br1(Taken), the lookahead depth is determined by the path confidence estimator in the previous stage.

The runtime loop-detection algorithm capitalizes on the idea that if during one complete lookahead process, the same branch is visited more than once, it implies a likely loop in the dynamic instruction stream. The MHT table contains entries to allow the detection and prefetching for loop-based program sequences, allowing dynamic identification of loop based code. The *LoopDelta* field holds the difference between the generated effective addresses over consecutive execution instances of the same instruction. The *LoopCnt* field monitors the iteration count of the loop in the lookahead mode. Equation 3.3 shows the prefetch effective address generation formula as it is implemented to cover loops.

$$\begin{aligned} PrefetchAddress = [RegisterValue] + Offset \\ +(LoopCnt \times LoopDelta) \end{aligned} \tag{3.3}$$

Multiple Loads With The Same Index: The *negpatt* and *pospatt* fields capture the cases when there are, within the same BB, consecutive load instructions off the same source register as a bit vector. Consider a BB with an excerpt snippet of code in Figure 3.7, both load instructions have the same source register and without any modification. The *pospatt* in this case holds the difference between the static offsets. These fields record, at a granularity of cache block, the difference between the static offset of the load instructions whether negative or positive. This approach avoids duplicating entries in the sets of *RegIdx* for loads off the same source register within the same BB.

```
.
load r1, 24(r2)
load r3, 80(r2)
.
```

Figure 3.7: Consecutive loads off the same source register.

3.1.2.3 Prefetch Calculate

In this stage, the prefetch effective address is calculated according to Equation 3.3, using the data pulled out of the MHT and ARF in the previous stage. *B-Fetch* also examines if the associated load address has in the past generated reliable prefetches. If not a filtering mechanism is then used to avoid cache pollution using the Per-load Filter.

Per-load Filter: To avoid cache pollution, wasted bandwidth and energy, it is crucial to reduce the number of useless prefetches. Filtering prefetch requests becomes even more important for systems that prefetch directly into the L1 cache, and those that share an LLC with multiple applications. Conceptually, in *B-Fetch*, the branch confidence mechanism might be thought of as a prefetch filtering mechanism, however in practice we found that even when the branch confidence is high, some loads have effective addresses that are difficult to predict (often within the same BB as others that are predictable). To deal with difficult to predict loads, we implement a per-load PC filtering technique. Our per-load filter measures the confidence of prefetches launched from a given load PC. The filter is inspired by the skewed sampling predictor used in prior work to detect cache dead blocks [38].

The per-load filter consists of three different tables which contain 3-bit up-down saturating counters for corresponding prefetch loads. Each table is indexed, using

the PC of the load instruction, by different hash function and the counter is incremented when the prefetch address turns out to be accurate. If the prefetch address is inaccurate, the counter is decremented. Each access to the filter yields the sum of three counters and constructs a per-load confidence value. The per-load confidence has precedence over the branch confidence. That is, regardless of current branch confidence, if a per-load confidence falls below to a certain threshold, we stop prefetching for that load PC. In order to implement the per load filtering mechanism, each cache block in the L1D cache is augmented with a 10-bit hash of the load PC for the prefetch address and a 1-bit vector to indicate whether the prefetch is useful. These additional bits are accounted for in our storage overhead.

3.1.3 Hardware Cost

The additional hardware storage requirements for *B-Fetch*, as well as *SMS* are summarized in Table 3.1. To optimize *SMS* hardware, we have tested different sizes of spatial regions, accumulation tables, and pattern history tables with the SPEC CPU2006 benchmarks. We observe that the practical *SMS* configuration reported by Somogyi, *et al.* [63] shows the best performance improvement. Unless otherwise specified, we use 2KB spatial regions, a 64-entry accumulation table, and a 16K-entry pattern history table. The filtering table originally proposed by Somogyi, *et al.* is removed because the filtering process can be done by comparing bit vectors in the accumulation table [64]. To optimize storage overhead of *B-Fetch*, we reduce the number of entries in MHT, the largest structure in *B-Fetch*. A sensitivity study for *B-Fetch* with different storage overhead is discussed in the Evaluation section. Because programs typically have more memory instructions than control instructions, a branch-based prefetcher captures the same prefetch candidates at much reduced table sizes.

Prefetcher	Component	# Entries	Size (KB)
B-Fetch	Branch Trace Cache	256	2.06
	Memory History Table	128	4.5
	Alternate Register File	32	0.156
	Per-Load Prefetch Filter	2048	2.25
	Additional Cache bits	-	1.37
	Prefetch Queue	100	0.51
	Path Confidence Estimator	2048	2
	TOTAL SIZE		**12.84**
SMS	Active Generation Table	64	0.57
	Pattern History Table	16K	36
	TOTAL SIZE		**36.57**

Table 3.1: Hardware storage overhead in KB.

We assume that the tournament branch predictor in the main pipeline can maintain a limited number of access per cycle equal to the fetch width (4-wide in the baseline design). Seznec *et al.* present a four-way interleaved ALPHA EV8 branch predictor which supports up to 16 branches per cycle, for a branch predictor microarchitecture similar to the one used here [59]. Figure 3.8 shows the breakdown of the number of branch instructions (both conditional and unconditional) fetched each cycle across 18 SPEC CPU2006 benchmarks. We find that, on average, for more than 99.95% of fetch cycles, a maximum of two branches are fetched per cycle. We observe that fetching four consecutive branch instructions occurs only once per billion instructions. Thus, Figure 3.8 proves that, most of the time, the branch predictor is available to provide data to *B-Fetch* engine without additional hardware. Should this be deemed prohibitive, it would be trivial to include a copy of the branch prediction hardware for use in prefetching, at the cost of some additional state. In addition to the state elements listed in Table 3.1, a number of small adders and control logic is required, however, these components together are insignificant in area and power consumption relative to the arrays enumerated in the Table.

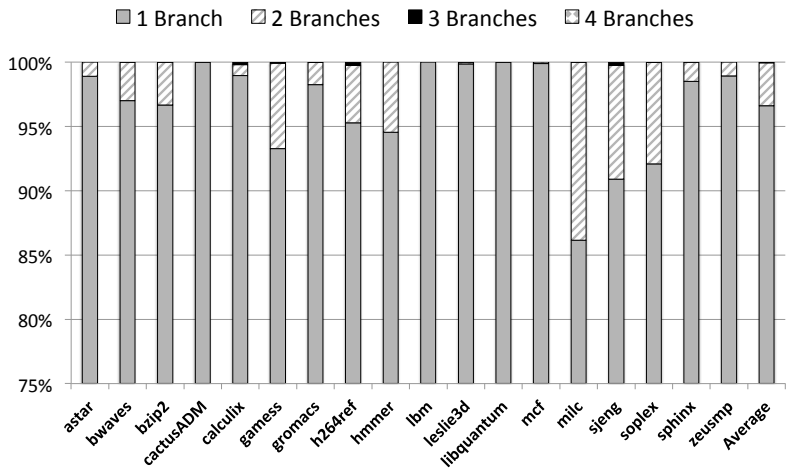


Figure 3.8: A breakdown of the number of branch instructions fetched per cycle.

3.2 BFetch

In this section, we discuss the experimental methodology used to evaluate *B-Fetch*. We compare the results obtained versus previous techniques. A sensitivity analysis of our technique is also presented.

3.2.1 Methodology

We use gem5 [7], a cycle accurate simulator, to evaluate *B-Fetch*. The baseline configuration is summarized in Table 4.1. We use the set of 18 SPEC CPU2006 benchmarks our simulation infrastructure currently supports for single-threaded and multiprogrammed workloads simulations. These benchmarks are compiled for the ALPHA ISA. We fast-forward 10 billions instructions, warmup for an additional 1 billion instructions then run in detailed mode for the next 1 billion instructions. For single-threaded workloads, we assume a 2MB LLC. Thus, our result is compared to a baseline with one core and a 2MB LLC. The memory controller bandwidth is limited to 12.8GB/s which is representative of a memory controller of a x64 DDR3.

We present the performance as the speedup compared to the baseline configuration ($IPC_{\text{B-Fetch}}/IPC_{\text{baseline}}$). For multiprogrammed workloads simulations, we use the frequency of access (FOA) inter-threads contention model proposed by Chandra *et al* [10] to select 29 mixes of workloads with the highest cache contention. The simulation is stopped when all applications have executed 1 billion instructions. The performance is expressed as the normalized weighted speedup. The weighted speedup is computed as $(\sum(IPC_{\text{multi}}/IPC_{\text{single}}))$, where IPC_{multi} is a workload IPC when executing in the multiprogrammed environment and IPC_{single} the one measured from a single application simulation.

B-Fetch results are compared against two light-weight prefetcher designs, the *Stride* prefetcher and the *SMS* prefetcher, configured as described in Section 3.1.3. We found that prefetching the next 8 strided addresses to provide the most speedup for a *Stride* prefetcher. We, therefore use such a configuration in our analysis.

3.2.2 Results and Analysis

We first present results for *B-Fetch* versus the competing light-weight prefetchers on single-threaded and multiprogrammed workloads. We then present a sensitivity analysis to explore *B-Fetch*'s performance in more detail.

3.2.2.1 Single-threaded Workloads

The speedup for single-threaded workloads is presented in Figure 3.9. The results are ordered in alphabetical order. We observe that *Stride* performs poorly, compared to other prefetchers, across all the workloads. Therefore, we focus on comparing *B-Fetch* and *SMS*. The *Geomean* column refers to the geometric mean across the entire set of workloads. *B-Fetch* achieves a geometric average speedup of 23.2% compared to 19.7% for *SMS*. *Geomean pf. sens.* refers to the average performance for the prefetch sensitive workloads (*i.e.* those which showed some benefit from the "Perfect

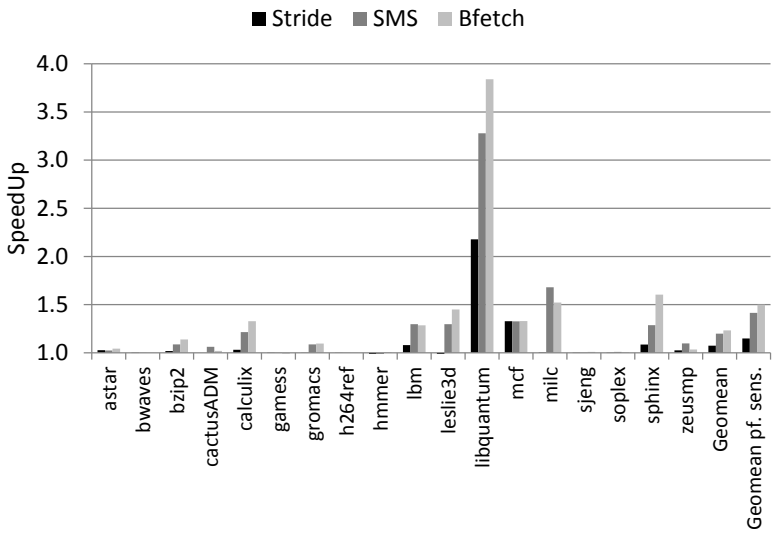


Figure 3.9: Single-threaded workload speedups.

Prefetcher" in Figure 1.2). The results show the *B-Fetch* prefetcher provides a mean speedup of 50.0% across prefetch sensitive benchmarks, compared to 41.5% for *SMS*.

B-Fetch outperforms *SMS* in all but four workloads *cactusADM* , *lbm*, *milc*, and *zeusmp*. Among them, only *milc* shows significant performance difference. *milc* is a corner case in that a single miss reference in *milc* tends to be a very strong predictor for a pattern of references within a large spatial region. As a single pattern history table entry in *SMS* covers 2KB spatial region while a comparable entry in *B-Fetch*'s MHT neg/posPatt field covers only 256 bytes. To verify, we evaluated *milc* with smaller spatial regions. With smaller spatial regions, *SMS*'s the performance drops significantly. When the spatial region is set equivalent to the 256 Bytes neg/posPatt size in *B-Fetch*, the performance gain drops to less than 49.23%, less than *B-Fetch*'s 52.15%. For all other memory intensive benchmarks *B-Fetch* shows same or significantly better performance (*lbm*, *leslie3d*, *libquantum*, *mcf*, and *sphinx*). The larger spatial regions are more likely to span unrelated data structures and require more storage overhead to be implemented. Taking the speedup and the cost of storage

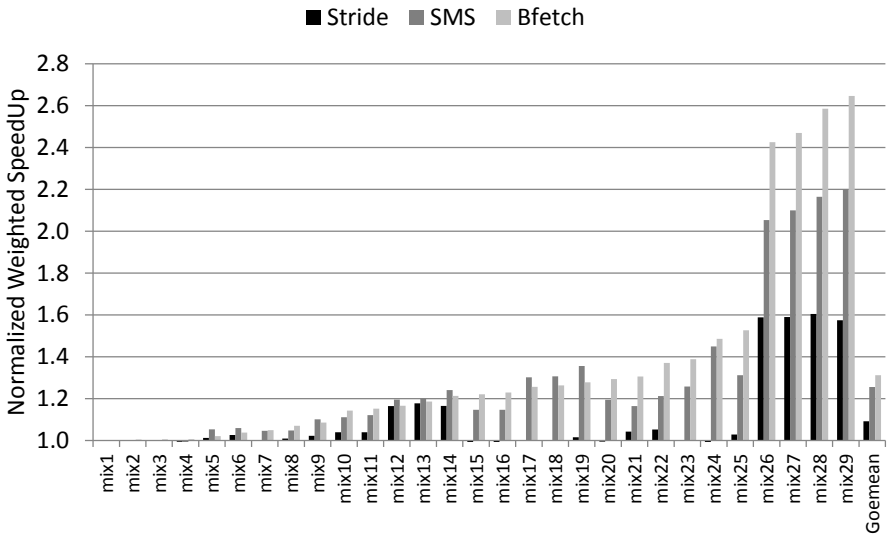


Figure 3.10: Normalized weighted speedup for mixes of 2 workloads.

overhead together, we believe *B-Fetch* presents a better solution for overall data prefetching in single-threaded workloads. We observe that the average lookahead depth is 8 BB with 0.75 branch path confidence.

3.2.2.2 Multiprogrammed Workloads

Figures 3.10 and 3.11 show the performance for mixes of 2 workloads (*mix-2*) and mixes of 4 workloads (*mix-4*) respectively. Note that the performance displayed on Figures 3.10 and 3.11 is ordered by increasing speedup for *B-Fetch*. In both *mix-2* and *mix-4* configurations, *B-Fetch* achieves a greater speedup as compared to *SMS*. For *mix-2*, *B-Fetch* achieves a speedup of 31.2% compared to 25.5% for *SMS*. Similarly, for *mix-4* *B-Fetch* achieves a performance speedup of 28.5% compared to 19.6% for *SMS*. Preliminary results with mixes of 8 workloads continue this trend.

The improved performance under multiprogrammed workloads, versus single-threaded workloads, is a direct consequence of *B-Fetch*'s improved prefetching accuracy. Jerger, *et. al*, emphasize the harmful effects of prefetching in CMP envi-

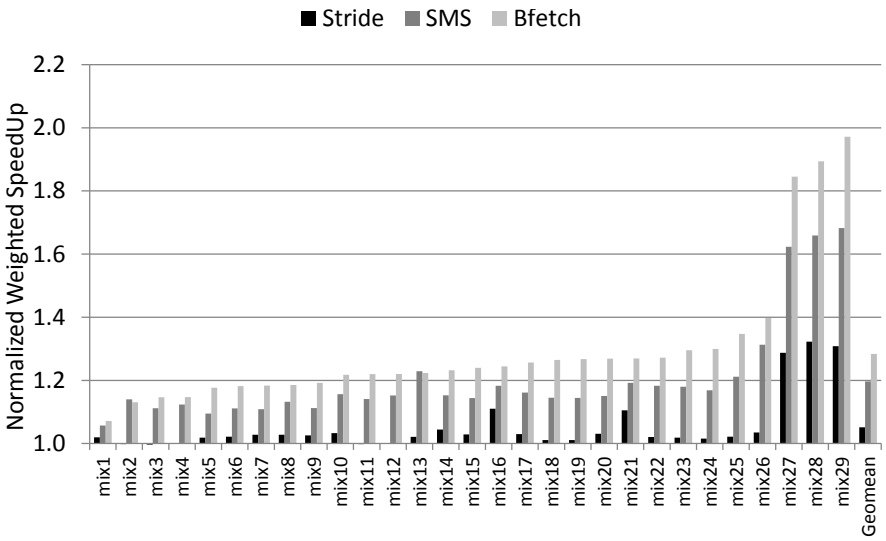


Figure 3.11: Normalized weighted speedup for mixes of 4 workloads.

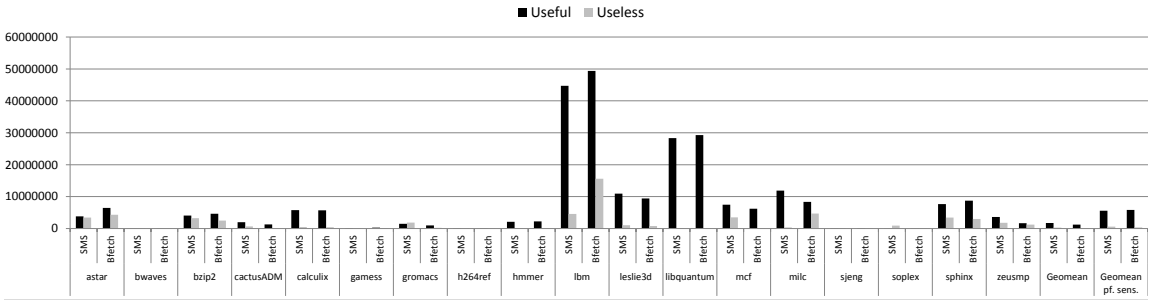


Figure 3.12: Number of useful and useless prefetches issued.

ronments, primarily due to cache and memory bandwidth contention [20]. Since *B-Fetch* provides confidence based control mechanisms (path confidence and per-load confidence), it generates less useless prefetch requests. As a result, it causes less pollution in shared LLC than *SMS*. Figure 3.12 shows the number of useful and useless prefetch issues by *SMS* and *B-Fetch* for all the workloads. We observe that on average *B-Fetch* issues about 4% more useful prefetches while issuing around 50% less useless prefetches compared to *SMS*.

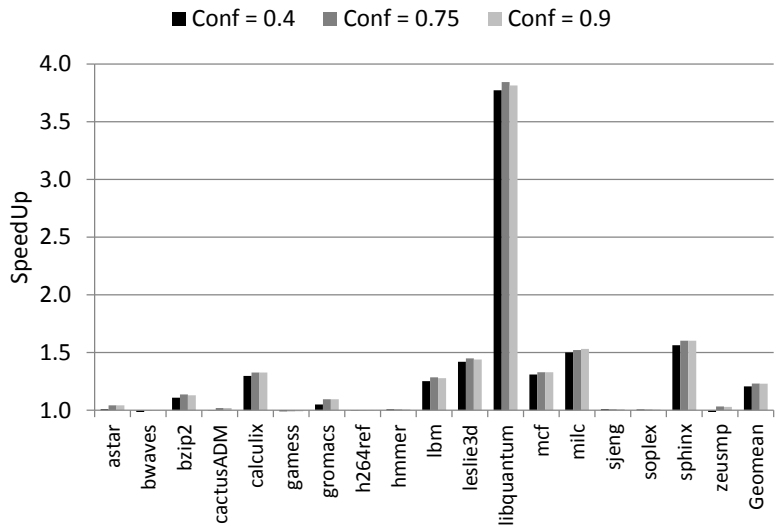


Figure 3.13: Branch confidence sensitivity.

3.2.3 Sensitivity Analysis

This section provides a sensitivity analysis of *B-Fetch*. We explore the impact of different parameters and structures on the performance.

3.2.3.1 Branch Confidence

As discussed in Section 3.1.2, the depth of the lookahead process is controlled by the branch confidence estimation. Figure 3.13 shows the performance speedup as the branch confidence threshold is varied. The average performance speedup is 20.6%, 23.2% and 23.0% for confidence threshold of 0.45, 0.75 and 0.90 respectively. We observe that the best performance is seen at 0.75. With a lower threshold the increased number of low confidence, potentially wrong-path, prefetches issued, leads to higher cache pollution. We note, however, the speedup difference between confidence 0.45 and 0.75 is not large. Hence, the performance is fairly stable across a range of lookahead confidence depths. This is due in part to the per-load filtering mechanism that is capable of filtering out useless prefetch requests. For the threshold values

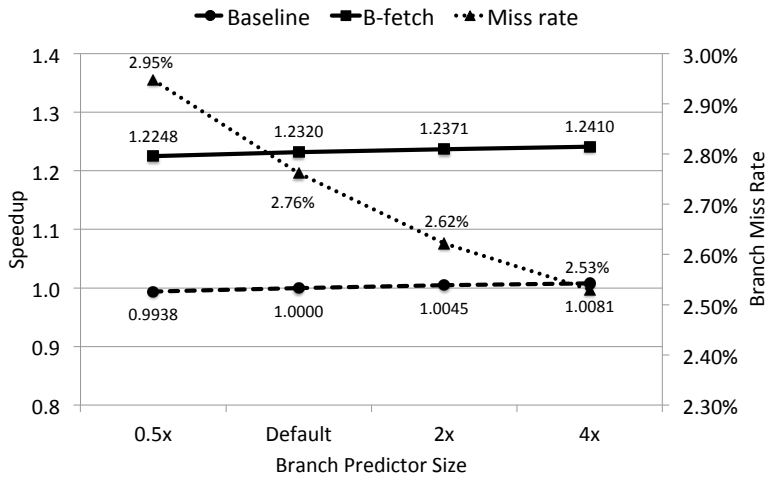


Figure 3.14: Branch predictor size sensitivity.

higher than 0.90, we observe that the lookahead depth decreases. Consequently, the *B-Fetch* engine becomes conservative and prefetches less often.

3.2.3.2 Branch Predictor Size

A branch predictor used in *B-Fetch* is a classic tournament predictor of which final prediction is determined by a simple voting schemes. Thus, we think there might be still room for improvement with more accurate branch predictor. Figure 3.14 shows the performance variation of baseline and *B-Fetch* enabled processors. To emulate a more accurate branch predictor, we simply increase the size of the tournament branch predictor. The figure also plots the branch miss rate as an arithmetic mean across all benchmark sets. Because the default tournament predictor already provides a very low miss rate for these applications, *B-Fetch* does not show a significant additional performance gain from a larger branch predictor. In the future work, we plan to evaluate *B-Fetch* with the state-of-art branch predictors.

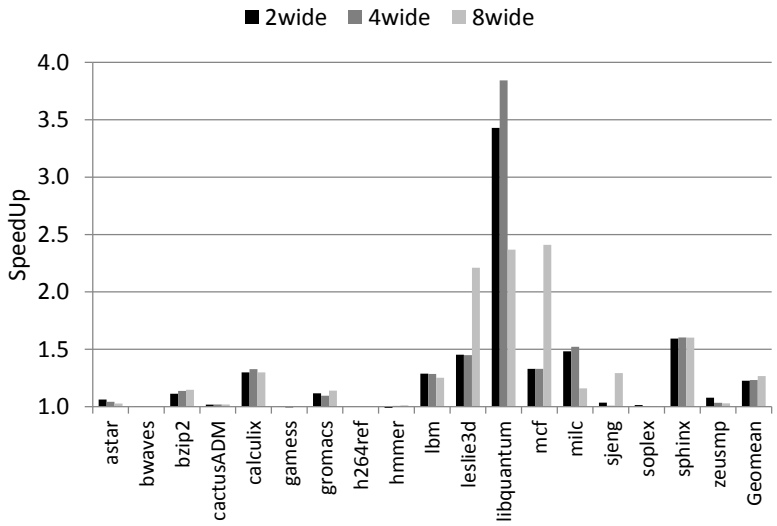


Figure 3.15: CPU pipeline width sensitivity.

3.2.3.3 CPU Pipeline Width

Figure 3.15 shows the performance speed up for a 2-wide, 4-wide, and 8-wide out-of-order pipeline. In general, the performance gradually increases as the pipeline width grows. The most considerable performance improvement with wide machine is observed from *leslie3d* and *mcf*. Conversely, the speedup of *libquantum* and *milc* gets saturated at 4-wide machine. The average speedup found was 22.6%, 23.21% and 26.71% for 2-wide, 4-wide, and 8-wide machines respectively, generally indicating that *B-Fetch* provides reasonable speedups across the spectrum from light-weight to heavy-weight cores.

3.2.3.4 B-Fetch *Size*

Since *B-Fetch* lies within the class of *light-weight* prefetchers, it is important to analyze the effects of its storage size on performance. Figure 3.16 compares the performance improvement for different sizes of *B-Fetch*. The storage overhead is changed to 8.01KB, 9.65KB, 12.94KB and 19.46KB by modifying the number of

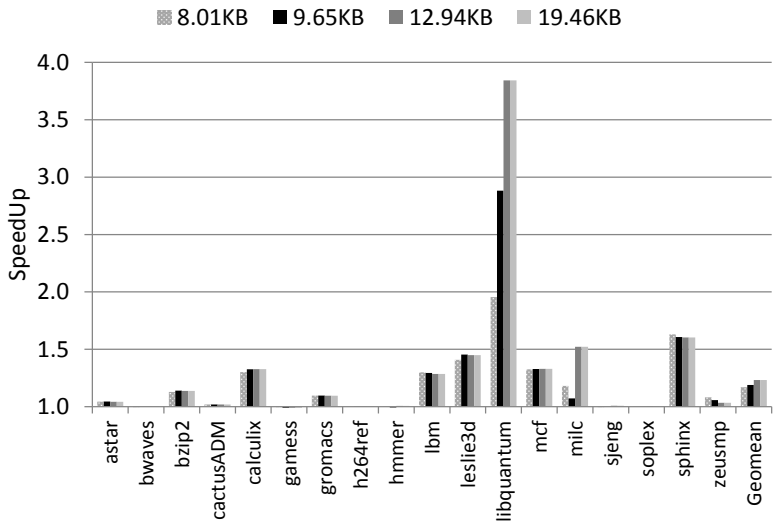


Figure 3.16: *B-Fetch* storage sensitivity.

entries in both the BrTC and the MHT to 64, 128, 256 and 512 respectively. The geometric average speed achieved is 17.0%, 18.9%, 23.21% and 23.1%. We observe the maximum performance speedup is obtained for a size of 12.94KB. Hence the size used for our *B-Fetch* implementation.

3.3 Summary

We proposed a prefetcher that leverages the high accuracy in branch prediction and a novel effective address speculation to identify prefetches candidates. Our proposed technique, *B-Fetch*, outperforms the best in class lightweight prefetcher. *B-Fetch* achieves 23% speedup over baseline with a low hardware impact.

4. PLATFORM POWER MANAGEMENT*

This chapter presents our framework to improve the efficiency in MpSoCs. We first present a CPU oriented approach.*We, later, extend our framework to accommodate for the GPU. *An overview of the framework is discussed. We present details of the implementation and the evaluation of a mobile platform. We show that our framework optimizes the energy consumption while meeting performance constraints.

4.1 A CPU Power Management

4.1.1 Overview

In the proposed framework, the power and performance targets are given by the user applications similar to the approach in [24]. These inputs are converted into power target and QoS bounds by the OS power manager (OSPM) as shown in Figure 4.1. Finally, the proposed *Power Budget* and *QoS controllers* shown in shaded boxes generate the frequency at which the resource under control should run. In what follows, we first introduce the analytical performance and power models, and then present the power budget and QoS controllers.

4.1.2 Power Models

A power model allows our framework to make informed power prediction at all frequencies. The details of the formulation of the CPU power and the platform

*Part of this chapter is reprinted with permission from *Towards Platform Level Power Management in Mobile Systems* by David Kadjo. Umit Ogras, Raid Ayoub, Michael Kishinevsky, Paul V. Gratz, 2014, The 27th IEEE International SOC Conference, IEEE Computer Society Washington DC, Copyright [2014] by IEEE Computer Society

*Part of this chapter is reprinted with permission from *A Control-Theoretic Approach for Energy Efficient CPU-GPU Subsystem in Mobile Platforms* by David Kadjo, Raid Ayoub, Michael Kishinevsky, and Paul V. Gratz, 2015, The 52th ACM/EDAC/IEEE The Design Automation Conference, IEEE Computer Society Washington DC, Copyright [2015] by IEEE Computer Society

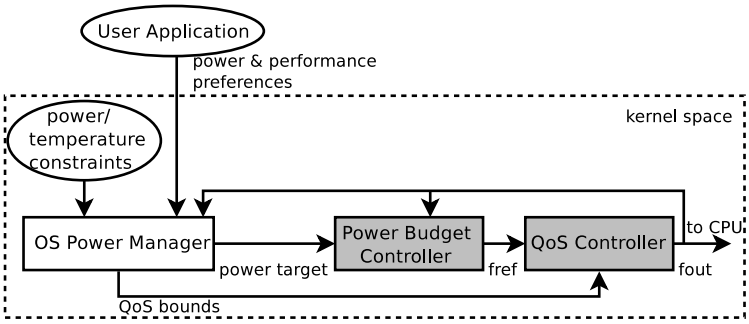


Figure 4.1: Overview of the proposed system (added implementation in shaded blocks).

models is discussed in the subsections below.

4.1.2.1 CPU Power Model

The total power consumption in CMOS circuits consists of a dynamic and a static components. Let us consider a CPU with m power states referred to as $C0, C1, .., Cn, .., Cm$, as defined in the Advanced Configuration and Power Interface (ACPI) specifications [30]. $C0$ is state in which the CPU executes instructions while the rest are different sleep states. We assume that all the internal CPU clocks are stopped beyond power state Cn. We express the CPU dynamic power, $P_{\text{cpu\_dyn}}$, as:

$$P_{\text{cpu\_dyn}} = r_{Ci} C_i V^2 f \quad i = \{0, 1, ..., n\} \tag{4.1}$$

where r_{C_i} is ratio of time spent in power state Ci relative to the sample period duration, C_i is the switching capacitance, V the supply voltage and f the frequency. Since the clocks are gated in power states greater than Cn, those states have no dynamic energy consumption. We express the switching capacitance during the active state ($C0$) as a linear function of the CPU activity expressed as the *ipc*; that is $C_0 = a * ipc + b$. On the other hand, the switching capacitance for power states $C1$ through Cn is constant $C_i = A_i$ since no instructions are executed in these states.

Then, the weighted average dynamic CPU power, $P_{\text{cpu\_dyn}}$, over the sample period is expressed as the sum of the dynamic power in states $C0$ through Cn as:

$$P_{\text{cpu\_dyn}} = r_{C0}(a * ipc + b)V^2 f + \sum_{i=1}^{n} r_{Ci} A_i V^2 f \tag{4.2}$$

Similarly, the static power $P_{\text{cpu\_static}}$ can be expressed as:

$$P_{\text{cpu\_static}} = \sum_{j=0}^{m} r_{Cj} B_j \quad j = \{0, 1, .., m\} \tag{4.3}$$

where r_{Cj} is the ratio of time spent in the j power state relative to the sample period duration and B_j is the leakage power of the corresponding state. Since the frequency is the only controlled parameter, we express the voltage as a function of the frequency as:

$$V = cf + d \tag{4.4}$$

The coefficients c and d are found through a regression analysis using actual voltage and frequency values used by the target platform. The total CPU power then becomes:

$$P_{\text{cpu}} = P_{\text{cpu\_dyn}} + P_{\text{cpu\_static}}$$

4.1.2.2 PMIC, Display and other Components Power Model

The PMIC is a highly integrated solution that provides power to the platform components and the processor. Hence, it is important to have high efficiency PMIC. However, typical PMIC's efficiency is within 60-85% leading to significant power losses [27]. While the efficiency in PMIC is function of the load current, it is relatively flat over the active operating range, as shown in Figure 4.2. Therefore, it can be approximated to a fixed value (80% in our model). Given a PMIC with efficiency ξ,

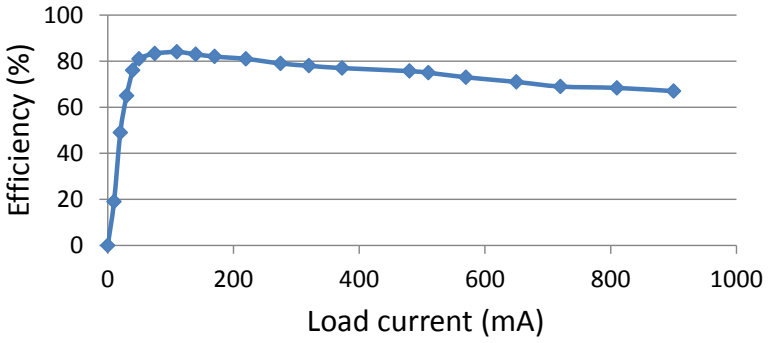


Figure 4.2: PMIC efficiency [27].

we express the power dissipation across the PMIC as a function of the total platform power as:

$$P_{\text{PMIC}} = (1 - \xi)(P_{\text{cpu}} + P_{\text{display}} + \psi) \qquad 0 < \xi < 1 \tag{4.5}$$

where P_{cpu} is the total power dissipated by the CPU, P_{display} is the display power, ψ is defined as the power consumption of the rest of the platform components. The LCD display power can be expressed as a linear function of the display brightness as

$$P_{\text{display}} = m\theta + p_{\text{base}}$$

where m is the display power change per unit of change in the brightness and p_{base} is the baseline display power.

4.1.2.3 Platform Power Model

The total platform power consumption P_{plat} can be expressed as:

$$P_{\text{plat}} = P_{\text{cpu}} + (1 - \xi)(P_{\text{cpu}} + P_{\text{display}} + \psi) + P_{\text{display}} + \psi \tag{4.6}$$

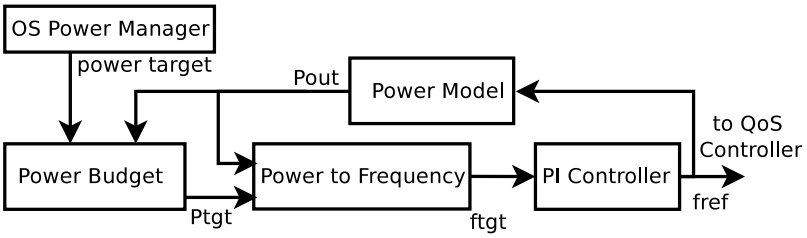


Figure 4.3: Design of the power budget controller.

By substituting Equation 4.4 into Equation 4.2, the platform power can be written in terms of the CPU frequency f as:

$$P_{\text{plat}} = Gf^3 + Hf^2 + Kf + L \tag{4.7}$$

We found the coefficients G, H, K, L using an offline regression analysis using the performance counters listed in Section 4.3.1. To find these coefficients, we swept the CPU frequency and measured the power of individual components (CPU, PMIC, display, etc.) under representative set of workloads.

4.1.3 Power Controller Implementation

The desired power target is computed by the OSPM based on the user inputs and the available temperature headroom as illustrated in Figure 4.3. Then, the computed power target, P_{tgt}, is fed through a *Power to Frequency* module, which computes the largest frequency, f_{tgt}, at which power budget constraint is not violated. That is:

$$Gf_{\text{tgt}}^3 + Hf_{\text{tgt}}^2 + Kf_{\text{tgt}} + L = P_{\text{tgt}} \tag{4.8}$$

Then, we use a proportional integral (PI) controller to find the reference frequency f_{ref} by comparing the power consumption predicted by Equation 4.7 against the power target, as show in Figure 4.3. Note that solving for f_{tgt} using the cubic relation

given in Equation 4.8 would incur a high computational overhead. Therefore, we leverage the fact that the CPU frequency can take a finite set of discrete values in practice. To this end, we implement two controller modes. The first mode finds the CPU frequency that provides the closest power consumption to P_{tgt}. That is, f_{ref} is set such that:

$$|P(f_{\text{ref}}) - P_{\text{tgt}}| = min|P_{\text{tgt}} - P(f_i)|$$

where f_i is the set of frequencies at which the CPU can be set to. Hence, the actual power consumption can be larger than the target under this mode. On the other had, the second controller mode sets the CPU frequency such that the power consumption is *always* maintained below the target power. Hence, f_{ref} is set such that

$$P_{\text{tgt}} - P(f_{\text{ref}}) = min\{P_{\text{tgt}} - P(f_i)\} \;\&\; P_{\text{tgt}} - P(f_i) \geq 0$$

4.1.4 QoS Controller

4.1.4.1 Performance Model

The runtime of an application A can be divided into frequency scalable and non-scalable phases. The duration of the scalable phases (T_{scalable}), observed during CPU intensive periods, is inversely proportional to the CPU frequency. The runtime of the non-scalable phases ($T_{\text{non-scalable}}$), characterized by off-core memory accesses, is oblivious to the CPU frequency. We define the scalability factor S_A of the application A as:

$$S_A = T_{\text{scalable}}/(T_{\text{scalable}} + T_{\text{non-scalable}}) \quad 0 < S_A \leq 1$$

S_A defines how much the application performance scales with the CPU frequency. Let T_1 be the total runtime at frequency f_1, and T_2 the total runtime at frequency

f_2. The scalability factor of application A can be expressed as:

$$S_A = \frac{T_2 - T_1}{T_1(\frac{f_1}{f_2} - 1)} \qquad f_1 > f_2 > 0 \tag{4.9}$$

Computation of Scalability Factor: Many mobile CPUs such as the one in our platform do not provide hardware monitors that directly give the scalability factor. Therefore, we estimate it using the hardware performance counters that measure the application number of page walks (PW) and its last level cache misses (LLC_{misses}), which characterize the CPU off-core activities. We express the scalability as a linear combination of those counters:

$$S_A = c_1 PW + c_2 LLC_{\text{misses}} + c_3 \tag{4.10}$$

In order to find the coefficients c_1, c_2 and c_3, we first ran a set of benchmarks with varying characteristics at different frequencies and empirically computed the scalability factor using Equation 4.9. Subsequently, we applied linear regression to determine these coefficients. The coefficients c_1 and c_2 are negative, as the scalability decreases with increased number of page walks and cache misses. They represent the change in the scalability per change of the page walks and cache misses respectively. Finally, the constant term c_3 represents the relative weight of the scalability for purely CPU intensive workloads.

Setting the Performance QoS Target: The target performance, I_{ref}, is specified relative to I_{max} the maximum number of instructions that can be executed per control interval.

$$I_{\text{ref}} = \beta I_{\text{max}} \qquad 0 \leq \beta \leq 1 \tag{4.11}$$

Note that a fixed performance target β cannot deliver optimal energy efficiency

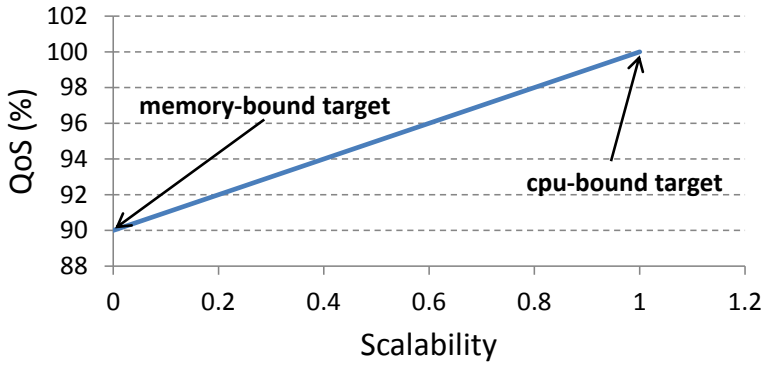


Figure 4.4: QoS as a function of the scalability factor.

across different types of applications. For instance, a CPU-bound application with a scalability factor close to 1, should have a QoS performance target close to 100% since any reduction in frequency results in a sizable increase in runtime. Hence, any power savings is likely to be compromised by a longer execution time. On the other hand, the CPU frequency could be safely reduced for a memory bound application without a significant increase in runtime. Consequently, we expressed the performance target as a function of the scalability factor:

$$\beta = qS_A + k \qquad q > 0 \quad \& \quad k > 0 \tag{4.12}$$

The constant k represents the minimum performance target which will be used for workloads entirely dominated by off core memory accesses. It is set to 90% as shown in Figure 4.4. On the other hand, q determines the increase in the performance target as S_A increases from 0 to 1, as illustrated in Figure 4.4.

4.1.4.2 Instruction Slack Controller

The number of instructions (I) executed by the CPU in a sample period may differ from the target performance for two reasons. First, the CPU can only be

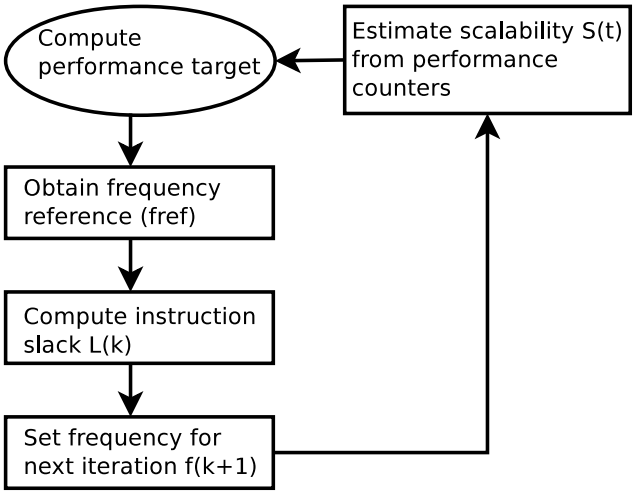


Figure 4.5: Design of the QoS controller.

set to specific frequencies which in some cases might differ from the controller requested frequency. Second, it is not possible to precisely predict the scalability factor which changes dynamically. Therefore, the actual workload execution can be faster or slower than the performance target. Suppose that performance target can be achieved by a reference frequency f_{ref}. We define the instruction slack as the discrepancy between the number of instructions executed by the CPU and the number of instructions that would have been executed at the reference frequency. The instruction slack can be expressed as follows:

$$L(t) = L(t-1) + \frac{S_A(t) I f(t)}{f_{\text{ref}}(t)} - S_A(t) I \tag{4.13}$$

$L(t)$ is the cumulative instruction slack until time t, $L(t-1)$ is the previous cumulative instruction slack, $S_A(t)$ is the running application scalability factor a time t, $f(t)$ is the CPU frequency at time t, I is the number of instruction executed and $f_{\text{ref}}(t)$ is the reference frequency. A positive slack means the workload has been executing faster than the performance target. A negative slack means the workload

has been executing slower to maintain the target performance. We implement a closed loop controller to regulate the CPU frequency and the instruction slack. The output frequency from the controller depends on the amount of slack accumulated. The purpose of the controller is to drive the slack to zero and achieve the desired performance target. The output frequency that will stabilize the system given in Equation 4.13 by placing its pole within the unit circle can be found as

$$f(t+1) = -G(t)L(t) + f_{\text{ref}}(t) \tag{4.14}$$

where the feedback gain $G(t)$ as a function of desired pole λ is

$$G(t) = \frac{(1-\lambda)f_{\text{ref}}(t)}{S_A(t)I} \qquad 0 \leq \lambda \leq 1 \tag{4.15}$$

4.1.4.3 QoS Controller Implementation

Figure 4.5 shows the block diagram describing the operation of the *QoS controller*. First, the reference frequency required to reach the target performance is computed. Then, the scalability factor and the instruction slack are computed periodically every control interval, as described in sections 4.1.4.1 and 4.1.4.2. Next, the reference frequency and instruction from the previous control interval is used to update the slack and compute next frequency, as explained in Section 4.1.4.2. Finally, the CPU frequency for the next epoch is computed using Equation 4.14. Since the CPU allows to be set to a few specific frequencies, we set the CPU to the highest available frequency below the controller output frequency when the slack is positive. For a negative slack, we set the CPU at the lowest frequency above the output frequency.

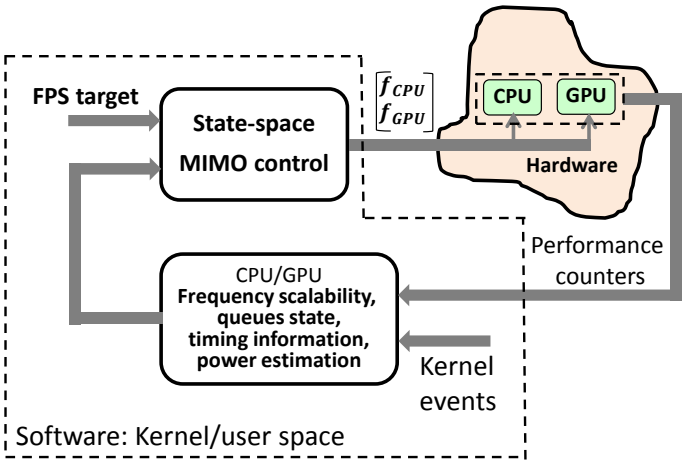


Figure 4.6: Closed system control overview.

4.2 CPU-GPU Power Management

4.2.1 Overview

In this section, we discuss the details of our approach. The goal of our work is to minimize the energy of the CPU and GPU subsystems while delivering a desired Frame Per Second (FPS) for a set of different graphic applications. FPS is a standard metric in the domain of graphic processing. Conceptually, the amount of compute resources (GPU and CPU) required to achieve a given FPS on these applications varies widely over time and from application to application. Thus, it is possible to achieve energy savings by using power management techniques to reduce the compute resources to just sufficient to achieve the desired FPS. To achieve this behavior, we propose a dynamic control system that intelligently adjusts the computing resources to optimize for energy. We modeled this problem as a control theoretic Multi-Input-Multi-Output (MIMO) state-space system to ensure control robustness. The control variables in this problem are the dynamic frequency and voltages of the CPU and the GPU components. Figure 4.6 shows the high-level overview of this design. Since

voltage frequency relation for a given design is known apriory, it is sufficient to control the frequency and let the firmware select the lowest possible voltage for CPU and GPU corresponding to the selected frequency.

Our proposed control subsystem is implemented at the software level, partitioned between the Android Linux kernel and user space. It takes two primary inputs, the desired FPS target and the feedback from the system. The default target FPS is available within the kernel but it can also be provided by the applications. The feedback input includes performance and power information collected from the system. This input is captured through OS kernel instrumentation (e.g. frame execution time) and via light-weight run-time models that use hardware performance counters as input (e.g. CPU/GPU power models). More details on these inputs are given in the subsequent subsections. The control system periodically samples the parameters of the system and computes the controlled values. It is assumed that the controller sets the frequency once at the beginning of the control interval and keeps it unchanged during the control interval.

4.2.2 Frequency Scalability

Changing the frequency of the device, e.g. CPU or GPU, affects the execution time of an application or a portion of the application between two sampling points e.g., a time to compute one or multiple frames of the graphic application. We will call the execution time of interest, *active time*, and denoted as T_a. The time when the device enters one of the sleep states and hence is not executing instructions is called a *sleep time*, which is denoted as T_s. The active time on a device can be divided into frequency scalable and non-scalable portions: $T_a(f) = T_{\text{scalable}}(f) + T_{\text{non-scalable}}$, where f is the operating frequency. The duration of the scalable phases (T_{scalable}) is inversely proportional to the operating frequency, i.e. $T_{\text{scalable}}(f_2) =$

$\frac{f_1}{f_2} \cdot T_{\text{scalable}}(f_1)$. The duration of the non-scalable phases ($T_{\text{non-scalable}}$), characterized by resource constraints such as off-core memory accesses, is oblivious to the CPU or GPU frequency. While off-core memory accesses may depend upon the frequency of the device execution due to different dynamics of cache and memory accesses at different frequencies, these second order dependencies are small and can be tolerated by the closed loop control. We now define the frequency scalability factor $S(f)$ of the application as:

$$S(f) = T_{\text{scalable}}(f)/(T_{\text{scalable}}(f) + T_{\text{non-scalable}}) \tag{4.16}$$

$S(f)$ defines which portion of the application performance scales with the device frequency. Its values are bounded between 0 (no portion of the execution is affected by the frequency) and 1 (the execution is fully scalable). We leverage the frequency scalability factor to predict the value of the active time at a next frequency. Assume the frequency changes from f_1 to f_2. Using the above equations, one can express the active time at f_2 using the scalability factor as follows:

$$T_a(f_2) = T_a(f_1) + S(f_1) \cdot T_a(f_1) \cdot (\frac{f_1}{f_2} - 1) \tag{4.17}$$

As an application goes through different phases of the execution, its scalability factor changes. Therefore, the runtime control should estimate S during each control period. We denote a frequency scalability factor prediction computed for the k-th control interval as S_f^k. The runtime estimator is computed as a function of selected device performance counters. The specific function is learned off-line via machine learning techniques. We ran a set of training applications with different characteristics and then applied a linear regression analysis to select the performance counters

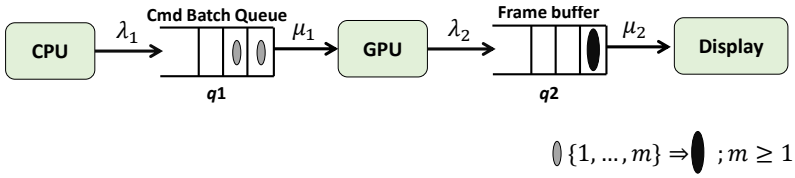


Figure 4.7: CPU-GPU-Display queuing system.

of interest along with the coefficients.

4.2.3 CPU-GPU-Display Queuing Model

The interaction between the CPU, GPU, and Display is illustrated in Figure 4.7. For a graphic application, the CPU generates batches of commands. A pointer to each batch is stored into a slot of the command batch queue, q_1. The batches are read and processed by the GPU which generates a display frame. The pointer to the frame is stored into the frame buffer, another queue denoted as q_2. In some cases, multiple batches may contribute to a single display frame. On the Figure, the rate of injection into q_1 and q_2 are denoted as λ_1 and λ_2, respectively. The rate of ejection from the two queues are denoted μ_1 and μ_2. These rates are computed over the sampling interval T^k (the time interval between two consecutive sampling points t^k and t^{k+1} is denoted as $T^k = [t^k, t^{k+1})$, where $k \geq 1$). We introduce a job ratio parameter r^k, which is the average number of batches per frame within the sampling interval T^k. The relation between μ_1 and λ_2 can then be expressed as:

$$\mu_1^k = r^k \lambda_2^k \tag{4.18}$$

Let us denote the ratio of injection rates into queue q_i during two consecutive sampling intervals λ_i^k and λ_i^{k-1} as Θ_i^k, $k \geq 2$. Assuming continuous flow of jobs without sleep periods, the injection rate is reciprocal to the active execution time

(during the frame period): $\lambda^k = 1/T_a^k$. Using equation 4.17 the injection ratios Θ_i^k for the CPU and GPU can be expressed via the scalability coefficient, S_f, as follows:

$$\Theta_i^k = \lambda_i^k / \lambda_i^{k-1} = \frac{1}{1 - S_{f,i}^{k-1}(1 - \frac{f_i^{k-1}}{f_i^k})} \tag{4.19}$$

Using equation 4.18, the occupancy of the queue, q^{k+1}, at the sampling point t^{k+1} can be described using queue flow equations as follows:

$$\begin{aligned} q_1^{k+1} &= q_1^k + T\lambda_1^k - Tr^k\lambda_2^k \\ q_2^{k+1} &= q_2^k + T\lambda_2^k - T\mu_2^k \end{aligned} \tag{4.20}$$

The values of the injection rates, λ_1^k and λ_2^k, during the next control period, are not known at sample point t^k. Using the injection ratio definition (Equation 4.19), however, the above queue equations can be rewritten in terms of the injection rates *during the previous period*. Assuming that the display ejection rate is fixed we can also use the previous period ejection rate $\mu_2^k = \mu_2^{k-1}$:

$$\begin{aligned} q_1^{k+1} &= q_1^k + T\lambda_1^{k-1}\Theta_1^k - Tr^{k-1}\lambda_2^{k-1}\Theta_2^k \\ q_2^{k+1} &= q_2^k + T\lambda_2^{k-1}\Theta_2^k - T\mu_2^{k-1} \end{aligned} \tag{4.21}$$

Rewriting in the matrix form, one gets the following recurrent state equation describing behavior of the CPU-GPU-Display system:

$$\boldsymbol{Q}^{k+1} = \boldsymbol{Q}^k + \boldsymbol{\Gamma}^k\boldsymbol{\Theta}^k + \boldsymbol{C}^k \tag{4.22}$$

where $\boldsymbol{Q}^{k+1}$ and $\boldsymbol{Q}^k$ are vectors of the queues state (queues occupancies) at sampling points t^{k+1} and t^k, input vector $\boldsymbol{\Theta}^k$ of injection ratios Θ_1^k and Θ_2^k for the two

queues q_1 and q_2, that depend on the control decision on the next frequencies f_i^k for the CPU and the GPU correspondingly, as shown by equation 4.19. The matrix $\mathbf{\Gamma}^k$ and the vector $\boldsymbol{C}^k$ are defined as:

$$\mathbf{\Gamma}^k = T \begin{bmatrix} \lambda_1^{k-1} & -r^{k-1}\lambda_2^{k-1} \\ 0 & \lambda_2^{k-1} \end{bmatrix}$$

$$\boldsymbol{C}^k = T \begin{bmatrix} 0 \\ -\mu_2^{k-1} \end{bmatrix}$$

4.2.4 State-space Controller Regulating QoS

We, next, designed a feedback controller that dynamically manages the controllable frequencies of the components, CPU and GPU, to achieve the desired performance (QoS) constraints. The proposed controller stabilizes the state of the queues around a target utilization as measured by the occupancy of the queues. Equation (4.22) represents the linearized dynamics of the CPU-GPU-Display system around an operating point. Maintaining sufficiently large utilization of queue q_1 guarantees that the GPU never stalls due to a lack of batches dispatched by the CPU even in presence of fluctuations in injection and ejection rates. Maintaining sufficiently large utilization of q_2 guarantees that the display has always a new frame to draw at the required display rate and hence the visual quality of service is not suffering.

Let $\boldsymbol{Q}_{ref}^k$ be the vector of reference utilizations for the system queues. We apply the control feedback for a system with input reference [28], to the state equation in Equation 4.22, to compute the input vector $\mathbf{\Theta}^k$ as follows:

$$\mathbf{\Theta}^k = -\boldsymbol{G}^k(\boldsymbol{Q}^k - \boldsymbol{Q}_{ref}^k) - (\mathbf{\Gamma}^k)^{-1}\boldsymbol{C}^k \tag{4.23}$$

where $\boldsymbol{G^k}$ is the state feedback gain matrix. The last term in the above equation is used to compensate for the term $\boldsymbol{C}^k$ in the state equation, which is not controllable by the input $\boldsymbol{\Theta}$ of the system. The selection of the gain $\boldsymbol{G^k}$ should ensure that the queues utilization is asymptotically converging to the reference values.

Substituting equation (4.23) into the state equation (4.22) we can express the state equation of the closed-loop system as follows:

$$\boldsymbol{Q}^{k+1} = (\boldsymbol{I} - \boldsymbol{\Gamma}^k \boldsymbol{G})\boldsymbol{Q}^k + \boldsymbol{\Gamma}^k \boldsymbol{G} \boldsymbol{Q}^k_{ref} \tag{4.24}$$

The state feedback matrix $\boldsymbol{G}^k$ must be selected such that the closed-loop system described by Equation 4.24 is stable. This can be achieved by placing the eigenvalues of the matrix $(\boldsymbol{I} - \boldsymbol{\Gamma}^k \boldsymbol{G}^k)$ within the unit circle.

$$(\boldsymbol{I} - \boldsymbol{\Gamma}^k \boldsymbol{G}^k) = \begin{bmatrix} \rho_1 & 0 \\ 0 & \rho_2 \end{bmatrix}$$

where ρ_1 and ρ_2 are the Eigen values corresponding to the poles that should be within the unit circle [28]. The feedback gain matrix can, then, be calculated as:

$$\boldsymbol{G}^k = -(\boldsymbol{\Gamma}^k)^{-1} \begin{bmatrix} \rho_1 - 1 & 0 \\ 0 & \rho_2 - 1 \end{bmatrix} \tag{4.25}$$

Finally, the intermediate output of the controller, the injection ratio, can be written as:

$$\boldsymbol{\Theta}^k = \frac{1}{T} \begin{bmatrix} \frac{\rho_1 - 1}{\lambda_1^{k-1}} & \frac{r_2^{k-1}(\rho_2 - 1)}{\lambda_1^{k-1}} \\ 0 & \frac{\rho_2 - 1}{\lambda_2^{k-1}} \end{bmatrix} (\boldsymbol{Q}^k - \boldsymbol{Q}^k_{ref}) + \begin{bmatrix} \frac{r_2^{k-1}\mu_2}{\lambda_1^{k-1}} \\ \frac{\mu_2}{\lambda_2^{k-1}} \end{bmatrix} \tag{4.26}$$

4.2.5 Energy Optimal Frequencies

We expressed the power consumption of each device i (CPU, GPU) as a function of the frequency that is the only controllable parameter as $P_i = a_3^i f_i^3 + a_2^i f_i^2 + a_1^i f_i + a_0^i$ [34]. Using this formula and Equation 4.17, the energy dissipation is computed as follows:

$$E^i(f_i) = (\sum\nolimits_i P_a^i(f_i) T_a^i(f_i) + \sum\nolimits_{i,j} P_j^i T_j^i)(1 + \xi^i) \tag{4.27}$$

where $f_i \geq f_{min}$ (f_{min} is the frequency at which the predicted active time of the frame, $T_a^i(f_{min})$, is less than a required time constraint), $P_a^i(f_i)$ is the power of device i in active state, $T_a^i(f_i)$ is the duration of the active execution, for example time to process one frame of the graphic workload, for a device i. P_j^i is power during sleep in the C_j^i state and T_j^i its sleep time duration. Each of the above power components adds a penalty ξ corresponding to the inefficiency of the power regulator .

Finding an analytical solution for the frequency, f_e^i, that minimizes energy in Equation 4.27 is not trivial during runtime since it would require solving a 4th order polynomial under the constraint of $f_i \geq f_{min}$. As each device has a small set of frequency values, finding f_e^i at runtime is done via sweeping the possible frequency values and finding the minimal value of Equation 4.27.

4.2.6 Adding Energy Optimization

The algorithm presented in previous subsection selects the energy optimal frequency that satisfies the performance constraints as seen by the performance predictor that uses runtime frequency scalability estimates. Unlike the performance controller output described by the Equations 4.23 and 4.30, this algorithm does not take into account any performance jitter (burstiness) since it does not use the reference constraint on the number of packets in the system queues. In addition,

it considers individual devices, such as a GPU and a CPU, in isolation and does not guarantee selection of the frequency for the producer (CPU) that would enable sufficient performance to the consumer (GPU).

When the CPU or the GPU enters sleep states, we can then leverage the sleep time as a slack to optimize for energy. We define β_i as a metric to measure how far the active time of recent frames is from the energy efficient target of device i. $\beta_i^{k-1} = T_a(f_{e_i}^{k-1})/T_a(f_i^{k-1})$. The target value for β_i^{k-1} is 1. When $\beta_i^{k-1} > 1$ then we need to reduce f_i^k in the subsequent control interval and vice versa. We define the matrix $\boldsymbol{E}^k$ as a diagonal matrix in which the (i,i) element is equal to β_i^{k-1}. We use the matrix $\boldsymbol{E}^k$ as a linear transformation on the injection rate vector $\boldsymbol{\Theta}^k$ and apply it as part of the state equation of the queuing system to enable energy efficient decisions when possible:

$$\boldsymbol{Q}^{k+1} = \boldsymbol{Q}^k + \boldsymbol{\Gamma}^k \boldsymbol{E}^k \boldsymbol{\Theta}^k + \boldsymbol{C}_e^k \tag{4.28}$$

where $\boldsymbol{C}_e^k = \boldsymbol{\Gamma}^k \boldsymbol{E}^k (\boldsymbol{\Gamma}^k)^{-1} \boldsymbol{C}^k$, applies this linear algebraic transformation to the vector $\boldsymbol{C}^k$ to ensure convergence to the target FPS during steady-state. We can now derive the energy efficient, $\boldsymbol{\Theta}^k$, vector using Equation 4.28 and the control related derivation steps from Section 4.2.4.

$$\boldsymbol{\Theta}^k = \frac{1}{T} \begin{bmatrix} \frac{\rho_1 - 1}{\beta_1^{k-1} \lambda_1^{k-1}} & \frac{r_2^{k-1}(\rho_2 - 1)}{\beta_1^{k-1} \lambda_1^{k-1}} \\ 0 & \frac{\rho_2 - 1}{\beta_2^{k-1} \lambda_2^{k-1}} \end{bmatrix} (\boldsymbol{Q}^k - \boldsymbol{Q}_{ref}^k) + \begin{bmatrix} \frac{r_2^{k-1} \mu_2}{\lambda_1^{k-1}} \\ \frac{\mu_2}{\lambda_2^{k-1}} \end{bmatrix} \tag{4.29}$$

Given $\boldsymbol{\Theta}^k$, we compute the final output of the controller, the next frequencies for the CPU and the GPU:

$$f_i^k = \frac{\Theta_i^k S_{f,i}^k f_i^{k-1}}{1 - \Theta_i^k (1 - S_{f,i}^k)} \tag{4.30}$$

where index i is equal to 1 for the CPU, and 2 for the GPU.

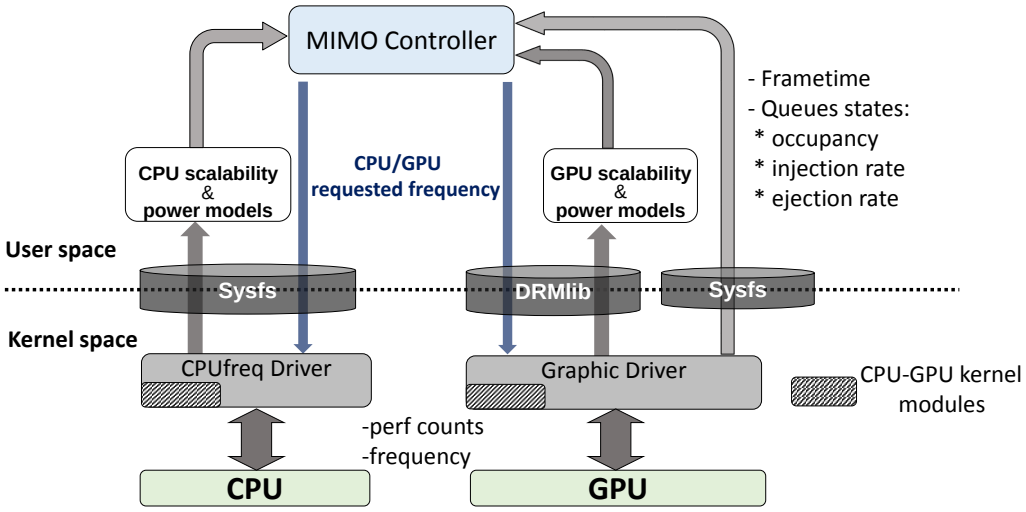


Figure 4.8: Implementation overview.

4.2.7 Implementation

Our proposed implementation is partitioned into user and kernel space, as shown in Figure 4.8. At the control sampling rate, the states of the CPU and GPU are fed into the controller which sets the target frequencies. The inputs to the controller are the CPU and GPU runtime frequency scalability and the power estimated through the user-space analytic models. The models take as input the information from the hardware counters and output the power and performance estimation to the controller. Additional inputs are the frame processing time (frametime) and the queues states (the injection and ejection rate, the occupancy) which are relayed by the kernel layer. The requested frequencies are communicated to the kernel layer.

The kernel space implementation is partitioned into CPU and GPU modules. The CPU module is implemented as a frequency governor driver. This module periodically, at the rate of the control period, samples the CPU hardware counters and the cores frequencies then reports them to the models in user space. The requested CPU

frequency from the controller is communicated to the CPU module. The communication between the CPU module and the user space layer is done through the *sysfs*, the filesystem interface for exporting kernel objects. The GPU module is written within the graphic driver. This module computes and collects the queue properties and reports them to the controller. The GPU module also communicates to the user space through the *sysfs* interface. The GPU hardware performance counters are queried using the DRM library [26]. Likewise, the requested GPU frequency by the controller is communicated to the GPU using the DRM library.

4.3 CPU Optimization

4.3.1 Methodology

Since the analytical power models are based on data collected on an experimental platform, we begin with describing the data measurement setup and methodology.

4.3.1.1 Host Platform

We performed data collection, model validation and experimental evaluation using a mobile platform based in the Intel© Atom™ Clovertrail processor and running Android Jelly Bean based on the Linux kernel version 3.0.34. The Atom core in the target processor has four power states referred to as $C0$, $C2$, $C4$ and $C6$. We implemented the proposed technique as modules within the Linux kernel and compared them against the default frequency governor.

4.3.1.2 Power Measurement and Validation

The mobile device under test is hooked up to a data acquisition system, NI-DAQ-6070E, that allows fine granular voltage and current readings across the entire platform. Using this setup, we measure the power consumption at a rate of 3000 samples per second. As a result, we obtain a very accurate power consumption

decomposition as a function of time. We use the execution time as the primary performance metric.

4.3.1.3 Runtime Monitoring

The proposed controllers rely on analytical power and performance models, which rely on hardware performance counters, since run-time data acquisition is not feasible for end users. We use the low level assembly functions available within the kernel to read these counters. In particular, we monitor the number of instructions retired (`Instruction Retired`), the numbers of clock cycles (`UnHalted Core Cycles`), the page walks (`PAGE_WALKS.WALKS`) and the last level cache misses (`L2_LINES_IN`). The power states residencies are determined by reading the corresponding residency counters (`MSR_PKG_C2_RESID.`, `MSR_PKG_C4_RESID.`, `MSR_PKG_C6_RESID.`). Power and performance predictions are computed using these monitors in a control interval of 50ms. The overhead of this process is measured as 0.1%.

4.3.2 Results and Analysis

In this section, we compare the proposed techniques against the default ondemand frequency governor in the Linux kernel. The on-demand policy sets the CPU frequency to the highest available frequency when the CPU utilization rises above a threshold. When the utilization falls below the threshold, the policy decreases the CPU frequency in steps [52].

We used two classes of workloads. The first class includes *cpu-bound* and *memory-bound* C/C++ benchmarks which are compiled using the Android standalone compiler provided by the Android Native Development Kit (NDK) and run from the adb command shell. The second class consists of Android workloads *Caffeine* and *Quadrant*, which are widely used for mobile device characterization. We implement three QoS performance target functions (equation 4.12) with different minimum per-

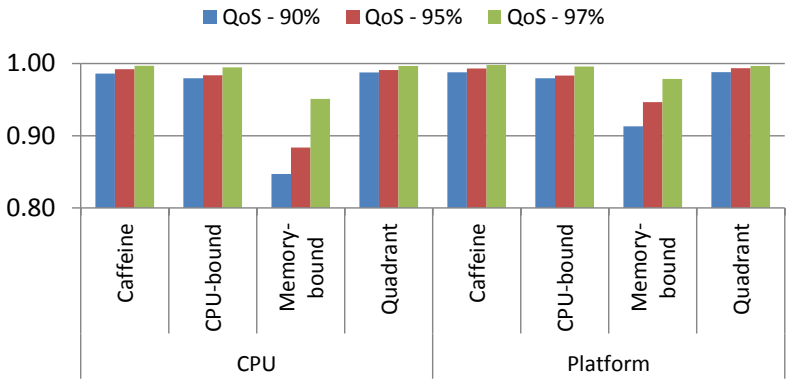


Figure 4.9: Energy consumption normalized to the ondemand frequency governor.

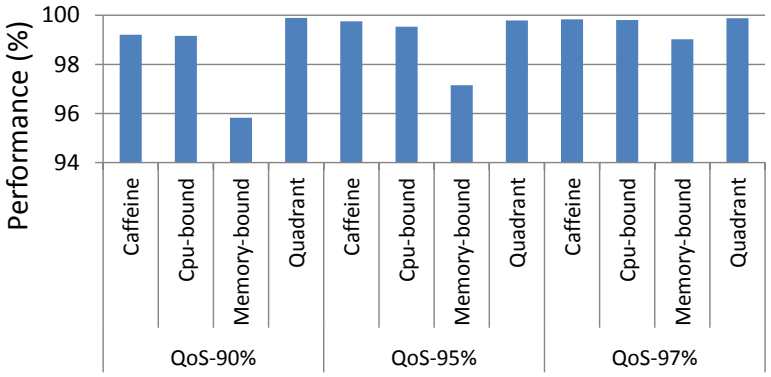
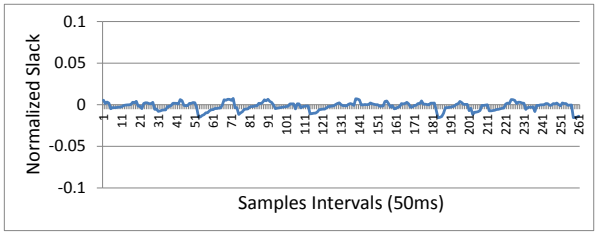


Figure 4.10: Application performance relative to max performance.

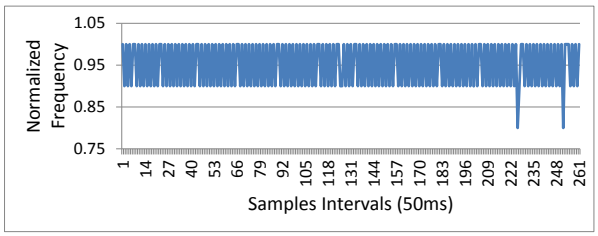
formance targets (90%, 95% and 97%). These functions are referred to on the figures as *QoS-90%*, *QoS-95%* and *QoS-97%*.

4.3.3 Performance

The goal of the performance controller is to meet the performance requirement while minimizing the energy consumption. This is achieved by decreasing CPU voltage and frequency to save power whenever that results in no or negligible loss in performance. The normalized energy consumption obtained using the proposed controller under each QoS setting is shown in Figure 4.9. CPU energy consumption



(a) Instructions slack



(b) Controller output frequency

Figure 4.11: QoS controller behavior for the *cpu-bound* workload.

under memory bound application is decreased by more than 15% under 90% QoS setting, since the proposed controller recognizes that this workload is dominated by off-core activities. Thus, the CPU frequency is reduced to save energy at negligible performance degradation. The corresponding reduction in platform energy is 8%. Considering that the platform power including all resources can reach ∼5W, 8% energy savings at platform level is significant. For instance, 10$hour$ of baseline battery operation could be extended by almost 50min by simply using this approach.

We also observe that CPU and platform energy savings for *QoS-95%* and *QoS-97%* settings are lower than the saving observed at *QoS-90%*. This is expected since the controller is less aggressive in reducing the power as the QoS requirement increases. The proposed approach is also more energy efficient than the on-demand governor for other workloads. In particular, the Cpu-bound workload shows an en-

ergy saving of ∼2% at the platform level for *QoS-90%*. The same scenario is observed for *Caffeine* and *Quadrant*. This is because these workloads have a scalability factor close to 1. Therefore, any attempt to reduce the voltage and frequency increases the runtime and undermines any potential savings in platform energy consumption despite the reduction in power consumption. Overall, the proposed policy consistently outperforms the default on-demand policy across all the workloads.

Figure 4.17 shows that the proposed QoS controller achieves the desired objectives. More precisely, there is no performance degradation for the workloads (*Caffeine*, *cpu-bound* and *Quadrant*) with scalability close to 1. For all cases, the achieved performance is greater than the minimum QoS target. Detailed operation of the controller when executing the *cpu-bound* workload at QoS of 95% is illustrated in Figure 4.11. The controller tries to drive the instructions slack to zero and maintain the target performance, as expected (Figure 4.11a). Since the CPU allows only a discrete set of specific frequencies, the exact output frequency requested by the controller cannot be always selected. Therefore, the controller oscillates between closest available CPU frequencies as shown in Figure 4.11b such that the average coincides with the ideal frequency.

4.3.4 Power Budgeting

In this experiment, we set the power target to 50%, 70% and 90% of the maximum power consumption. Figure 4.12 shows the result for the controller mode that maintains the CPU power around the target power. On average, the CPU power is maintained at 89.75%, 72.03% and 51.48% when the knob is set to 90%, 70% and 50% respectively. Overall, the CPU power consumption is maintained within less than 1% of the specified power target. Figure 4.13 shows the results when the CPU power is maintained below a target power. It is observed that the power is success-

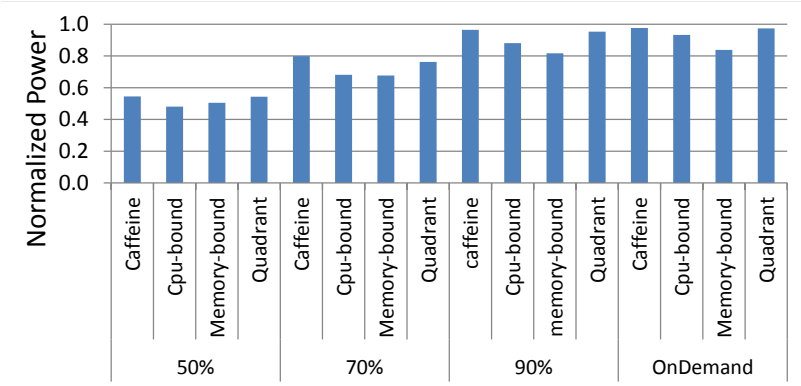


Figure 4.12: Normalized CPU power for the mode that maintains the power at around the target power.

fully maintained below the target across all the workloads. On average, the power is maintained at 84.52%, 67.25% and 44.16% of the maximal power when the knob is set to 90%, 70% and 50% respectively. These results show that the power model introduced in Section 4.1.3 is an accurate representation of the CPU power. Therefore, it can be used as a soft power budget constraint. We note that the on-demand policy has no power limiting feature. The CPU frequency depends only on its utilization. Therefore, the proposed power budget controller adds a new functionality which is not possible with the on-demand governor and keeps the power consumption within the target power budget.

4.4 CPU-GPU Optimization

4.4.1 Methodology

We implemented our technique on a platform based on the Intel Baytrail SoC running Android Jelly Bean. The results are compared against the default Android power management policy. The default policy implements the common interactive frequency governor for selecting the CPU frequency, and the default GPU power management policy for controlling the GPU frequency. The platform under test was

CPU	4-wide O3 processor 192-entry ROB
L1I & L1D cache	64KB 8-way 2-cycle latency
L2 cache	Unified 256KB 8-way 10-cycle latency
Shared L3 cache	2MB/Core 16-way 20-cycle latency
Off-chip DRAM	200-cycle latency
Branch predictor	6.55KB Tournament predictor 2.76% branch prediction miss rate
Branch Path Confidence Threshold	0.75
Per-Load Filter Threshold	3

Table 4.1: Baseline configuration.

connected to a data acquisition system, the NI-USB6289, which we used for reading the CPU and GPU power data. The DAQ was configured to measure the CPU and GPU power and energy at a rate of 3000 samples per second. We use FPS as a QoS metric in our results.

To evaluate our approach we selected a set of representative applications which can be partitioned into two groups. The first group consists of graphic intensive or performance test benchmarks (*GLBench*, *3DMark*, and *Citadel*). The second set targets common graphic applications (*Angry Birds*, *Fruit Ninja*, *Candy Crush*, *Drag Racing*, and *Hill Climb*). All the applications used in our experiments are available in the Android Market. The sampling period of our control is empirically set to 50ms as a good trade-off between accuracy and run-time overhead. The components of the occupancy reference vector $\boldsymbol{Q}_{ref}$ are the frame buffer queue reference occupancy (Q_{ref_gpu}) and the command batch queue occupancy (Q_{ref_cpu}). We set Q_{ref_gpu} to 1 and Q_{ref_cpu} to the number of batches corresponding to a frame. These references were set empirically as a good trade-off between performance and energy.

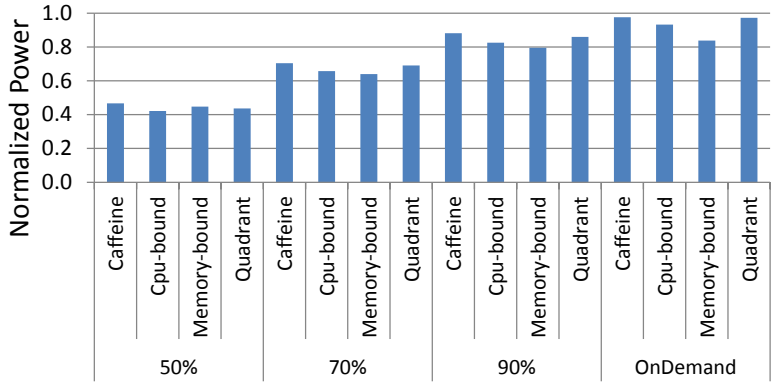


Figure 4.13: Normalized CPU power for the mode that maintains the power below target power.

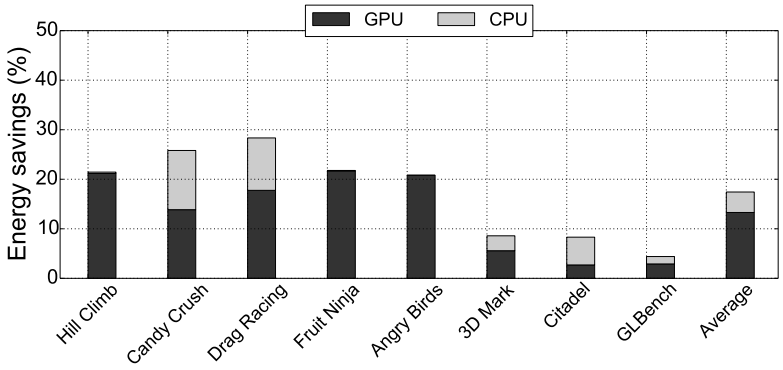
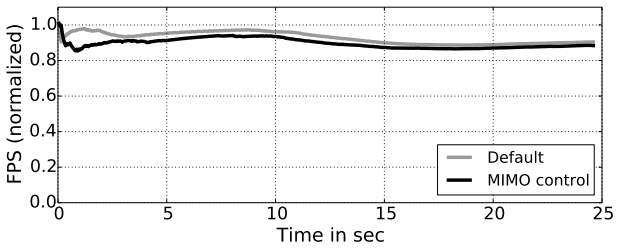


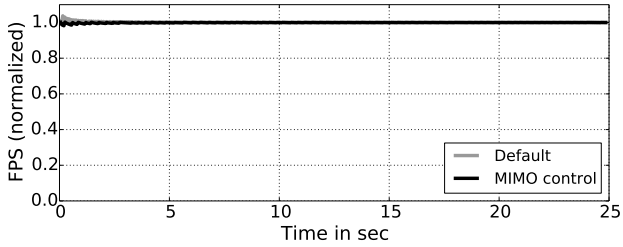
Figure 4.14: Energy saving per-frame.

4.4.2 Results and Analysis

Figure 4.14 shows the total (CPU+GPU) energy savings per frame compared to the default policy. The results show a significant combined GPU+CPU average energy savings of 17.4% across all the benchmarks. Our technique outperforms the default policy on all workloads. On average, the GPU's contribution to the energy savings is 13.3% while the CPU's contribution is 4.1%. A larger savings came from the GPU because it typically consumes more energy than the CPU during graphic



(a) GLBench



(b) Angry Birds

Figure 4.15: Frame-rate over time normalized to the target rate.

applications. The savings are more pronounced on the realistic applications; ranges from 20.7% on *Angry Birds* to 28.3% on *Drag Racing*. As expected, the graphic intensive tests show less energy savings: the highest being 9% for *3DMark* and the lowest 4% for *GLBench*.

With the GPU frequency set to the maximum, the frame-time of graphic intensive benchmarks is close to, and in some cases larger than, the target frame-time. Consequently, our controller has no room for energy optimization as it tries to satisfy the QoS. As a result, the opportunity for energy saving running the performance test applications is minimal. Our controller tries to find the energy optimal frequencies for the CPU and GPU without violating the target frame-time. The energy savings observed in the performance test benchmarks come from the exploitation of the few available less intensive graphic phases. Our controller reacts to those phases by lowering the GPU frequency toward the optimal point. Figure 4.15a shows the normalized

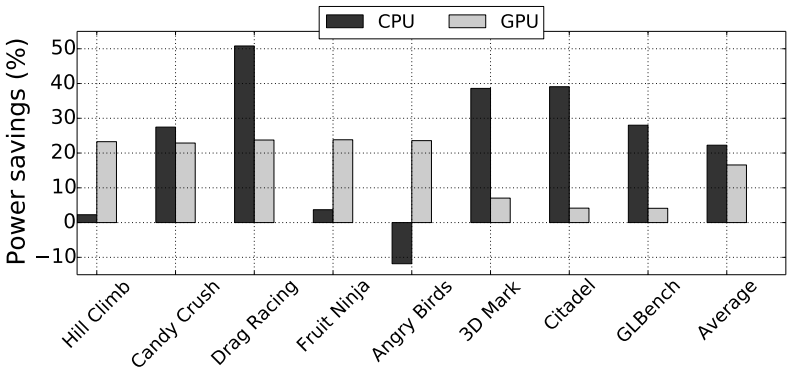


Figure 4.16: CPU and GPU power savings.

instantaneous frame-rate for *GLBench*, one of the graphic intensive workloads. Even though the default policy maintains the GPU at the maximum, the frame-rate is just slightly lower than the target frame-rate. Hence, our controller does not have room for performance slack. The small savings observed for *GLBench* is due to pockets of low graphic intensive phases, mostly at the beginning of the benchmark. Alternately, the realistic workloads allow for greater energy savings. Although the default policy set the CPU and GPU frequencies lower than the maximum, the frame processing time is still shorter than the target. As such, our controller utilizes the available slack to select a more energy efficient operating frequencies for the CPU and GPU without violating the QoS target. Figure 4.15b shows the FPS over time for *Angry Birds*. The frame-rate with both the default policy and our proposed technique are identical. The energy saving observed in *Angry Birds* comes from the fact that our technique requested a much lower GPU frequency than the default policy. The results in Figures 4.15a and 4.15b also show that the FPS over time is stable, reflecting the stability of the control despite its aggressive energy optimization decisions.

The average CPU and GPU power savings are depicted in Figure 4.16. The realistic applications exhibit the most GPU power savings; mostly because the GPU

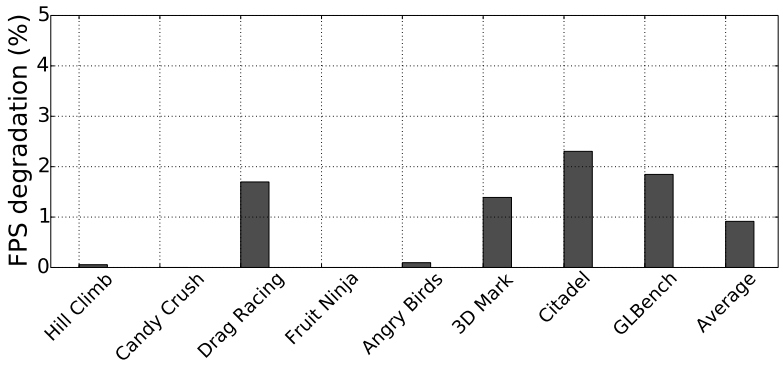


Figure 4.17: Performance overhead.

frequency (and hence voltage) is set lower than the default policy for these applications. On the other hand, graphic intensive benchmarks exhibit less power savings, compared to the default policy, since the frequency must be maintained at the maximum value to achieve the target QoS. We observe an average GPU power saving of 17% and CPU power saving of 20% across all benchmarks. Unlike the GPU case, the most saving for CPU power is observed in graphic intensive benchmarks. Since the GPU is on the critical path of these application performance, the CPU frequency can be reduced without impact to the performance.

The performance overhead of our solution is depicted in Figure 4.17. We observe an insignificant FPS degradation of a 0.9% on average across all benchmarks. Our analysis indicates that more than 50% of this overhead is a consequence of our implementation and mostly related to the query mechanism of the GPU performance counters; a performance request command is inserted into the GPU ring-buffer, disrupting the rendering commands flow. A more efficient implementation which would minimize this overhead is part of our future work. As expected, The results also indicate that graphic intensive workloads exhibit a higher performance degradation than the realistic applications.

4.5 Summary

A platform level framework for power management targeted as mobile devices is presented. The proposed approach reduces the energy consumption while meeting performance constraints. The technique is first applied to the CPU then extend to the CPU and GPU for systems on chips. We observed energy savings of 17.4% while incurring a low performance impact of 0.9%.

5. PRIOR WORK

5.1 Cache Power Reduction

Two circuit-level approaches to leakage power reduction in caches have been previously proposed. The first, *gated-V_{dd}* or power-gating, introduced by Powell *et al* [55], and employed in much of prior work [58, 35, 78], exploits the stacking effect of placing a high-V_t transistor between the memory cell and *GND*. This method typically incurs a performance penalty as the memory cell loses its data. Power-gating is the underlying method to disable cache in our technique. A second circuit technique, *drowsy cache*, proposed by Flautner *et al* [25], uses dynamic voltage scaling to put the memory cell into "drowsy mode" by reducing its voltage enough to maintain its state. This technique incurs a latency to wake up the drowsy line when being accessed. To date most prior work in microarchitectural cache power management leverages one of these two techniques.

Previous techniques use per-block time based assessments of temporal locality to determine *dead* blocks in the cache [35, 78]. Energy efficiency in these techniques is achieved by shutting down these blocks. These schemes, however, are not adequate for shared LLCs because their temporal locality speculation does not account for evictions due to coherence protocols, and they require large counters due to the low access rates seen in LLCs. Abella *et al* proposed a temporal locality predictor to turn-off individual L2 cache lines. Such an approach, however, incurs a very high hardware overhead due to the per-line cache statistics book-keeping hardware [1]. Yang *et al* present a technique to reduce the leakage power in the instruction cache [75]. Their method, DRi Dynamically Resizable icache, which resizes the instruction cache to fit the application working set by changing the number of active sets, is not applicable

to data caches because the instruction locality estimation used does not map well to data working sets. Cache partitioning mechanisms that control the number of active ways in a set associative cache to adjust its effective size and, in turn, to find a good trade-off between performance and power have been proposed. Albonesi *et al* examine the impact of such a technique on performance and power [2]. This technique, however, requires software support to find the necessary cache capacity. Most recently, Sato *et al* [58] proposed a LLC leakage power management technique by partitioning the cache among active threads and activate or deactivate individual ways according to the estimated threads locality. Their technique constructs a stack distance profiling on the LLC and compute the application locality as the ratio of the first and last stack. The drawback of this technique is that a partition of the cache may be underutilized while the other partition is thrashed since threads exhibit different phases at different times. It, further, fails to take into account the application overall reuse behavior.

In this work, we propose an intelligent runtime migration of temporal locality blocks to facilitate power gating while minimizing the performance impact. We evaluate our technique against the state of the art technique proposed by Sato *et al* for energy and performance.

5.2 Cache Prefetching

This section describes the related work in cache prefetching as well as other control-flow speculative techniques. We classify data prefetchers into two classes, *light-weight* and *heavy-weight*. The light-weight prefetchers have low hardware overhead in terms of state required by the prefetcher. They often suffer from relatively low accuracy and performance. On the other hand, the heavy-weight prefetchers provide high accuracy with substantial performance improvement, at the cost of large

amounts of off-chip meta-data memory or additional OS and compiler support.

5.2.1 Light Weight Prefetchers

Data Prefetching techniques have been explored extensively as a means to tolerate the growing gap between processor and memory access speeds. Two widely used prefetchers are "Next-n Lines" [61] and *Stride* [14], both of which capture *regular* memory access patterns with very low hardware overhead. The "Next-n Lines" prefetcher simply queues prefetches for the next n lines after any given miss, under the expectation that the principle of spatial locality will hold and those cache lines after a missed line are likely to be used in the future. The stride prefetcher is slightly more sophisticated; it attempts to identify simple stride reference patterns in programs based upon the past behavior of missing loads. Similar to the "Next-n Lines" technique, when a given load misses, cache lines ahead of that miss are fetched in the pattern following the previous behavior in the hope of avoiding future misses. Both prefetchers have been widely used due to the simple design and low hardware overhead. However, without further knowledge about temporal locality and application characteristics, these prefetchers cannot do more than detecting and prefetching regular memory access patterns with limited spatial locality.

Somogyi *et al.* proposed one of the current top-performing, practical, low-overhead prefetchers, the Spatial Memory Streaming (*SMS*) prefetcher [63]. *SMS* leverages code-based correlation to take advantage of spatial locality in the applications over larger regions of memory (called spatial regions). It predicts the future access pattern within a spatial region around a miss, based on a history of access patterns initiated by that missing instruction in the past. While the *SMS* prefetcher is effective, it is indirectly inferring future program control-flow when it speculates on the misses in a spatial region. As a result, the state overheads of this predictor can be higher than

the others in this class.

Generally, while these light-weight prefetching techniques are quite efficient in terms of storage state versus the performance improvement they provide, they have some disadvantages. In all cases they cannot predict the first misses to a region, and further, they achieve relatively low accuracy for irregular accesses. In the context of chip-multiprocessors with shared LLCs, this low accuracy can even cause performance loss, as inaccurate prefetch streams from one application knock out useful data from another, the "friendly fire" scenario outlined by Jerger, *et al.* [20] and Wu, *et al.* [73].

5.2.2 Heavy Weight Prefetchers

To overcome the disadvantages of *SMS*, Somogyi *et al.* proposed an extension called Spatio-Temporal Memory Streaming (STeMS) [62]. STeMS exploits temporal access characteristics over larger spatial regions and finer access patterns within each spatial region to re-create a temporally ordered sequence of expected misses to prefetch. Exploiting both temporal and spatial characteristics, it improves the performance by 3% over the *SMS* scheme. This performance benefit is achieved at the expense of a large storage overhead (on the order of several megabytes). To manage this overhead, STeMS keeps most of the meta-data off-chip at any given time, shuttling it on- and off-chip as the program goes through its execution phases [70].

Roth *et al.* proposed a novel prefetching technique for pointer based data structures which extracts a simplified kernel of the data's pointer reference structure and executes it without the intervening instructions [57]. While this is effective for these types of data structures and uneven memory accesses, it provides no benefit for other types of code. The recently proposed Irregular Stream Buffer (ISB) prefetcher introduced by Jain and Lin [31] also attempts to capture irregular memory access patterns.

The key idea of the ISB prefetcher is to use an extra level of indirection to create a new structural address space in which correlated physical addresses are assigned consecutive structural addresses. In doing so, streams of correlated memory addresses are both temporally and spatially ordered in this structural addresses space. Thus, the problem of irregular prefetching is converted to sequential prefetching in structural address space. Although the ISB prefetcher shows reasonable performance improvement with less overhead than STeMS, it still requires 8MB of off-chip storage for off-chip meta-data. In addition to that, ISB sees 8.4% memory traffic overhead due to meta-data accesses which does not occur in light-weight prefetchers.

Generally these heavy-weight prefetching techniques show very high accuracy. However, these advantages come at a high cost in terms of meta-data overheads. In energy/power constrained environments it may not be feasible to implement such prefetchers.

5.2.3 Branch Directed and Related Techniques

Prior branch-prediction directed prefetching have focused on simple augmentations to the stride based prefetchers [54, 46], with significantly lower performance than current best of class, light-weight prefetchers (*e.i. SMS*). Although these techniques often accurately speculate on which loads will occur, their performance tends to be poor because they do not accurately speculate on the effective address of those loads. In Tango [54], effective addresses from the last execution of a load are combined with offsets to produce the new expected value, using a technique similar to the traditional value speculation techniques. A key insight and novelty of our technique is, for prefetching, effective address values can be predicted more accurately based upon their variance from the *current architectural state* at an earlier BBs, as opposed to an offset off the *previously generated effective address*, determined by the

last execution of that instruction.

Some early work exists in branch-directed, instruction cache prefetching [66, 56, 11, 65]. Recent work by Ferdman, *et al.*, focusing on server and commercial workloads, shows quite significant gains for instruction cache prefetching [23, 22]. We view these approaches as mostly orthogonal, and potentially complimentary to our data cache prefetching design. In our future work we plan to examine how our path confidence estimation scheme might be used to further improve instruction prefetching.

Our technique bears a passing similarity to Runahead execution-based techniques [51, 19]. Runahead execution effectively functions as a prefetcher by speculatively executing past stalls incurred by long-latency memory instructions. While this approach can be effective at producing a prefetch address stream, it incurs a huge cost in terms of energy. In the presence of a long-latency load, a typical core would idle, once the ROB is full, saving energy. In runahead approaches, the processor continues execution a full speed, potentially wasting significant energy. The *B-Fetch* prefetch pipeline is much smaller and lower complexity than the main pipeline, thus it incurs a much lower energy cost in operation. Furthermore, the prefetch pipeline acts independently and simultaneously with the right-path instructions and need not wait for miss-induced stalls to become active. We are aware of no existing prefetcher design which uses a branch predictor to speculate on control-flow, combined with effective address speculation based upon current architectural state in a light-weight prefetcher design.

Among the numerous existing prefetchers, we compare *B-fetch* with *Stride* and *SMS* prefetchers, two other prefetchers in the *light-weight* class. Both *Stride* and *SMS* prefetchers are considered light-weight because they show substantial performance improvement with reasonable hardware complexity. *SMS* has a more sophisticated

design than that of the *Stride*, but its additional hardware cost (approximately 37KB) is low enough to be implemented entirely on-chip.

5.3 Platform Power Management

Power management policies in traditional computers have been extensively studied; from the earlier work presented by Flautner and Mudge [24] to more recent studies [52, 40, 29]. Ayoub *et al* [5] presented a DVFS policy for power minimization and performance optimization on servers. Current DVFS policies used in mobile devices are direct port from the ones designed for traditional desktops. These policies, in general, do not address system level energy optimization since desktops do not have a strict power limitation. Thus, desktop oriented DVFS policies are ill applicable to battery powered mobile platforms. With the proliferation of mobile devices, there have been considerable effort towards designing energy aware DVFS policies. However, a number of these policies are applications specific [16, 36, 18]. Wu and Li [72] proposed a policy to reduce power while maintaining maximum performance. However, their technique requires an offline profiling of applications. Wu *et al* [74] presented a dynamic compilation DVFS framework. They intercept the application execution and insert DVFS instructions into the program binary at runtime to change the CPU frequency and voltage. Their method requires a dynamic compiler layer which introduces additional performance overhead. Another limitation of their technique is that it is specific to a particular Instruction Set Architecture (ISA). Porting such a technique to other processors requires the existence of a just in time (JIT) compiler software layer for the target ISA. Carta *et al* [9] presented a control theoretic approach to minimizing the energy on MPSoCs with throughput guarantees. However, their implementation requires micro-architecture changes to the processor since their technique exploit the presence of buffers within the pipeline stages. Lee

et al [44] presented a combination of both DVFS and clock gating to reduce power in muliticore systems in desktops and mobile devices. Choi *et al* [17] proposed a DVFS policy to reduce the power dissipation in low power devices through workloads decomposition with the available hardware performance counters. However, their method presents no guarantee to maintaining a performance QoS target.

We present a technique to optimize the platform energy consumption in mobile systems. Our technique accounts for the inefficiency in the power management IC. We construct power and performance models using the information from the hardware performance counters. In addition, we minimize the platform energy consumption while achieving performance and power guarantees.

5.3.1 CPU-GPU Optimization

Most of the research on CPU-GPU heterogeneous systems has been geared towards the acceleration and improvement of specific GPGPU algorithms. These studies advocate distributing tasks among the CPU and GPU [13, 68, 12, 33, 77, 45]. Some work has been oriented towards energy optimization. Suda and Ran [67] proposed a method to estimate the power consumption of CUDA kernels in discrete GPU systems for the purpose of energy optimization. Wang and Ran [69] proposed a power efficient work distribution for CPU-GPU heterogeneous systems. They associate tasks distribution with frequency scaling. The tasks are distributed among the CPU and GPU to reduce applications execution time. Komoda *et al.* [41] presented a DVFS technique that embodies a coordinated DVFS, a task mapping technique and a load balance distribution. They mapped tasks to either CPU or GPU, hence controlling the load balance, and set the frequencies accordingly. Chiesi *et al.* [15] also proposed a similar scheduling technique while maintaining the power under constraints. All of the work cited above focuses on general purpose computing workloads.

These workloads allow the partition of applications into tasks; Hence facilitating the mapping of these tasks to either the CPU or GPU. These techniques, however, cannot be applied to 3D graphic applications where the tasks mapping cannot be altered. In these applications, typically, the GPU renders the images while the CPU processes the scene, handles physics emulation.

Most recently, Pathania *et al.* proposed a power management scheme in CPU-GPU systems for 3D graphic applications on mobile devices [53]. They performed an offline analysis on the impact of the different CPU-GPU frequencies combinations on applications performance and the system power. They, then, implement a heuristic method that tries to find the correct CPU-GPU frequencies combination to meet the target QoS. Their technique is reactive; that is, it cannot predict the GPU-CPU performance nor the power at specific frequencies. Furthermore, the set of frequency combinations grows exponentially with increasing number of CPU and GPU frequencies steps.

Our technique models the interaction between CPU and GPU in order to achieve a good synchronization of both compute engines. We construct power and performance models using the information from the hardware performance counters. These models allow us to make informed decisions on frequencies for both the CPU and GPU to achieve the target QoS.

6. CONCLUSIONS

We proposed a solution to reduce the LLC leakage energy dissipation in CMPs while mitigating performance impact. We compute the program's memory footprint and migrate the useful data to facilitate cache resizing. Using a cycle accurate simulator, we evaluate our proposed solution for single-threaded, multiprogrammed and multithreaded workloads and observe significant energy savings ranging from 66% to 50%, experiencing a low performance degradation of 2.16% average, while requiring a small 3.64% hardware overhead. We also proposed *B-Fetch*, a data prefetcher that takes advantage of control flow speculation in the branch predictor to accurately generate data prefetches. *B-Fetch* utilizes the strong correlation between the effective addresses generated by memory instructions and the values of the corresponding source registers at prior branch locations. *B-Fetch* leverages a copy of architectural state of registers at the time of the prefetch together with learned knowledge of the register transformations which occur over BBs to generate accurate and timely prefetches for data exhibiting both regular and irregular access patterns. *B-Fetch* achieves a mean speedup of 23.0% over a baseline and outperforms the state-of-the-art prefetcher, while incurring a minimal additional hardware cost. A power management policy for mobile platforms is presented. The proposed approach reduces the energy consumption while maintaining a QoS performance and satisfy power budget constraints. We extend our platform to include the GPU and optimize energy consumption for 3D graphic applications. The interaction between the CPU and GPU and reduces the energy consumption by synchronously managing the CPU and GPU frequency. The implementation of the proposed techniques on an Atom based Android tablet shows platform energy savings of up to 8% and CPU energy savings of up to 15%. When

accommodating for the GPU the energy saving increases to 17.4%; while incurring a low performance impact of 0.9%.

REFERENCES

[1] J. Abella, A. Gonzalez, X. Vera, and M. F. P. OBoyle. IATAC: a smart predictor to turn-off L2 cache lines. *Transactions on Architecture and Code Optimization*, 2(1):55–77, 2005.

[2] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *The 32nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 248–259, 1999.

[3] Hrishikesh Amur, Ripal Nathuji, Mrinmoy Ghosh, Karsten Schwan, and Hsien hsin S. Lee. Idlepower: Application-aware management of processor idle states. In *Proceedings of MMCS, in conjunction with HPDC 08*, 2008.

[4] Raid Ayoub, Rajib Nath, and Tajana Rosing. Jetc: Joint energy thermal and cooling management for memory and cpu subsystems in servers. In *The 18th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–12, 2012.

[5] Raid Ayoub, Umit Ogras, Eugene Gorbatov, Yanqin Jin, Timothy Kam, Paul Diefenbaugh, and Tajana Rosing. Os-level power minimization under tight performance constraints in general purpose systems. In *The International Symposium on Low Power Electronics and Design (ISLPED)*, pages 321–326, 2011.

[6] Luciano Bertini, Julius C.B.Leite, and Daniel Mosse. Power optimization for dynamic configuration in heterogeneous web server clusters. In *Journal of Systems and Software*, pages 585–598, 2010.

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, So-

mayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammand Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comp. Arch. News*, 39:1–7, May 2011.

[8] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *USENIXATC*, pages 21–21, 2010.

[9] Salvatore Carta, Andrea Alimonda, Alessandro Pisano, Andrea Acquaviva, and Luca Benini. A control theoretic approach to energy-efficient pipelined computation in mpsocs. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(4):27, 2007.

[10] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip-multiprocessor architecture. In *The 11th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 340–351, 2005.

[11] I-CK Chen, Chih-Chieh Lee, and Trevor N Mudge. Instruction prefetching using branch prediction information. In *IEEE International Conference on Computer Design (ICCD)*, pages 593–601. IEEE, 1997.

[12] Linchuan Chen, Xin Huo, and Gagan Agrawa. Accelerating mapreduce on a coupled cpu-gpu architecture. In *International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 223–238, 2012.

[13] Shuo Chen and Xiaoming Li. A hybrid gpu/cpu fft library for large fft problems. In *Performance Computing and Communications Conference*, pages 1–10, 2013.

[14] Tienfu Chen and Jeanloup Baer. Effective hardware-based data prefetching for high-performance processors. *Transactions on Computers*, 44:609–623, 1995.

[15] Matteo Chiesi, Luca Vanzolini, Claudio Mucci, Eleonora Franchi Scarselli, and Roberto Guerrieri. Power-aware job scheduling on heterogeneous multicore architectures. In *Transactions on Parallel and Distributed Systems*, pages 868–877, 2014.

[16] Keng-Mao Cho, Chun-Hung Liang, Jun-Ying Huang, and Chu-Shing Yang. Design and implementation of a general purpose power-saving scheduling algorithm for embedded systems. In *ICSPCC*, pages 1–5, 2011.

[17] Kihwan Choi, Wonbok Lee, R. Soma, and M. Pedram. Dynamic voltage and frequency scaling under a precise energy model considering variable and fixed components of the system power dissipation. In *ICCAD*, pages 29–34, 2004.

[18] Morteza Damavandpeyma, Sander Stuijk, Twan Basten, Marc Geilen, and Henk Corporaal. Throughput-constrained dvfs for scenario-aware dataflow graphs. In *Real-Time and Embedded Technology and Applications Symposium*, pages 175 – 184, 2013.

[19] James Dundas and Trevor Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *The International Conference on Supercomputing (ICS)*, pages 68–75, 1997.

[20] Natalie D Enright Jerger, Eric L Hill, and Mikko H Lipasti. Friendly fire: understanding the effects of multiprocessor prefetches. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 177–188, 2006.

[21] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011.

[22] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. Proactive instruction fetch. In *44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 152–162, 2011.

[23] Michael Ferdman, Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Temporal instruction fetch streaming. In *41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–10, 2008.

[24] Krisztián Flautner and Trevor Mudge. Vertigo: Automatic performance-setting for linux. *ACM SIGOPS Operating Systems Review*, 36(SI):105–116, 2002.

[25] Krisztin Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy caches: Simple techniques for reducing leakage power. In *Annual International Symposium on Computer Architecture (ISCA)*, pages 148–157, 2002.

[26] Freedesktop.org. Direct rendering manager (drm). `http://dri.freedesktop.org/wiki/DRM/`, 2014.

[27] Freescale. Mc13892: Power management integrated circuit (pmic) for i.mx35/51. `http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MC13892`, 04 2014.

[28] M. Workman G. Franklin, J. Powell. *Digital Control of Dynamic Systems*. Addison-Wesley, 1998.

[29] Sebastian Herbert and Diana Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *The International Symposium on Low Power Electronics and Design (ISLPED)*, pages 38–43, 2007.

[30] HP, Intel, Microsoft, Phoenix Tech, and Toshiba. Advanced configuration and power interface specification, revision 5.0, 2011.

[31] Akanksha Jain and Calvin Lin. Linearizing irregular memory accesses for improved correlated prefetching. In *The 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 247–259, 2013.

[32] Daniel A. Jimenez. Composite confidence estimators for enhanced speculation control. In *The 21st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 161–168, 2009.

[33] Mark Joselli, Marcelo Zamith, Esteban Clua, Anselmo Montenegro, Aura Conci, Regina Leal-Toledo, Luis Valente, and Bruno Feij0. Automatic dynamic task distribution between cpu and gpu for real-time systems. In *International Conference on Computational Science and Engineering*, pages 48–55, 2008.

[34] David Kadjo, Umit Ogras, Raid Ayoub, Michael Kishinevsky, and Paul Gratz. Towards platform level power management in mobile systems. In *IEEE International System-on-Chip Conference*, pages 146–151, 2014.

[35] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Annual International Symposium on Computer Architecture (ISCA)*, pages 240–251, 2001.

[36] Jabran Khan, Sebastien Bilavarn, and Cecile Belleudy. Impact of operating points on dvfs power management. In *Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–6, 2012.

[37] Samira Khan, Yingying Tian, and Daniel A. Jimenez. Sampling dead block prediction for last-level caches. In *The 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 175–186, 2010.

[38] Samira Manabi Khan, Yingying Tian, and Daniel A. Jimenez. Sampling dead block prediction for last-level caches. In *The 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 175–186. IEEE Computer Society, 2010.

[39] Sangman Kim, Indrajit Roy, and Vanish Talwar. Evaluating integrated graphics processors for data center workloads. In *HotPower*, 2013.

[40] Tejaswini Kolpe, Antonia Zhai, and Sachin S. Sapatnekar. Enabling improved power management in multicore processors through clustered dvfs. In *Design Automation and Test in Europe (DATE)*, pages 1–6, 2011.

[41] Toshiya Komoda, Shingo Hayashi, Takashi Nakada, Shinobu Miwa, and Hiroshi Nakamura. Power capping of cpu-gpu heterogeneous systems through coordinating dvfs and task mapping. In *International Conference on Computer Design (ICCD)*, pages 349–356, 2013.

[42] Cyril Kowaliski. Gelsinger reveals details of nehalem, larrabee, dunnington. `http://techreport.com/news/14361/gelsinger-reveals-details-of-nehalem-larrabee-dunnington`, 2008.

[43] H.-H. Lee, G. S. Tyson, and M. Farrens. Eager writeback-a technique for improving bandwidth utilization. In *The 33rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 11–21, 2000.

[44] Jungseob Lee, Chi-Chao Wang, Hamid Ghasemi, Lloyd Bircher, Yu Cao, and Nam Sung Kim. Workload-adaptive process tuning strategy for power-efficient multi-core processors. In *The International Symposium on Low Power Electronics and Design (ISLPED)*, pages 225–230, 2010.

[45] Xue-Xin Liu, Hai Wang, and Sheldon X. Tan. Parallel power grid analysis using preconditioned gmres solver on cpu-gpu platforms. In *International Conference on Computer-Aided Design (ICCAD)*, pages 561–568, 2013.

[46] Yue Liu and David R. Kaeli. Branch-directed and stride-based data cache prefetching. In *The International Conference on Computer Design*, International Conference on Computer Design (ICCD), pages 225–230, 1996.

[47] K. Malik, M. Agarwal, V. Dhar, and M.I. Frank. Paco: Probability-based path confidence prediction. In *The 14th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 50–61. IEEE, 2008.

[48] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[49] David Meisner, Brian T. Gold, and Thomas F. Wenisch. Powernap: eliminating server idle power. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 205–216, 2009.

[50] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *The 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 3–14, 2007.

[51] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors. In *The 9th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 129–140, 2003.

[52] V. Pallipadi and A. Starikovskiy. The ondemand governor: Past, present, and future. In *Linux Symposium*, pages 223–238, 2006.

[53] Anuj Pathania, Qing Jiao, Alok Prakash, and Tulika Mitra. Integrated cpu-gpu power management for 3d mobile games. In *Conference on Design Automation Conference*, pages 1–6, 2014.

[54] S.S. Pinter and A. Yoaz. Tango: a hardware-based data prefetching technique for superscalar processors. In *The 29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 214–225. IEEE Computer Society, 1996.

[55] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. Gated-Vdd : a circuit technique to reduce leakage in deep-submicron cache memories . In *The International Symposium on Low Power Electronics and Design (ISLPED)*, pages 90–95, 2000.

[56] G. Reinman, B. Calder, and T. Austin. Fetch directed instruction prefetching. In *The 32nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 16–27, 1999.

[57] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence based prefetching for linked data structures. In *The Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 115–126, 1998.

[58] Masayuki Sato, Ryusuke Egawa, Hiroyuki Takizawa, and Hiroaki Kobayashi. A voting-based working set assessment scheme for dynamic cache resizing mechanisms. In *International Conference on Computer Design (ICCD)*, pages 98–105, 2010.

[59] Andre Seznec, Stephen Felix, Venkata Krishnan, and Yiannakis Sazeides. Design tradeoffs for the alpha ev8 conditional branch predictor. In *The 29th Annual In-*

ternational Symposium on Computer Architecture(ISCA), pages 295–306. ACM, 2002.

[60] Findlay Shearer. *Power management in mobile devices.* Newnes, 2011.

[61] A. J. Smith. Sequential program prefetching in memory hierarchies. *Computer*, 11:7–21, December 1978.

[62] Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki, and Babak Falsafi. Spatio-temporal memory streaming. In *Annual International Symposium on Computer Architecture (ISCA)*, pages 69–80, 2009.

[63] Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. Spatial memory streaming. In *The 39th Annual International Symposium on Computer Architecture(ISCA)*, pages 252–263, 2006.

[64] Stephen Somogyi, Thomas F. Wenisch, Michael Ferdman, and Babak Falsafi. Spatial memory streaming. *Journal of Instruction-Level Parallelism*, 13, 2011.

[65] Lawrence Spracklen, Yuan Chou, and Santosh G Abraham. Effective instruction prefetching in chip multiprocessors for modern commercial applications. In *The 11th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 225–236. IEEE, 2005.

[66] Viji Srinivasan, Edward S. Davidson, Gary S. Tyson, Mark J. Charney, and Thomas R. Puzak. Branch history guided instruction prefetching. In *The 7th International Conference on High Performance Computer Architecture (HPCA)*, pages 291–300, 2001.

[67] Reiji Suda and Da Qi Ren. Accurate measurements and precise modeling of power dissipation of cuda kernels toward power optimized high performance

cpu-gpu computing. In *Parallel and Distributed Computing, Applications and Technologies*, pages 432–438, 2009.

[68] George Teodoro, Tahsin M. Kurc, Tony Pan, Lee A.D. Cooper, Jun Kong, Patrick Widener, and Joel H. Saltz. Accelerating large scale image analyses on parallel, cpu-gpu equipped systems. In *Parallel and Distributed Processing Symposium*, pages 1093–1104, 2012.

[69] Guibin Wang and Xiaoguang Ren. Power-efficient work distribution method for cpu-gpu heterogeneous system. In *Parallel and Distributed Processing with ApplicationsDesign*, pages 122–129, 2010.

[70] Thomas F. Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Practical off-chip meta-data for temporal memory streaming. In *The 15th International Symposium on High Performance Computer Architecture (HPCA)*, pages 79–90, 2009.

[71] D. A. Wood, M. D. Hill, , and R. E. Kessler. A model for estimating trace-sample miss ratios. In *SIGMETRICS*, pages 79–89, 1991.

[72] Bin Wu and Peng Li. Load-aware stochastic feedback control for dvfs with tight performance guarantee. In *VLSI and System-on-Chip (VLSI-SoC)*, pages 231–236, 2012.

[73] Carole-Jean Wu, Aamer Jaleel, Margaret Martonosi, Simon C. Steely, Jr., and Joel Emer. Pacman: Prefetch-aware cache management for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture(MICRO)*, pages 442–453, 2011.

[74] Qiang Wu et al. A dynamic compilation framework for controlling microprocessor energy and performance. In *The 38th Annual IEEE/ACM International*

Symposium on Microarchitecture (MICRO), pages 12 pp.–282, 2005.

[75] S.-H. Yang, M. D. Powell, B. Falsafi, K. Roy, and T. N. Vijaykumar. An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance I-caches . In *The 7th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 147–157, 2001.

[76] Daecheol You and K.-S. Chung. Dynamic voltage and frequency scaling framework for low-power embedded gpus. In *Electronics Letters*, pages 1333–1334, 2012.

[77] Xiang Jun Zhao, MeiZhen Yu, and Yong Beom Cho. Gpu-cpu based parallel architecture for reduction in power consumption. In *Global High Tech Congress on Electronics*, pages 182–185, 2012.

[78] H. Zhou, M. C. Toburen, E. Rotenberg, and T. M. Conte. Adaptive mode control: a static-power-efficient cache design. *Transactions on Embedded Computing Systems*, 2(3):347–372, 2003.