DYNAMIC CACHE RECONFIGURATION FOR ENERGY OPTIMIZATION IN CHIP MULTIPROCESSORS

by

GAURAV PURI

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN COMPUTER ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2012

ACKNOWLEDGEMENTS

I am grateful to my thesis supervisor, Dr. Ishfaq Ahmad for his invaluable guidance from the very beginning. I would like to thank Mr. David Levine and Dr. Gergely Zaruba for taking time out of their busy schedules for serving on my committee.

I would like to thank my family and friends for their enduring support. Special thanks to Mr. Dennis O'Hare, my work supervisor at SESPMG, UT Arlington for allowing me to keep a flexible work schedule without which this thesis would not have been possible.

April 23, 2012

ABSTRACT

DYNAMIC CACHE RECONFIGURATION FOR ENERGY

OPTIMIZATION IN CHIP MULTIPROCESSORS

Gaurav Puri, M.S.

The University of Texas at Arlington, 2012

Supervising Professor: Ishfaq Ahmad

Cache reconfiguration for chip multiprocessors (CMP) in order to reduce energy consumption is a challenging research problem. This thesis presents a dynamic scheme for level one cache design where the processing core can dynamically switch to a lower energy configuration. The proposed scheme is based on a heuristic that can either be executed as part of the firmware or implemented in hardware with very low additional overhead. The heuristic functions periodically after a pre-defined fixed interval for each processing core. The heuristic first examines the statistics calculated over the previous intervals and then switches the cache to an improved configuration if it can lead to energy savings. We evaluate our approach extensively by testing benchmarks from the PARSEC 2.1, MiBench and SPLASH-2 benchmark suites on a simulated 4-core CMP using a modified Multi2Sim 3.2.1 simulator. The results indicate that the proposed scheme provides on average 16.92% savings in energy consumption with only a 3.01% reduction in instructions per cycle as compared to a regular 16KB 4-way L1 cache.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure Page

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Energy consumption in embedded microprocessors

Energy consumption is one of the most, if not the most, important consideration in the design of an embedded system. For battery operated devices, the battery life is directly impacted by the energy consumption - an increase in energy consumption causes a corresponding reduction in battery life. Modern microprocessors used in current smartphones have been pushing the boundaries of an embedded processor and can have power consumption of a few watts [1] which is a major step up from the few milliwatts of power consumption of most embedded processors. Thus, it is paramount to keep the power / energy consumption in check to ensure long battery life in such devices.

1.1.1 Power and Energy consumption in CMOS circuits

Power consumption in CMOS circuits [2] is due to dynamic power and static power.

1.1.1.2 Dynamic power

Dynamic power consumption in a CMOS circuit arises due to a circuit switching between two voltage states. It is dissipated only when such a switch occurs and depends directly on the frequency of switch. Dynamic power consumption of a CMOS circuit can be approximated as:

$$P_{dynamic} = \alpha \cdot C \cdot V^2_{dd} \cdot f \quad (1.1)$$

Where,

C is the capacitive load switched

V_{dd} is the supply voltage

α is the activity factor representing the fraction of the circuit that is switching, often approximated as being 1/2.

f is the clock frequency

Using α = 1/2, we can calculate the dynamic power dissipation as:

$$P_{dynamic} = \frac{1}{2} \cdot C \cdot V^2_{dd} \cdot f$$

1.1.1.3 Static power

Static power is dissipated in a CMOS circuit is due to the fact that a transistor cannot be completely turned off. Leakage power is the primary contributor to static power. The static power dissipation in a CMOS circuit can be approximated as:

$$P_{static} = N \cdot I_{leakage} \cdot V_{dd} \quad (1.2)$$

Where,

N is the number of devices leaking current

$I_{leakage}$ is the per-device leakage current

V_{dd} is the supply voltage

Until a few years ago, dynamic power used to be the major contributor to the overall power consumption of a device. However, with the continued shrinking of transistor sizes with every successive semiconductor technology generation resulting in the move to the deep-submicron era, static power has overtaken dynamic power as the primary contributor to the overall power dissipation as the width of the gate gets thinner with every successive device shrink. This trend can be observed by examining power consumption data for caches.

Given power consumption data, energy consumption can be simply calculated as the product of power and duration of operation, T. Therefore,

$$E_{dynamic} = \alpha \cdot C \cdot V^2_{dd}$$

$$E_{static} = N \cdot I_{leakage} \cdot V_{dd} \cdot T$$

And, total energy consumption,

$$E = E_{dynamic} + E_{static}$$

Or,

$$E = (\alpha \cdot C \cdot V_{dd}^2) + (N \cdot I_{leakage} \cdot V_{dd} \cdot T) \quad (1.3)$$

Given the relationship between power and energy it might seem that minimizing power would also minimize energy. However, this may not always be the case as can be seen in the following example:

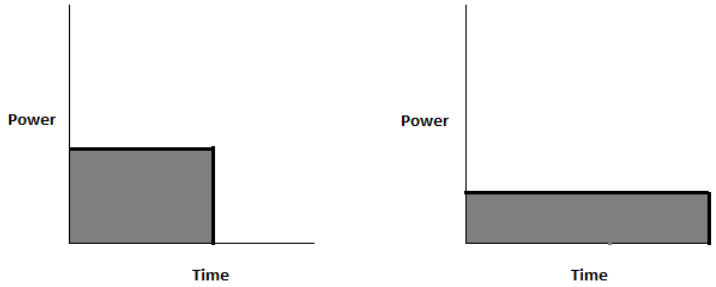

Figure 1.1 Relationship between power and energy consumption. Figure not drawn to scale. The shaded area inside the graphs represents the total energy consumption.

As shown in Figure 1.1 above, power consumption in the second case is half of the power consumption in the first case. However, the task also takes twice as long in the second case. Therefore, energy, which is the product of power and time, remains the same.

1.1.2 Contribution of cache to energy consumption

On chip caches have been found to be responsible for over 50% of the power consumption of a microprocessor [3]. This makes cache the foremost candidate for energy optimization. The advent of chip multiprocessors (CMPs) has further increased this contribution as most CMP designs employ an L1 cache per core as well as a shared L2 cache. Energy is consumed whenever a tag is matched with the contents of the cache tag directory and whenever data is read out on a cache hit from the data array. Whenever a match is done, a differential voltage is developed in the internal bitlines which is amplified by the sense amplifiers (SA) which is fed into a comparator to determine if it is a hit or not [2]. For a set-associative cache, this process is done either in parallel for all ways of the cache, or if way prediction [4] is

employed, a match is made with the predicted way and the rest of the ways are searched only if no matching tag was found in the predicted way.

Due to the growing disparity between microprocessor and memory speed with every successive semiconductor technology generation [5], the cost of a cache miss has also increased substantially. Therefore, a large number of microprocessor cycles are wasted whenever a cache miss is encountered since the microprocessor needs to be stalled until the read from the main memory is completed. This also results in a significant amount of energy consumption, E_{miss}, which is much greater than that of a cache hit.

The static and dynamic components of the cache energy consumption can be approximated using the following equations [6]:

$$E_{dynamic} = N_{accesses} \cdot E_{access} + N_{misses} \cdot E_{miss} \quad (1.4)$$

$$E_{static} = P_{static} \cdot N_{cycles} \cdot t_{cycle} \quad (1.5)$$

Where,

$N_{accesses}$ is the number of accesses made to the cache

E_{access} is the energy consumed every time the cache is accessed

N_{misses} is the number of times a cache miss occurred during program execution

E_{miss} is the additional energy consumed because of a cache miss

N_{cycles} is the number of clock cycles required for the execution of the program

t_{cycle} is the time period of a clock cycle

For the purpose of this research, we obtained the E_{access} and P_{static} for the examined cache configurations using CACTI [7]. E_{miss} can be approximated as,

$$E_{miss} = E_{next\_level\_access} + E_{\mu P\_stall} + E_{block\_fill}$$

Where,

$E_{next\_level\_access}$ is the energy consumed due to accessing the next levels in the memory hierarchy which may be on-chip (L2) or off-chip (main memory)

$E_{\mu P\_stall}$ is the energy consumed while the microprocessor is stalled while the data is being fetched from the backing store

E_{block_fill} is the energy consumed while filling up a cache block

Calculating the terms $E_{next\_level\_access}$ and $E_{\mu P\_stall}$ and hence E_{miss} can be tricky. These terms depend largely on the specific microprocessor used and the specifics of the off-chip memory used.

1.2 Understanding access times and energy consumption in caches

The energy consumption and access times of a cache depend heavily on the cache size and the number of ways (associativity). The effect of block size is dependent upon the application under consideration.

1.2.1 Direct mapped vs. set associative caches

Set associative caches can reduce miss rate as compared to an equal sized direct mapped cache at the expense of greater hit times as well as power consumption [8] [9].

The increased power consumption in set-associative caches can be attributed to the parallel search in multiple tag arrays, the simultaneous data read out in parallel from multiple data arrays as well as the presence of an encoder, multiplexer and additional circuitry which is required to support such an operation. The increased access times are due to the inability of selecting the right data block before finding out the result of the tag comparison, additional encoder in the critical path to select the correct way and the multiplexer to select the correct data output.

A higher associativity is beneficial only if a reduction in miss rate and the corresponding reduction in dynamic energy consumption balance out the additional increase in energy consumption due to the higher associativity.

1.2.2 Smaller cache size vs. larger cache size

A larger cache obviously consumes more static energy than a smaller cache. However, a lower miss rate in a larger cache would result in a lower dynamic energy consumption which

might result in a reduction in the overall energy consumption relative to a smaller cache. Moreover, for large caches, the access times can be substantially higher than a small cache due to increased wire delay. To reduce access times, large caches are often partitioned into multiple subarrays, referred to as subarray partitioning [10].

1.2.3 Smaller block size vs. larger block size

Energy consumption per hit is always more for larger block sizes since more words are fetched. This also means increased access time during a miss. The access time for a hit is not affected. However, these effects are amortized over the runtime of an application if a reduction in miss rate is achieved. A larger block size benefits applications having high spatial locality whereas, it can prove detrimental if spatial locality is low since that would result in more misses and would also mean that a lot of data fetched remains unused thereby wasting energy.

1.3 Cache reconfiguration

From the discussion in section 1.2 above, we can conclude that each cache configuration can have varied energy consumption and access time characteristics. Cache reconfiguration can be defined as the process of changing the characteristics i.e., total size, line size and associativity, of the cache memory to fit a given application's requirements while maximizing the value of a given objective. For the purpose of this research, the objective is obtaining a reduction in energy consumption without a significant performance loss. We only consider the reconfiguration of L1 cache in our research.

1.3.1 Determining whether to switch or not

It is important to assess the relative performance characteristics of different cache configurations before switching to a lower energy configuration, which in most cases is also a lower performance configuration. We explain this by means of the following example:

Assume the available configuration space, C to be given as:

C = {16K1W, 16K2W, 16K4W, 8K1W, 8K2W, 4K1W}

Where, each "xKyW" represents a cache of size x KB and associativity y.

Switching from 16K4W to any other configuration would always lower the per-access energy as well as the static energy consumption of the cache. However, if this results in a large performance drop due to a substantial increase in miss rate, the overall energy consumption would actually be much higher since the energy consumed on a miss is several times (tens of times) larger than a hit. Similar argument can be presented while switching from 8K2W to 8K1W or from 8K1W to 4K1W. Therefore, while switching to a lower energy configuration, the performance impact should be assessed.

1.3.2 Switching in presence of a secondary cache

It is also important to note how the addition of a secondary L2 cache can impact the switching decision. If a miss can be served by an L2 cache, the energy consumption of the miss as well as the performance hit due to the miss is reduced by a factor of ten as compared to main memory. Therefore, if most of the misses due to reconfiguration can be served by an L2 cache, the performance hit is reduced. Again, even if all the misses can be served by L2 it does not necessarily mean that the lower L1 energy cache configuration is better. The impact must still be monitored. We just get much more leeway due to the presence of an L2 cache.

Cache inclusion [11] or exclusion [9] can also have an impact on the switching. An inclusive cache hierarchy requires that the data in an upper level cache be a subset of the data in the lower levels. Therefore, a miss in L1 that cannot be served by L2 requires eviction of a block in both L1 and L2. This means substantially more energy consumed if the L2 cannot serve most of the misses. On the other hand, exclusion requires that contents of L1 and L2 be exclusive: a block evicted in L1 causes the block to be loaded into L2. This requires tight synchronization between L1 and L2 caches. Therefore, a configuration switch in L1 would impact the contents of L2.

1.3.3 Switching in chip multiprocessors

Chip multiprocessors (CMPs) usually employ a private L1 I-cache and D-cache per core and a shared unified L2 cache to enable data sharing among different cores. While at first

glance it might seem that the reconfiguration in a CMP would be similar to a uniprocessor with a secondary L2 cache, it actually presents a number of new issues.

Firstly, the L2 cache is shared which means that each of the cores is competing for the maximum available space in L2. Therefore, whether or not misses from a given core's L1 cache can be handled is not dependent on just that core but also on the configurations and execution characteristics of the applications on the other cores.

Secondly, the switching of an L1 core impacts the L2 cache partitioning [12] substantially. If inclusion in L2 cache is maintained and the miss rate of the application is not impacted substantially due to a switch to a lower sized and hence lower energy configuration, we can actually utilize the shared L2 space better since far fewer blocks (equal to the size of the new L1 cache) would need to be cached in L2. This means, far fewer unutilized blocks while maintaining inclusion and hence improved performance.

Lastly, reconfiguration decisions of different L1 caches are not independent. As discussed above, the contents of the L2 cache are impacted by a reconfiguration decision in one of the upper level caches due to the presence of a memory consistency protocol. This change impacts the performance of the applications running on all cores. Therefore, reconfiguration decisions of one core impact the reconfiguration decision of other cores.

CHAPTER 2

RELATED WORK

Several methods for reducing energy consumption, both at the circuit as well as firmware level, have been proposed in literature and many have been successfully implemented in commercial microprocessors. In this chapter, we discuss briefly how each of these methods reduces the overall energy consumption.

2.1 Dynamic Voltage and Frequency Scaling

Dynamic voltage and frequency scaling (DVFS) [2] is the most widely used method for reducing power as well as energy consumption in microprocessors. From Eq. 1.3, we can see that dynamic energy consumption varies quadratically with respect to V_{dd}, whereas leakage energy consumption varies linearly with V_{dd}. However, the degree by which V_{dd} can be lowered is bounded by the threshold voltage, V_t of the CMOS circuit. We cannot arbitrarily lower the supply voltage.

When the workload is low, and there is enough slack time, we can reduce the processor frequency, so that the number of idle cycles is reduced while the work is still finished in a reasonable time. Note that only reducing the processor frequency would reduce the power dissipation but not the energy consumption as was explained earlier in section 1.1.1.

The relationship between V_{dd}, V_t and f can be approximated as [2]:

$$f_{MAX} \sim (V_{dd} - V_t)^2 / V_{dd} \quad (2.1)$$

Therefore, by combining a reduction in V_{dd} with a reduction in the frequency f, we can obtain a reduction in the overall energy consumption while maintaining the threshold voltage.

Various heuristics have been proposed in literature for DVFS. These include general scheduling algorithms which modify the schedule for reducing energy [13], scheduling algorithms for real-time systems [14], and also scheduling algorithms which combine cache

reconfiguration and DVFS in real-time systems [15]. Operating system support is required for DVFS so that it can signal to the CPU to switch to a lower voltage / frequency state.

2.2 Clock gating

Clock gating turns off unused functional blocks which are not needed thereby reducing dynamic power. The functional blocks retain their state through latches and do not draw any current. Since there is no switching activity, their dynamic power consumption is zero. However, static power is still consumed as the functional blocks are still on. In devices like DRAM or disks, they can be put into a sleep mode in which they consume very low power. In CMPs, core gating can be done by putting a core to sleep when its processing power is not needed.

Powell et al. proposed Gated V_{dd} [16] as a means of reducing leakage current in caches by introducing a transistor in series with the SRAM power supply thereby providing a tremendous reduction in the leakage current drawn due to the stacking effect of the transistor. However, the contents of the memory cell are lost.

Kaxiras et al. [17] showed that a lot of cache lines experience heavy initial activity but remain unused thereafter. Also, some lines remain unused for very long periods. They propose the use of gated V_{dd} technique to turn of these “dead” cache lines to save leakage power.

Kim et al. [18] introduced drowsy caches as a means of reducing leakage power in SRAM by providing a second low-voltage power supply to supply power to the cache lines that have been classified as being the infrequently accessed. By supplying a lower than usual supply voltage, a DVS effect is produced which reduces the leakage power. A small performance loss is however, experienced in waking up a drowsy cache line.

2.3 Way prediction

Way prediction [4] is a widely used technique for improving performance as well as reducing energy consumption in a set-associative cache. In a way predicting set-associative cache, tag matching is first done only in the predicted way. The remaining ways are searched only if the match fails in the predicted way. If the prediction accuracy is high, way prediction can

reduce delays due to a set-associative cache relative to a direct mapped cache. Moreover, since only one way is searched initially and if that way has the data most of the time, the other tag and data arrays are not accessed leading to substantial savings in energy. An MRU algorithm is generally employed for way prediction where the MRU information per set is used to speculate the correct way for the incoming tag. In case the predicted way is not the correct way, a delay of one clock cycle is encountered in finding the matching block.

2.4 Single-ISA Heterogeneous multicore architectures

Kumar et al. [19] proposed a heterogeneous multicore processor where each core has the same instruction set but different performance capabilities. These cores can be either designed from scratch or existing cores can be reused. The objective is to select a core for a given application that minimizes the energy-delay product.

Although substantial savings compared to a homogeneous multicore processor having the highest performing core can be achieved using this approach, operating system support is required to assign the right application to each core and also to switch application between cores in case of a phase change or a context switch so that the applications continue to be assigned to the most energy efficient core. Moreover, the mix of applications should be such that each application has differing resource requirements. There is no way to adapt to an application mix where all applications have high performance demands.

2.5 Cache reconfiguration

Cache reconfiguration is a well-studied technique for reducing L1 cache leakage energy. Albonesi [10] proposed a cache architecture which provided the ability to turn off infrequently used ways in a set-associative cache which could be turned back on under periods of high activity thereby providing energy savings with a small hit in performance. The reason that this scheme works is that cache requirements vary widely among different applications and also during different phases of the same application. If an application does not need the extra ways, it is worthwhile to turn off those unused ways for energy savings. Subarray partitioning of

the tag and data arrays is required so that individual banks, each bank representing one way, can be turned off.

Malik et al. [3] introduced the Motorola M Core M340 processor which supported configurable cache ways through a user writable cache control register whose value can be modified in order to enable / disable the ways of the cache.

Zhang et al. [6] introduced a highly configurable cache architecture providing the ability to configure the cache size, associativity and block size by setting bits in configuration registers. Through a method called way concatenation, a cache can be configured as 1, 2 or 4-way while keeping the cache size constant. The choice of associativity for a given cache size is, obviously, limited by the number of banks available. Thus, given 4 banks of 4KB each, a 16KB 4-way cache is possible but an 8KB 4-way cache is not. Similarly, a 4KB cache can only be configured as direct-mapped.

Our research modifies the cache architecture proposed by Zhang to suit dynamic reconfiguration in CMPs. Therefore, we take a detailed look at this architecture here and present our modifications to this architecture in the next chapter.

The block diagram in Figure 2.1 (adapted from [6]) shows the way concatenation method for a 32-bit word size, configurable 16KB 4-way cache with a 32 byte line size. The cache tag is 20-bits wide. The index can vary in size from 9-bits (512 sets) to 7-bits (128 sets). The cache tag size is kept constant to avoid additional logic. Checking two extra bits does not entail almost any overhead in cache energy. reg_0 and reg_1 are two 1-bit registers which along with bits a_{13} and a_{12} are fed into the configuration circuit. The outputs C_0 – C_3 are used to enable / disable each one of the ways. The configuration circuit consists of a few NAND gates and inverters in order to be able to generate distinct outputs for enabling / disabling specific ways.

Line concatenation is used to change the number of blocks to read from the backing store. The physical line size remains the same, whereas the number of blocks read per access

of the backing store is changed by writing to a configuration register. We do not employ line concatenation in our research. The reasons for our choice are explained in chapter 3.

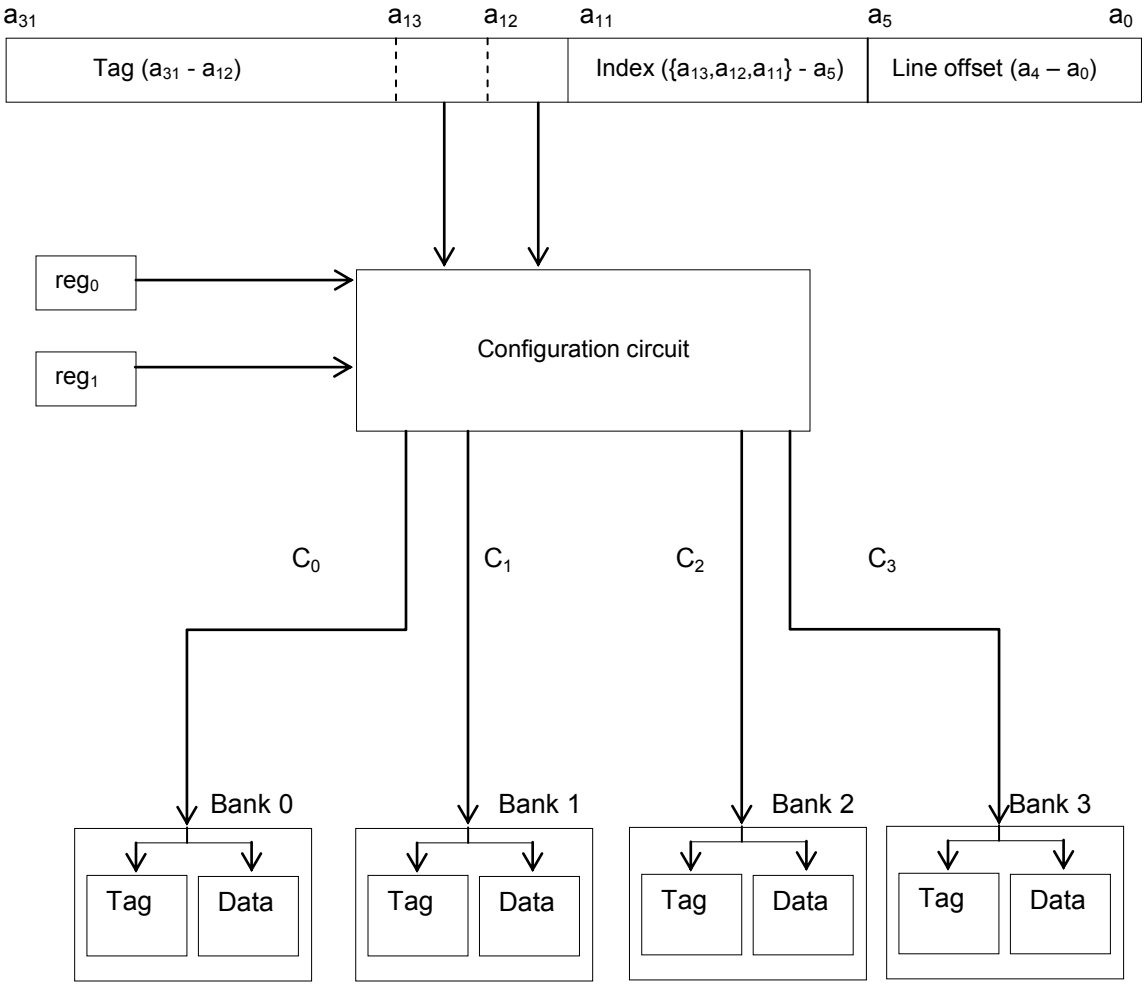

Figure 2.1 Architecture of a configurable cache

It is important that caches are tuned to their optimal configurations so that maximum energy savings can be achieved. Both static and dynamic methods for tuning have been proposed.

2.5.1 Dynamic reconfiguration with static profiling

Wang et al. [15] modeled dynamic cache reconfiguration in uniprocessor real-time multitasking systems as a minimum cost path finding problem in an extended complete bipartite graph and provided a dynamic programming solution to the cache reconfiguration problem. Their method requires static profiling of each application under multiple configurations in order to determine the configuration for the application at a given instant that would lead to minimum energy consumption.

Wang et al. [20] gave an algorithm for L1 cache reconfiguration and L2 cache partitioning in a real-time multicore system which uses static profiling information for every application under each L1 cache configuration with every possible L2 partition, where partitioning is done at the granularity of ways.

Static profiling of each and every task is unfeasible due to long profiling times. For multicore processors, profiling time per application can run into days. Only if the subset of tasks to be run is very limited or if static profiling is a must to determine application behavior, as is the case for a real-time system, can these algorithms prove useful.

2.5.2 Self-tuning configurable caches

A self-tuning cache is a cache architecture which changes its cache configuration dynamically to adapt to the applications running on it. This requires some hardware support so that the applications running can be monitored for their runtime characteristics. It is important that the energy overhead of the additional hardware be low as compared to the energy savings it achieves in order for the design to be useful. While the energy savings for a self-tuning cache may not be as much as those achieved with a method employing static profiling, it is much more widely applicable and practical.

Zhang et al. [21] introduced a self-tuning mechanism for the configurable cache design that they had proposed earlier. They propose the use of on-chip hardware to implement their heuristic that searches for energy optimal configurations by examining different configurations in

an order which minimizes cache flushing. The number of hits, misses and miss rate per configuration are fed into a finite state machine which does energy calculations using Eq. 1.4 and Eq. 1.5 to determine the lowest energy configuration. An additional 3% overhead in chip area and a 40% savings in energy on average are reported.

Gordon-Ross et al. [22] introduced a configurable cache tuner which calculates and examines the changes in energy after variable intervals, which it adjusts over time to match the phase interval. The tuner can therefore adapt to phase changes. An average energy savings of 29% for the instruction cache are reported. The tuner itself is reported to have an energy overhead of about 7% due to the tuning process. The efficiency of the tuning process depends upon the phase change characteristics of the application being executed.

CHAPTER 3

PROPOSED WORK

3.1 Motivation

This thesis proposes a self-tuning cache architecture geared towards chip multiprocessors with online monitoring of statistics in hardware. Before we go into the details of our proposed architecture, it is important to see why existing strategies are unsuitable for CMPs.

3.1.1 Limitations of existing self-tuning schemes

We believe that existing self-tuning strategies [21] [22] are not suitable for multicore systems because of the following reasons:

Firstly, exact energy calculations can be expensive, and involve floating point calculations which themselves consume quite a bit of overall energy if the tuning process is continued over the application's lifetime rather than just once as is evident in the results of Gordon-Ross et al.'s heuristic [22]. For a 4-core CMP with private I-cache and D-cache, there are a total of 8 reconfigurable caches. A per-cache tuner with a high energy overhead is therefore, not practicable.

Secondly, chip multiprocessors involve a shared secondary L2 cache whose energy calculations also need to be factored in the overall energy calculations. This requires per-core statistics of L2 usage or otherwise the cache tuning problem needs to be addressed as a whole rather than on a per-core basis.

Lastly, switching to a larger cache size in a write-back L1 cache is not trivial and may involve a dirty write-back and invalidation cycle which does not fit well with Zhang et al.'s heuristic [21]. We discuss this problem later in this chapter.

3.1.2 Limitations of schemes involving static profiling

The drawbacks of static profiling are not limited to just the longer profiling times. A dynamic approach can address several other shortcomings of a static approach for our purposes.

Firstly, a major advantage of a dynamic approach over a static approach is the responsiveness to context switches. Whenever there is a context switch, the newly scheduled application on the core may have cache requirements which vary widely from that of the previously scheduled application. This may also mean differing cache requirements for each phase. With a static approach, there is no way of identifying a change in requirements unless the entire schedule is known beforehand.

Secondly, the execution path of a program could vary depending on the supplied input. With static profiling, it is not feasible to evaluate all possible input sets for many programs given that a simulating a complete program for a single input set for all possible cache configurations can take days.

Lastly, since the system in consideration is a chip multiprocessor with a shared L2 cache, the variations in the behavior of an application executing on some other core may cause a change in the cache requirements for a given application. For instance, assume that inclusion is maintained between L1 and L2 caches. Therefore, if the phase of a program executing on another core changes [23] or a program with very high cache demand is scheduled on another core, it may occupy a large amount of L2 cache and might possibly evict a number of blocks of our application in L2. Since we have inclusive L1 and L2 caches, this would also evict the corresponding blocks in the L1 cache of our application and thus, increase its miss rate. A static profiling approach has no way of detecting such changes unless all possible combinations are known and simulated beforehand. The number of combinations to simulate in advance increases exponentially in the number of concurrently executing applications.

Therefore, a new approach is needed to address the cache reconfiguration problem in CMPs.

3.2 Proposed self-tuning cache architecture

A tuning heuristic is executed whenever a miss occurs after every fixed *tuning_interval*. The heuristic determines if a more energy optimal configuration is possible under the current overall system conditions, without largely impacting system performance, it switches the cache to that configuration. By constraining the switch to be made only when there is a miss, we can hide the latency of writing dirty blocks in a cache flush substantially. Moreover, the latency of a bank wakeup, if there is any, is completely hidden.

It must be noted that *tuning_interval* can also be made variable. A heuristic may determine a phase change and overwrite the value of the *tuning_interval* stored in the register. However, we only consider a fixed tuning interval in our current research and leave variable *tuning_interval* as future work.

Unlike previous research, most of which is concerned with uniprocessor system, we do not consider a variable block size. We assume a modified 16KB 4-way 32B line size configurable cache with 4KB banks for the L1 I-cache and D-cache per core for all examples presented here on. The L2 cache is shared and is a regular unified 512KB 8-way 64B line size cache on a 4-core CMP. Cache coherence is enforced through the MOESI protocol [24]. This results in the following 6 valid configuration options for each L1 cache:

{16KB 4-way, 16KB 2-way, 16KB 1-way, 8KB 2-way, 8KB 1-way, 4KB 1-way}

3.2.1 Development of tuning heuristic

As explained by Zhang et al. [21], it is not energy efficient to blindly switch between arbitrary cache configurations. Way-shutdown involves cache flushing. So, it is important to minimize the number of switches from a larger cache size to a smaller size or while reducing associativity as both are achieved by way shutdown.

Another important consideration is that increasing number of sets for a write-back L1 cache is not trivial. Unlike the self-tuning heuristic of Zhang et al. [21] which was concerned with uniprocessor systems, increasing set size in a L1 write-back cache involves writing back dirty

data and invalidating the active part of the cache. To see why, consider an L1 I-cache currently configured as 4KB 1-way which needs to be switched to an 8KB 1-way configuration. Therefore,

Old index bits = 7 Old number of sets = 128

New index bits = 8 New number of sets = 256

Assume that the block at address 0xFFFFFAA0 is currently in the cache and is dirty. This would map to set 85 in the old configuration.

Now, if we did not write-back dirty data in the L1 cache while switching and address 0xFFFFFAA0 is requested, this would map to set 213 in the new configuration and result in a miss (data is lost on way shutdown). This miss will fetch data from the backing store. However, the backing store does not have the current copy since the updated data for that address resides in set 85 which was never written back!

We would like to point out that even an L1 I-cache needs to go through a similar cycle in order to handle self-modifying code. We also need to invalidate the cache line in addition to writing back to prevent similar problems that might arise on switching back to the old configuration. As can be observed, there is a lot of overhead, both in performance and energy on increasing set size. We therefore, keep such switches to a minimum. It might be argued that a write-through cache be employed, however, the energy overhead of a write-through cache is far greater and this overhead will be encountered throughout the execution unlike switching to higher set size which would occur rarely, even if it does.

According to Zhang et al. [21], increasing associativity is preferable to reducing associativity as we need not write back dirty data. However, as discussed above, this holds true only if set size remains the same during the increase.

Keeping these considerations in mind, we arrived at the state machine shown in Figure 3.1 for a configurable 16KB 4-way 32B L1 cache with 4KB banks. Our tuning heuristic tries to switch states according to this state machine in the most energy optimal way.

The current system conditions are determined by taking into account the miss rate observed over previous intervals and bounding it to within a threshold of the miss rate during a previous state (configuration). This is discussed in detail later.

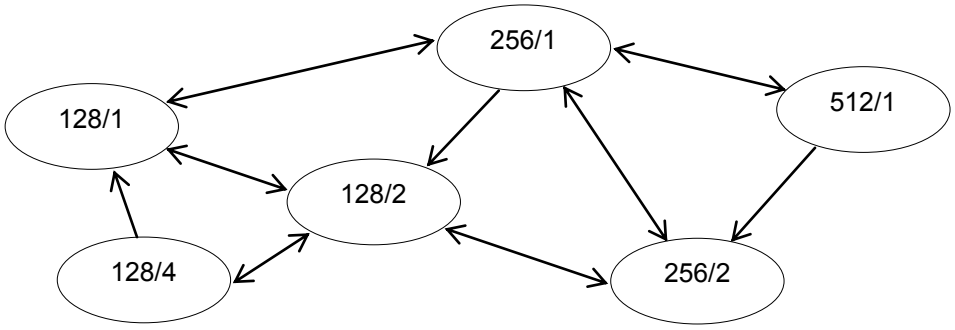

Figure 3.1 State machine for the proposed configurable cache. The numbers inside each circle are in the form: set size / associativity.

3.2.2 Not considering a variable line size

As mentioned previously, we do not consider a variable block size in our configurable cache architecture. This is due to the difficulty in assessing the impact of variable block sizes without switching to that configuration and also the difficulty in isolating the per-core impact even if we examine each possible configuration. Also, a given CMP design might require enforcing cache inclusion or exclusion. While we do not enforce inclusion or exclusion for the simulated system, maintaining inclusion in a chip multiprocessor can become very difficult if we keep the L1 block size much smaller than L2. Moreover, exclusion requires L1 and L2 block sizes to be equal [9].

According to Baer and Wang [11], in order to enforce inclusion with n L1 caches sharing an L2 cache, the associativity A_2 of an L2 cache with block size B_2 and associativity A_1 of an L1 cache with block size B_1 are related as:

$$A_2 \geq n \times A_1 \times (B_2 / B_1)$$

For a 2-core CMP, n=4. Using $B_2 = 64$, $A_1=4$, we get: $A_2 \geq 16 \times (64 / B_1)$

If $B_1 = 64$, $A_2 = 16$ which is reasonable for an L2 cache.

Making B_1 less than 64 would require a highly associative L2 which would have large power consumption. In addition, such a large associativity might be not beneficial.

The reason one might choose to enforce inclusion is that one need not maintain a separate copy of tag directories of each of L1 I-cache and D-cache to handle coherence request from I/O devices or other external devices [25] [9]. The energy savings of not maintaining a separate copy of tag directories would far outweigh any energy savings due to a different block size. This makes an inclusive design highly favorable from a low energy design standpoint.

3.2.3 Hardware support

We modify the configurable cache architecture of Zhang [6] for use in a CMP by adding the following additional hardware:

3.2.3.1 Set mapping identifier (SMI) counters

We introduce Set mapping identifier (SMI) counters as a means of obtaining the count of the number of sets that map to a given interval. The number of SMI counters is given as, N_{SMI} = $\log_2$ (max associativity) + 1. For example, for a 16KB 4-way configurable cache, N_{SMI} = $\log_2$ 4 + 1 = 3 and the intervals are [0, 127], [0, 255] and [0, 511]. The counters can be easily updated while the set number is decoded as shown in Figure 3.2 below. Note that these counters are updated irrespective of whether a given address hits in the cache or not. The width of these counters is *ceil* ($\log_2$ *tuning_interval*) where *ceil* is the mathematical ceiling function.

The counters SMI_1, SMI_2, SMI_3 in the figure represent the number of references made to the intervals [0, 127], [0, 255] and [0, 511] respectively.

3.2.3.2 Way counters

Way counters [26] [27] [28] count the number of hits to each position from MRU to LRU plus one counter for those accesses that do not hit anywhere (i.e., misses). The number of way counters can be calculated as, N_{WC} = (max associativity) + 1.The width of each of these counters is given by *ceil* ($\log_2$ *tuning_interval*) where *ceil* is the mathematical ceiling function. These counters are used to estimate the loss in switching from a 4-way to 2-way or from 2-way

to 1-way configuration. Except the last counter (for misses), all these counters are cumulative counters like SMI counters. Their implementation involves logic similar to SMI counters and access to LRU information. They are also very simple to implement as has been demonstrated in previous works.

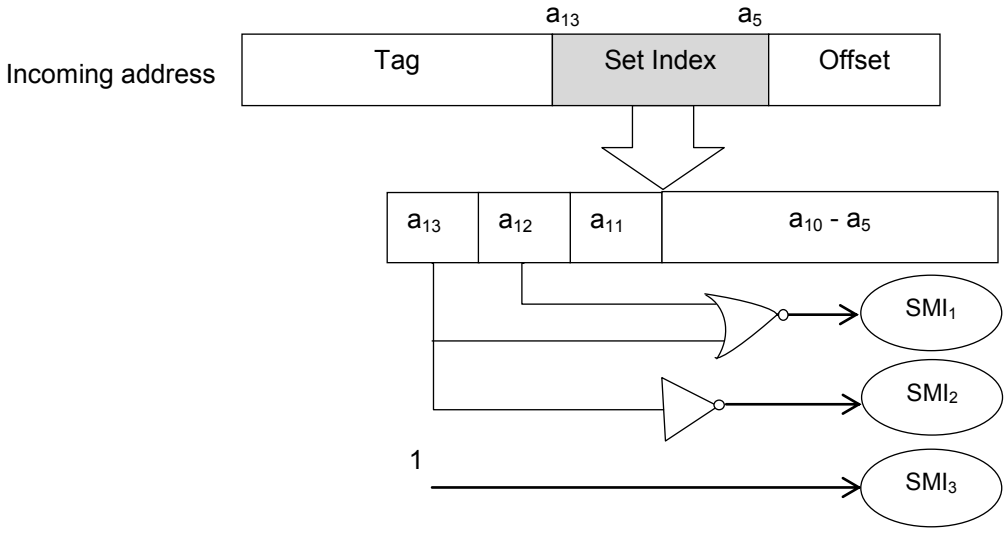

Figure 3.2 Set mapping identifier (SMI) counters for a 16KB 4-way configurable cache

3.2.3.3 Miscellaneous hardware

The following hardware assists in performing the reconfiguration process:

1) A counter for the number of accesses to the cache since the last switch (ALS).

2) A 1-bit register (PFS) to indicate the core to prepare for switch.

3) A register (OMR) for storing the old miss rate, i.e. miss rate calculated during the last tuning interval that resulted in a switch.

4) Two registers for storing the new number of sets (NS) and the new associativity (NA) respectively. The contents of these registers are valid only when PFS = 1. These two registers store the new number of sets and associativity until the actual switch takes place.

3.3 Implementation issues

3.3.1 Stalling a microprocessor core

In case of pipelined superscalar processors, a switch cannot occur as soon as a core finds out if it should switch. This is due to the reason that there might be a pending read/write request to a given set/way which might not be valid after the switch is completed. To overcome this problem, the new associativity and new set size are stored in the NS and NA registers described in the previous section and the PFS bit is set but the switch is not made immediately. On seeing that the PFS bit is set, the microprocessor core stalls on the set decode / operand fetch pipeline stage for any new instruction. When it asserts that there are no pending reads / writes, the switch takes place on the next miss and the PFS bit is reset. Note that other cores need not be stalled unless there is an upward invalidation request from the L2 cache to another sharer while modifying a directory entry during a cache flush.

3.3.2 Maintaining counter values

The set mapping identifiers (SMI) and the way counters need to be reset over time. Whenever a switch is made, we reset them to zero. However, we need not set them to zero every time we don't switch but instead only halve their values so that we keep some history of the previous tuning intervals to aid us in our switching decisions. Switching only on the basis of the last tuning interval may result in abruptly switching among configurations. However, we need to set the counters to zero after some time so that we only use the recent history while making the switching decisions. For our experiments, we reset the counters every 5 tuning intervals.

3.4 Tuning heuristic

The following tuning heuristic is executed every tuning interval to determine if a configuration switch should occur. It simply implements the state machine described earlier.

Cache tuning heuristic:

curr_assoc is the current associativity
half_assoc is half of the maximum associativity
curr_nsets is the current number of sets
hits is Way_counter[$N_{WC} - 1$]
misses is Way_counter[N_{WC}]
accesses_to_min_sets is SMI_1
accesses_to_half_sets is SMI_2
accesses_to_all_sets is SMI_3

begin

current_miss_rate = *misses* / (*hits* + *misses*);
switch_loss_half = Way_counter[*curr_assoc*] - Way_counter[*half_assoc*];
switch_loss_min = Way_counter[*half_assoc*] - Way_counter[1];

if (*curr_assoc* == max associativity)
{
    if (*switch_loss_half* / *hits* < *threshold_1*)
    {
        if (*switch_loss_min* / *switch_loss_half* < *threshold_2*)
            NA = min associativity, PFS = 1, update OMR
        else
            NA = half associativity, PFS = 1, update OMR
    }
}
else if (*curr_assoc* == half associativity)
{
    if (*curr_nsets* == min size)
    {
        if (*current_miss_rate* < OMR) //Try switching to lower energy configuration
        {
            if (*switch_loss_min* < *threshold_3*)
                NA = min associativity, PFS = 1, update OMR
        }
        else if (*current_miss_rate* – OMR < *threshold_4*)
            NS = half size, PFS = 1, update OMR
        else
            NA = max associativity, PFS = 1, update OMR
    }
    else // half size
    {
        if (*accesses_to_min_sets* / *accesses_to_all_sets* > *threshold_5*)
            NS = min size, PFS = 1, update OMR
        else if (switch_loss_half < threshold_6)
            NA = min associativity, PFS = 1, update OMR
    }
}
else // min associativity
{

```
        if ( curr_nsets == min size )
        {
                if ( current_miss_rate – OMR < threshold_7)
                        return;
                else if ( accesses_to_min_sets / accesses_to_all_sets > threshold_8 )
                        NA = half associativity, PFS = 1, update OMR
                else
                        NS = half size, PFS = 1, update OMR
        }
        else if ( curr_nsets == half size)
        {
                if ( current_miss_rate – OMR < threshold_9 )
                {                                               //try switching to a lower energy state
                        if ( accesses_to_min_sets / accesses_to_all_sets > threshold_10 )
                                NS = min size, PFS = 1, update OMR
                }
                else if ( current_miss_rate – OMR > threshold_11 )
                {
                        if ( accesses_to_min_sets / accesses_to_all_sets > threshold_12 )
                                                        //try a higher associativity
                                NA = half associativity, PFS = 1, update OMR
                }
                else if ( current_miss_rate – OMR > threshold_13 )
                {
                        if ( accesses_to_half_sets / accesses_to_all_sets < threshold_14 )
                                NS = max size, NA = min associativity, PTS = 1
                        else
                                NA = half associativity, PFS = 1, update OMR
                }
        }
        else    // max size
        {
                if ( current_miss_rate – OMR < threshold_15 )
                {
                        if ( accesses_to_half_sets / accesses_to_all_sets > threshold_16 )
                                NS = half size, PFS = 1, update OMR
                }
                else
                        NS = half size, NA = half associativity, PFS = 1, update OMR
        }
}

end
```

Figure 3.3 Cache tuning heuristic

The threshold parameters are derived taking into account the current configuration and the overhead in performance and energy of switching to a new configuration. We obtained the

energy consumption figures for each configuration using CACTI [7] and then estimated the threshold parameters. For instance, we made the following threshold choices for our experiments with a 16KB 4-way configurable cache:

(1) *threshold_1* = *threshold_2* = 15% : Since we are in the maximum energy consuming configuration, we should switch to a lower configuration as better options might be available.

(2) *threshold_3* = *threshold_6* = 5% : A hit rate, 95% of the current hit rate seems to be a good tradeoff for a lower energy configuration. Similar choice for *threshold _5* = 90%.

(3) *threshold_4* = 4% : The system should be experiencing severe performance loss to switch to maximum energy configuration. Otherwise, other better performing configurations are available.

Other values were also arrived at using a similar logic. We ended up with the following values for the rest of the thresholds: *threshold_7* = 3 %, *threshold_8* = *threshold_12* = 70% or ~ 2/3, *threshold_9* = *threshold_11* = 2.5 %, *threshold_10* = *threshold_16* = 90%, *threshold_13* = *threshold_15* = 2%, *threshold_14* = 75%.

It must however, be noted that a slight variation in these parameters gives similar results and therefore, these parameters should not be taken as absolute values. It is important to choose an appropriate tradeoff for the cache configurations in hand so that the performance and energy saving requirements are met.

We used a tuning interval of 100,000 cache accesses for our experiments.

CHAPTER 4

EXPERIMENTS

4.1 Simulation setup

We extensively modified the Multi2Sim [29] simulator version 3.2.1 to implement our proposed heuristic and cache architecture on a 4-core CMP. The memory hierarchy was defined as in Table 4.1 below:

Table 4.1 Memory hierarchy of the simulated system

I - L1 and D - L1 caches	Reconfigurable 16KB 4-way cache 32 B block size 2 cycle access latency
L1 - L2 interconnect	64-bit wide bus topology
L2 cache	512 KB 8-way cache 64 B block size Shared, Unified, inclusion is not enforced 10 cycle access latency
L2 - Main memory interconnect	64-bit wide bus topology
Main memory	125 cycle access latency

The processor simulated was a 4-core CMP running at 1 GHz clock frequency. We used the Instructions per cycle (IPC) and the total number of cycles for benchmark execution as a metric for evaluating relative performance. We compare results with a system having a 16KB 4-way L1 cache as well as another system having a 16KB 2-way L1 cache. All other parameters were kept identical for all the configurations tested.

We made energy calculations using Eq. 1.4 and Eq. 1.5. Dynamic energy and static power values were obtained from CACTI [7] for a 45nm node. Note that we need not count L2 dynamic energy separately as that calculation was made while calculating miss energy for L1.We take miss energy where the block is to be fetched from main memory to be 45nJ (for a

1.8V main memory module). Energy loss due to microprocessor stall was taken as 15nJ resulting in an overall main memory access energy of 60nJ per access. It should be noted that this penalty is several hundred times more than the average L1 access energy.

4.2 Benchmarks

Both single as well as multithreaded benchmarks were chosen from the PARSEC 2.1 [30] {*x264*, *bodytrack*, *facesim*, *fluidanimate*, *vips*, *ferret*, *blackscholes*, *canneal*}, SPLASH-2 [31] {*fft*, *radix*, *lu*, *raytrace*} and MiBench [32] {*gsm*, *sha*, *crc32*, *adpcm*, *jpeg*, *dijkstra*, *patricia*, *ispell*} benchmark suites. We ran a total of 15 different benchmark combinations for a total of 200 million instructions each. Table 4.2 gives further details about the benchmark combinations that were executed.

Table 4.2 Benchmarks executed

S.No.		Core 1	Core 2	Core 3	Core 4
1	Single threaded application mix	x264	facesim	sha	raytrace
2		gsm	jpeg	blackscholes	bodytrack
3		bodytrack	ferret	crc32	fft
4		dijkstra	sha	patricia	fluidanimate
5		ispell	vips	blackscholes	adpcm
6		vips	ferret	lu	adpcm
7		canneal	x264	fft	blackscholes
8		gsm	dijkstra	ispell	sha
9	Multithreaded	fft (4 threads)			
10		raytrace (4 threads)			
11		radix (4 threads)			
12		lu (4 threads)			
13		blackscholes (2 threads), vips (2 threads)			
14		ferret (3 threads), patricia (1 thread)			
15		bodytrack (4 threads)			

Benchmarks 10,13,14,15 were fast forwarded 200 million instructions each to get to the multithreaded phase. All other combinations were fast forwarded 10 million instructions each. The core numbers are the initial cores where each application was scheduled to run.

4.3 Results

For each benchmark, we report the number of cycles, IPC, L1 energy, L2 static energy and overall energy consumption. Also, we present a graphical view of how L1 energy, L2 static energy and overall system energy compare across different the systems examined.

The fields in the results comparison table are explained below:

“Configurable” refers to our proposed 16KB 4-way architecture. 16K4W means a regular 16KB 4-way cache and 16K2W means a regular 16KB 2-way cache.

“% of 4W” refers to how a given metric for a configurable cache compares to a regular 16K4W cache. Similar meaning exists for “% of 2W”.

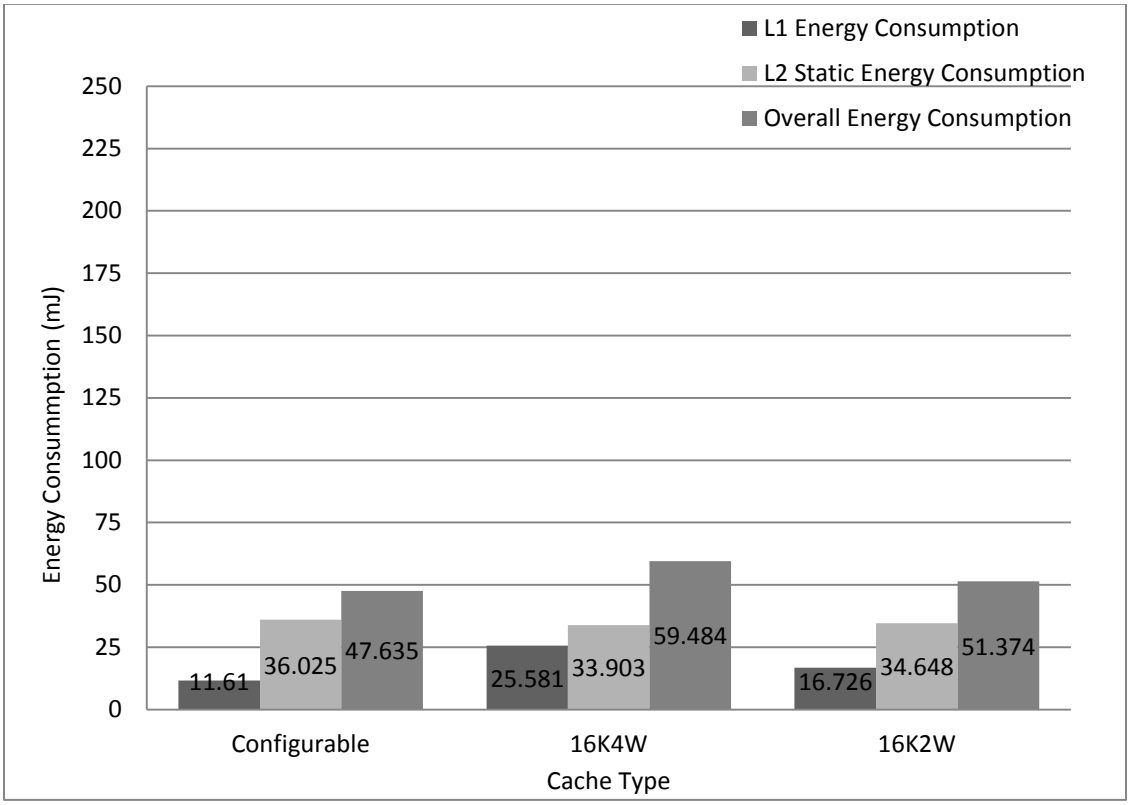

Figure 4.1 Energy statistics for {*x264*, *facesim*, *sha*, *raytrace*}

Table 4.3 Results comparison for {*x264*, *facesim*, *sha*, *raytrace*}

Cache Type	Cycles	IPC	L1 Energy Consumption (mJ)	L2 Static Energy Consumption (mJ)	Overall Energy Consumption (mJ)
Configurable	32770877	6.103	11.61	36.025	47.635
16K 4W	30840548	6.485	25.581	33.903	59.484
16K 2W	31518020	6.346	16.726	34.648	51.374
% of 4W	106.26 %	94.11 %	45.39 %	106.26 %	80.08 %
% of 2W	103.98 %	96.17 %	69.41 %	103.97 %	92.72 %

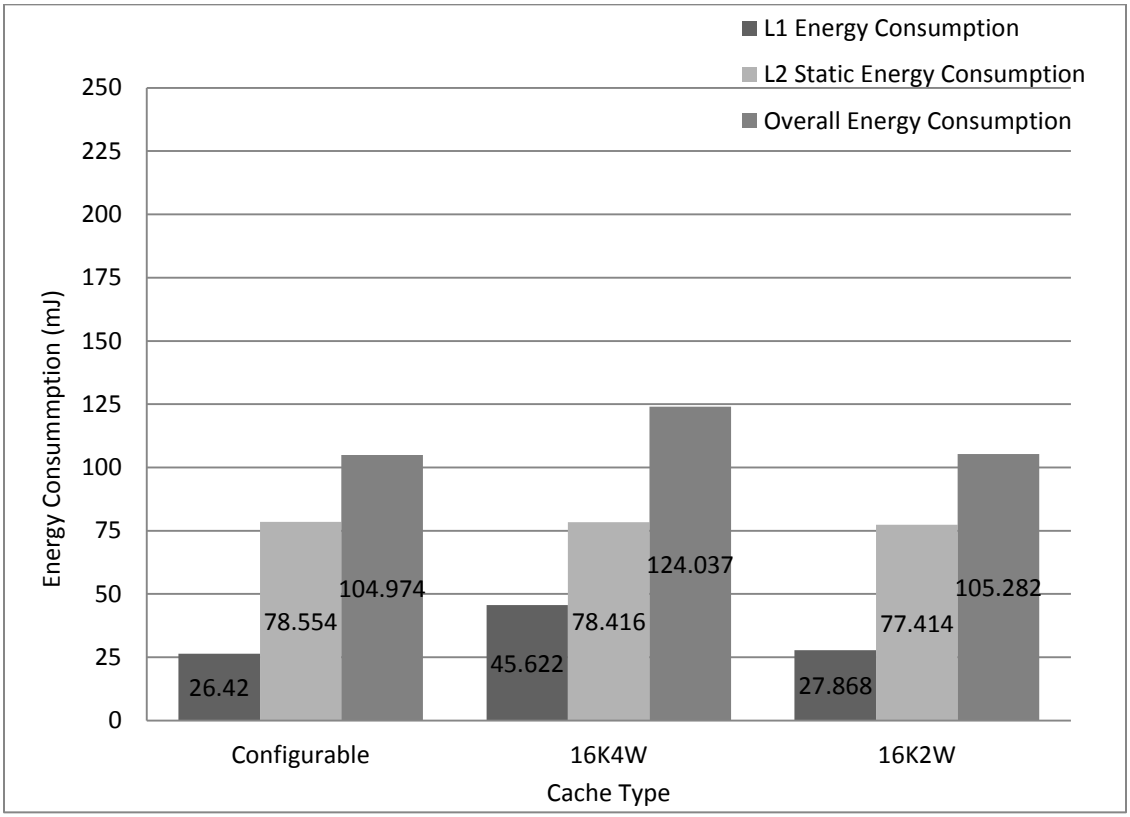

Figure 4.2 Energy statistics for {*gsm*, *jpeg*, *blackscholes*, *bodytrack*}

Table 4.4 Results comparison for {*gsm*, *jpeg*, *blackscholes*, *bodytrack*}

Cache Type	Cycles	IPC	L1 Energy Consumption (mJ)	L2 Static Energy Consumption (mJ)	Overall Energy Consumption (mJ)
Configurable	71458474	2.799	26.42	78.554	104.974
16K 4W	71332593	2.804	45.622	78.416	124.037
16K 2W	70421407	2.84	27.868	77.414	105.282
% of 4W	100.18%	99.82%	57.91%	100.18%	84.63%
% of 2W	101.47%	98.56%	94.80%	101.47%	99.71%

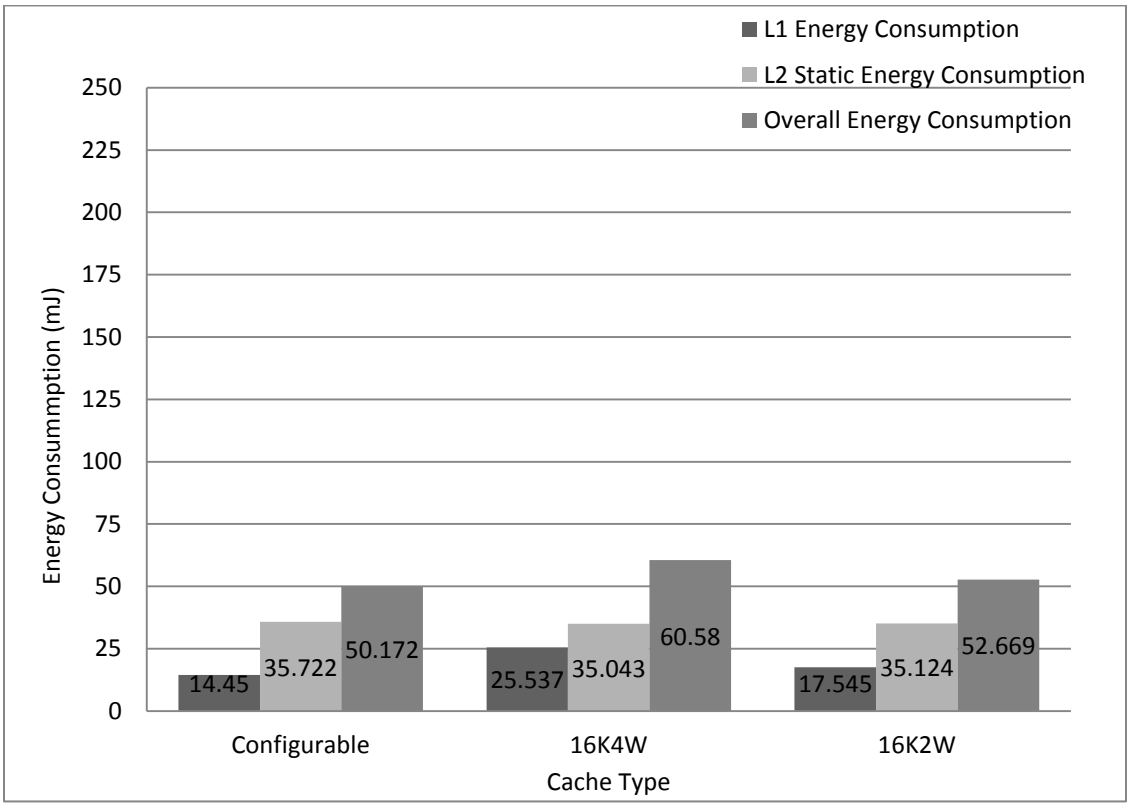

Figure 4.3 Energy statistics for {*bodytrack*, *ferret*, *crc32*, *fft*}

Table 4.5 Results comparison for {*bodytrack*, *ferret*, *crc32*, *fft*}

Cache Type	Cycles	IPC	L1 Energy Consumption (mJ)	L2 Static Energy Consumption (mJ)	Overall Energy Consumption (mJ)
Configurable	32495455	6.155	14.45	35.722	50.172

Table 4.5 - *Continued*

16K 4W	31877723	6.274	25.537	35.043	60.58
16K 2W	31951540	6.259	17.545	35.124	52.669
% of 4W	101.94%	98.10 %	56.58%	101.94%	82.82%
% of 2W	101.70%	98.34 %	82.36%	101.70%	95.26%

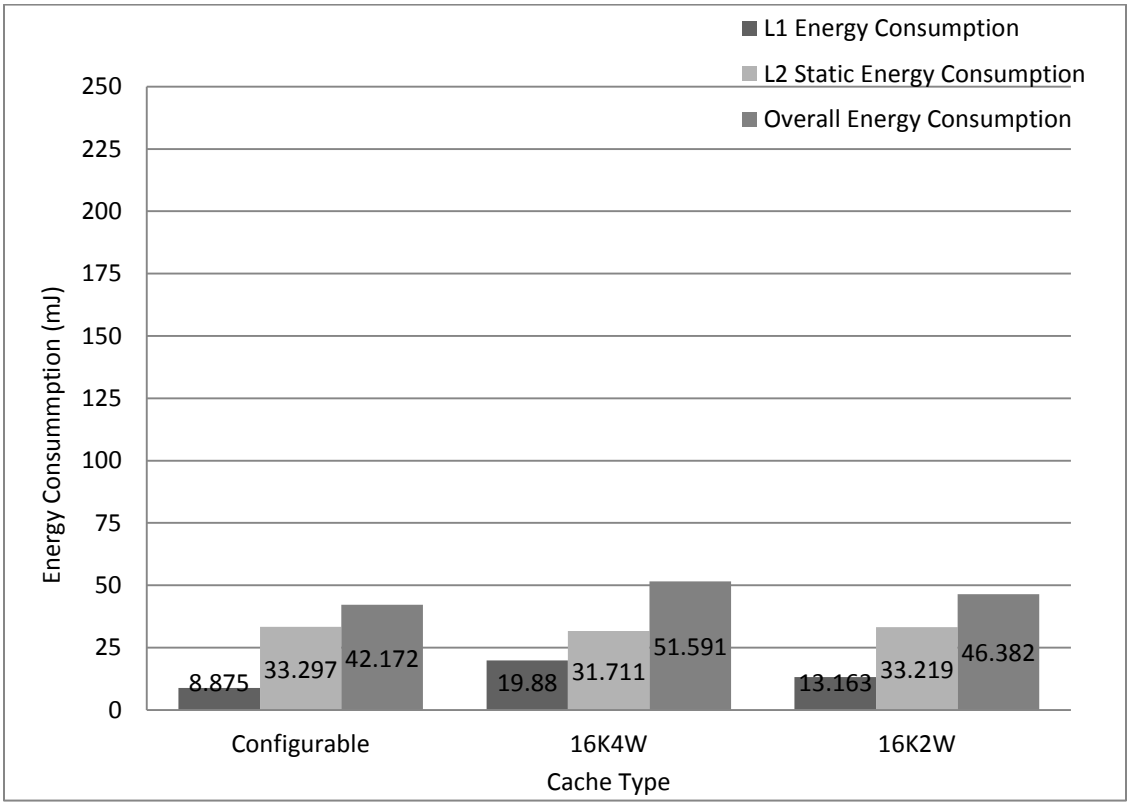

Figure 4.4 Energy statistics for {*dijkstra*, *sha*, *patricia*, *fluidanimate*}

Table 4.6 Results comparison for {*dijkstra*, *sha*, *patricia*, *fluidanimate*}

Cache Type	Cycles	IPC	L1 Energy Consumption (mJ)	L2 Static Energy Consumption (mJ)	Overall Energy Consumption (mJ)
Configurable	30289553	6.603	8.875	33.297	42.172
16K 4W	28846635	6.933	19.88	31.711	51.591
16K 2W	30218386	6.618	13.163	33.219	46.382

Table 4.6 - *Continued*

% of 4W	105.00%	95.24%	44.64%	105.00%	81.74%
% of 2W	100.24%	99.77%	67.42%	100.23%	90.92%

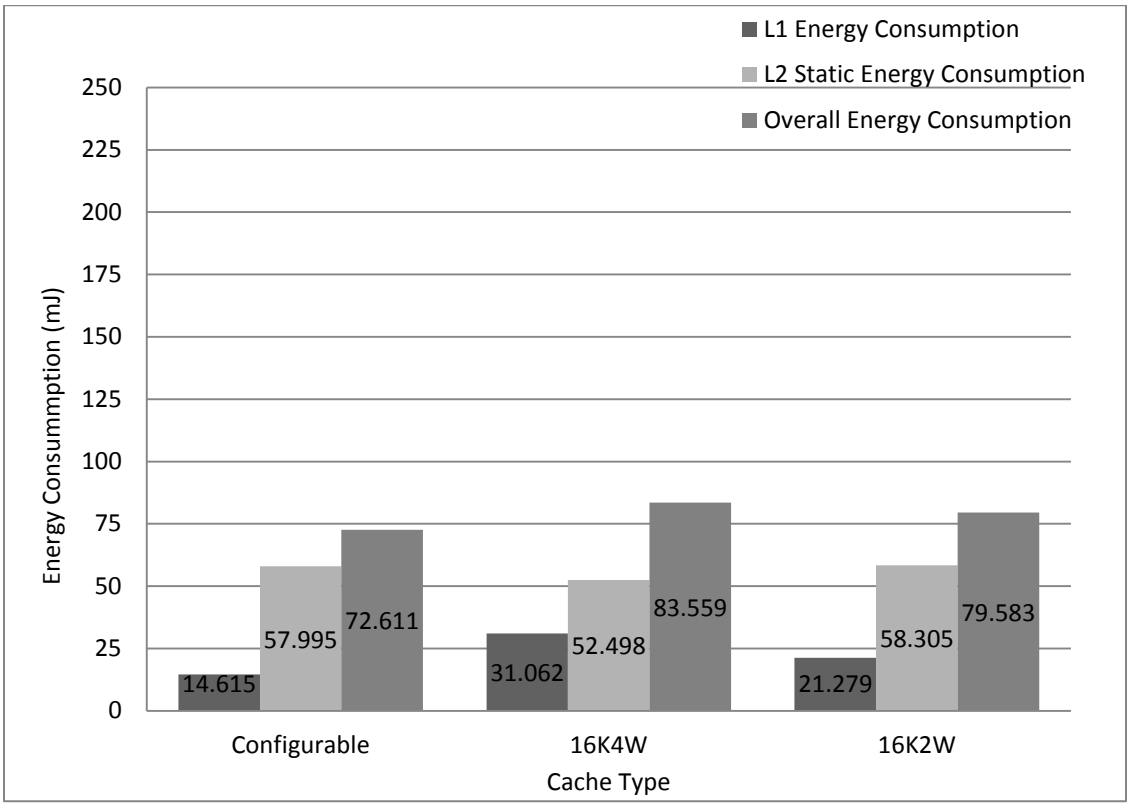

Figure 4.5 Energy statistics for {*ispell*, *vips*, *blackscholes*, *adpcm*}

Table 4.7 Results comparison for {*ispell*, *vips*, *blackscholes*, *adpcm*}

Cache Type	Cycles	IPC	L1 Energy Consumption (mJ)	L2 Static Energy Consumption (mJ)	Overall Energy Consumption (mJ)
Configurable	52756512	3.791	14.615	57.995	72.611
16K 4W	47755556	4.188	31.062	52.498	83.559
16K 2W	53038114	3.771	21.279	58.305	79.583
% of 4W	110.47%	90.52%	47.05%	110.47%	86.90%
% of 2W	99.47%	100.53%	68.68%	99.47%	91.24%

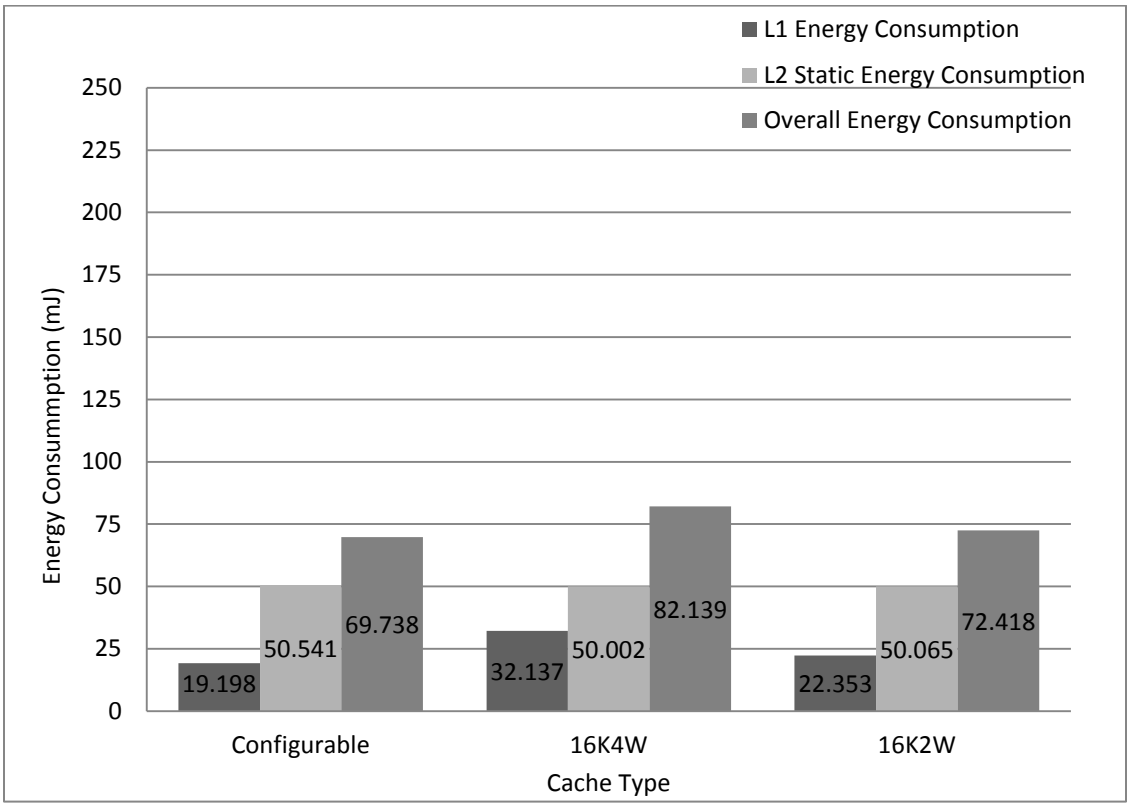

Figure 4.6 Energy statistics for {*vips*, *ferret*, *lu*, *adpcm*}

Table 4.8 Results comparison for {*vips*, *ferret*, *lu*, *adpcm*}

Cache Type	Cycles	IPC	L1 Energy Consumption (mJ)	L2 Static Energy Consumption (mJ)	Overall Energy Consumption (mJ)
Configurable	45975472	4.35	19.198	50.541	69.738
16K 4W	45485395	4.397	32.137	50.002	82.139
16K 2W	45542635	4.391	22.353	50.065	72.418
% of 4W	101.08%	98.93%	59.74%	101.08%	84.90%
% of 2W	100.95%	99.07%	85.89%	100.95%	96.30%

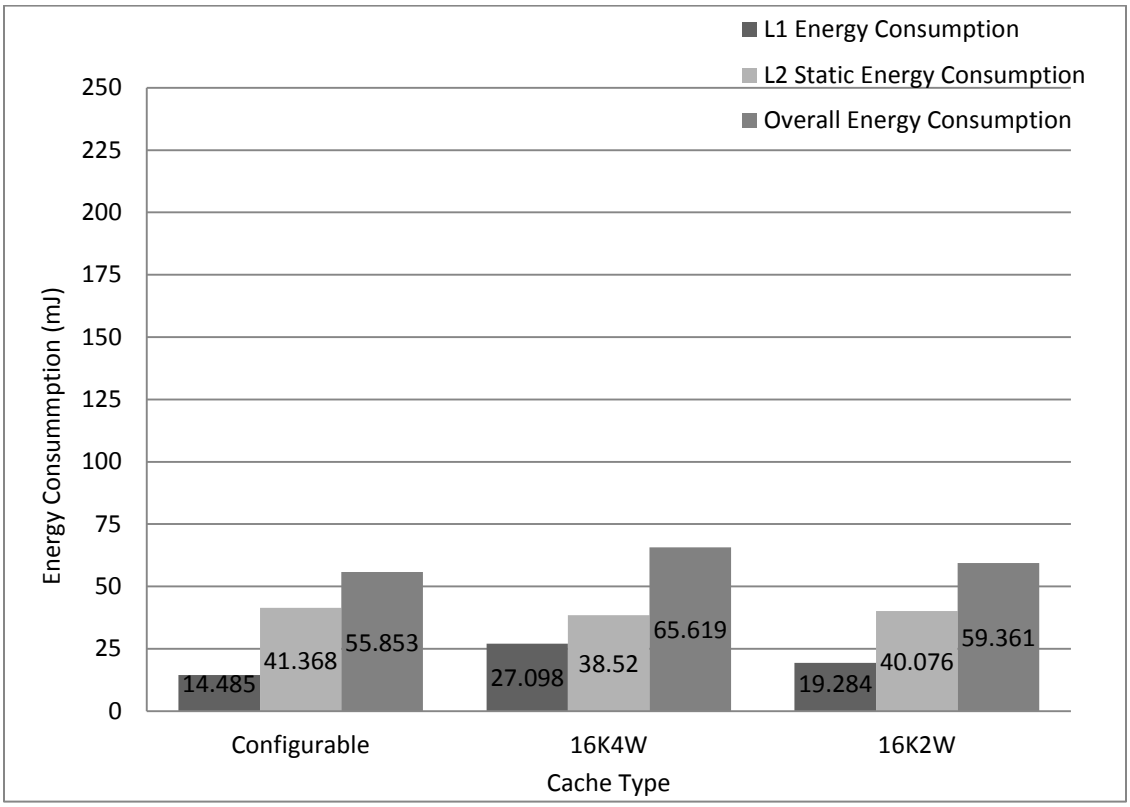

Figure 4.7 Energy statistics for {*canneal*, *x264*, *fft*, *blackscholes*}

Table 4.9 Results comparison for {*canneal*, *x264*, *fft*, *blackscholes*}

Cache Type	Cycles	IPC	L1 Energy Consumption (mJ)	L2 Static Energy Consumption (mJ)	Overall Energy Consumption (mJ)
Configurable	37631112	5.315	14.485	41.368	55.853
16K 4W	35040817	5.708	27.098	38.52	65.619
16K 2W	36456365	5.486	19.284	40.076	59.361
% of 4W	107.39%	93.11%	53.45%	107.39%	85.12%
% of 2W	103.22%	96.88%	75.11%	103.22%	94.09%

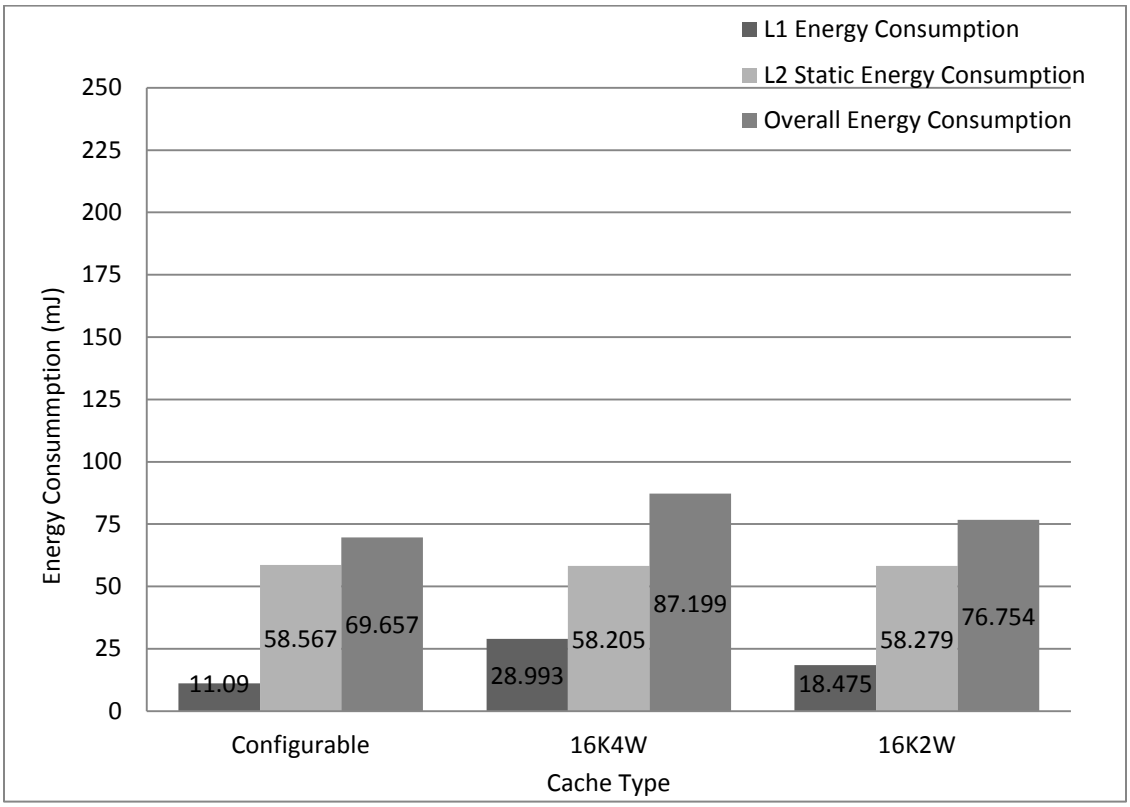

Figure 4.8 Energy statistics for {*gsm*, *dijkstra*, *ispell*, *sha*}

Table 4.10 Results comparison for {*gsm*, *dijkstra*, *ispell*, *sha*}

Cache Type	Cycles	IPC	L1 Energy Consumption (mJ)	L2 Static Energy Consumption (mJ)	Overall Energy Consumption (mJ)
Configurable	53277098	3.754	11.09	58.567	69.657
16K 4W	52947818	3.777	28.993	58.205	87.199
16K 2W	53014289	3.773	18.475	58.279	76.754
% of 4W	100.62%	99.39%	38.25%	100.62%	79.88%
% of 2W	100.50%	99.50%	60.03%	100.49%	90.75%

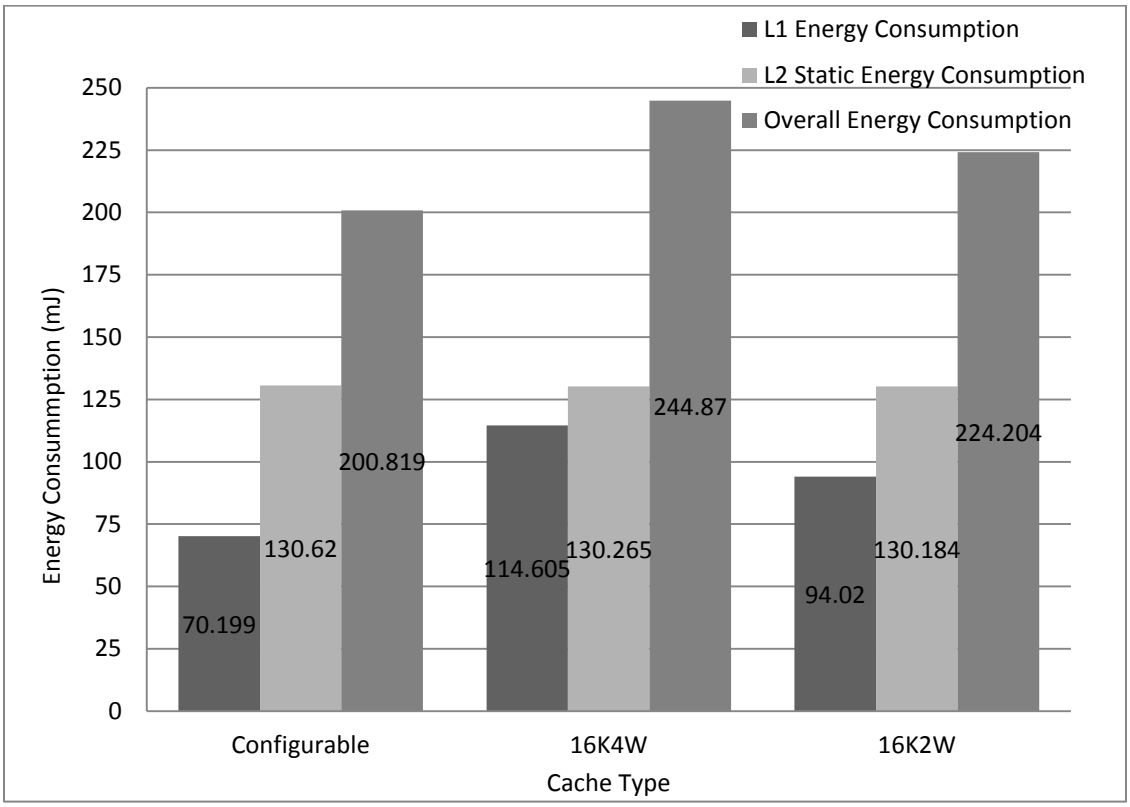

Figure 4.9 Energy statistics for {*fft* (4 threads)}

Table 4.11 Results comparison for {*fft* (4 threads)}

Cache Type	Cycles	IPC	L1 Energy Consumption (mJ)	L2 Static Energy Consumption (mJ)	Overall Energy Consumption (mJ)
Configurable	118821520	1.683	70.199	130.62	200.819
16K 4W	118498119	1.688	114.605	130.265	244.87
16K 2W	118424522	1.689	94.02	130.184	224.204
% of 4W	100.27%	99.70%	61.25%	100.27%	82.01%
% of 2W	100.34%	99.64%	74.66%	100.33%	89.57%

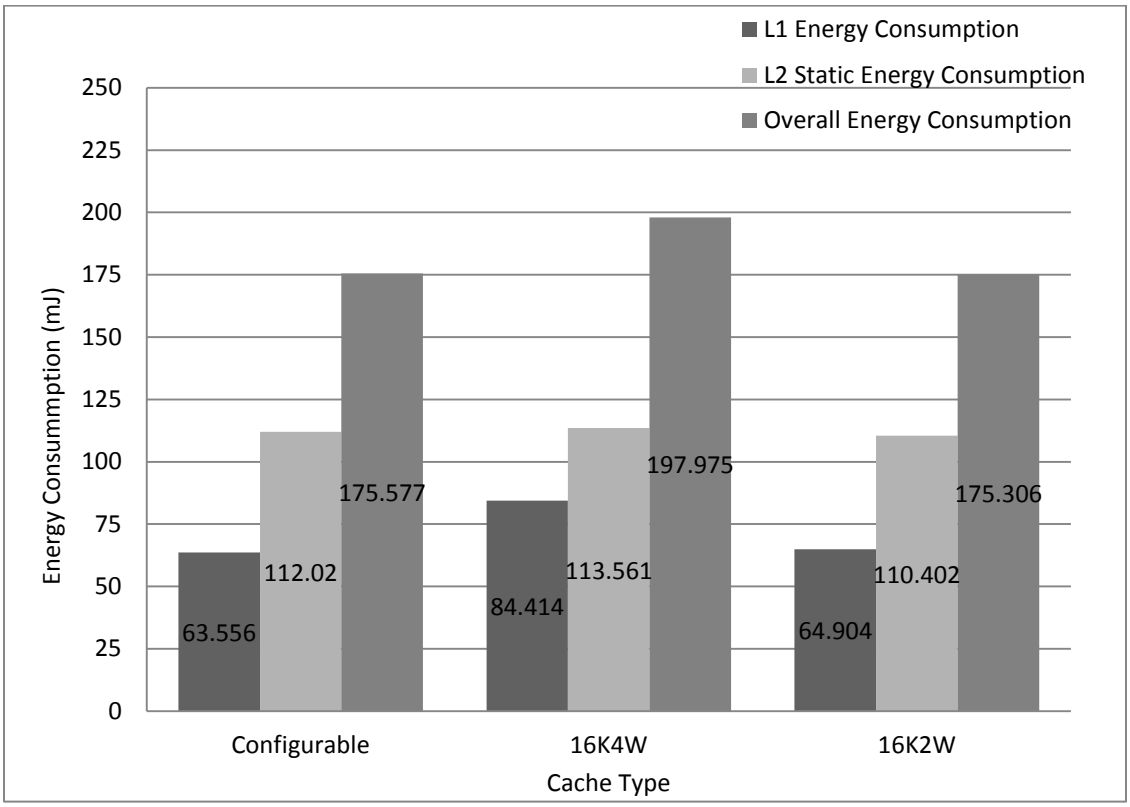

Figure 4.10 Energy statistics for {*raytrace* (4 threads)}

Table 4.12 Results comparison for {*raytrace* (4 threads)}

Cache Type	Cycles	IPC	L1 Energy Consumption (mJ)	L2 Static Energy Consumption (mJ)	Overall Energy Consumption (mJ)
Configurable	101901705	1.963	63.556	112.02	175.577
16K 4W	103302981	1.936	84.414	113.561	197.975
16K 2W	100429625	1.991	64.904	110.402	175.306
% of 4W	98.64%	101.39%	75.29%	98.64%	88.69%
% of 2W	101.47%	98.59%	97.92%	101.47%	100.15%

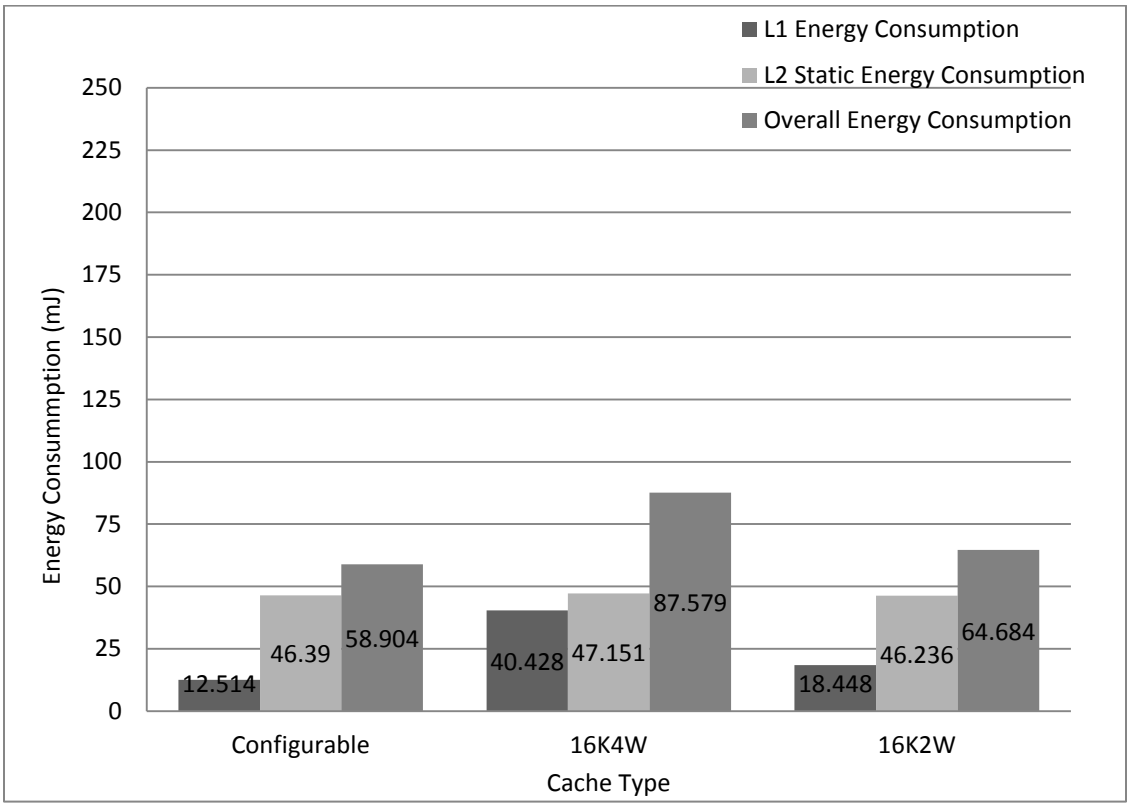

Figure 4.11 Energy statistics for {*radix* (4 threads)}

Table 4.13 Results comparison for {*radix* (4 threads)}

Cache Type	Cycles	IPC	L1 Energy Consumption (mJ)	L2 Static Energy Consumption (mJ)	Overall Energy Consumption (mJ)
Configurable	42199274	4.739	12.514	46.39	58.904
16K 4W	42891841	4.663	40.428	47.151	87.579
16K 2W	42059597	4.755	18.448	46.236	64.684
% of 4W	98.39%	101.63%	30.95%	98.39%	67.26%
% of 2W	100.33%	99.66%	67.83%	100.33%	91.06%

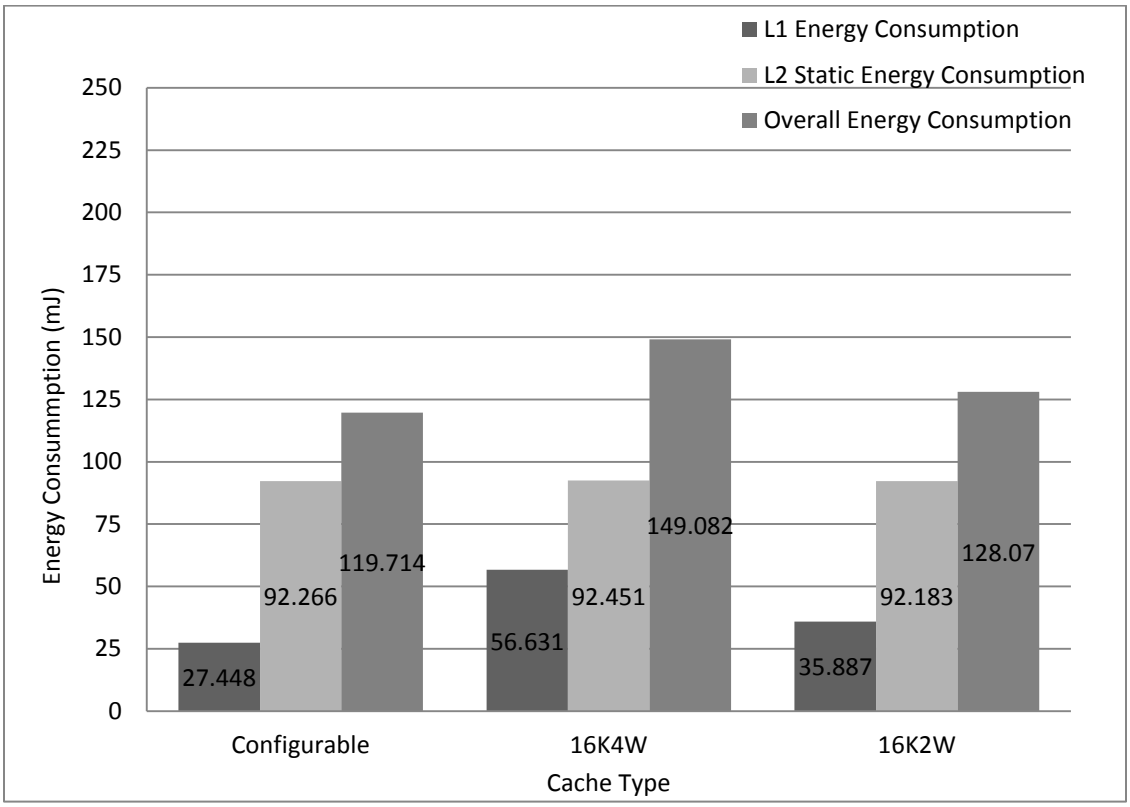

Figure 4.12 Energy statistics for {*lu* (4 threads)}

Table 4.14 Results comparison for {*lu* (4 threads)}

Cache Type	Cycles	IPC	L1 Energy Consumption (mJ)	L2 Static Energy Consumption (mJ)	Overall Energy Consumption (mJ)
Configurable	83932053	2.383	27.448	92.266	119.714
16K 4W	84099769	2.378	56.631	92.451	149.082
16K 2W	83856544	2.385	35.887	92.183	128.07
% of 4W	99.80%	100.21%	48.47%	99.80%	80.30%
% of 2W	100.09%	99.92%	76.48%	100.09%	93.48%

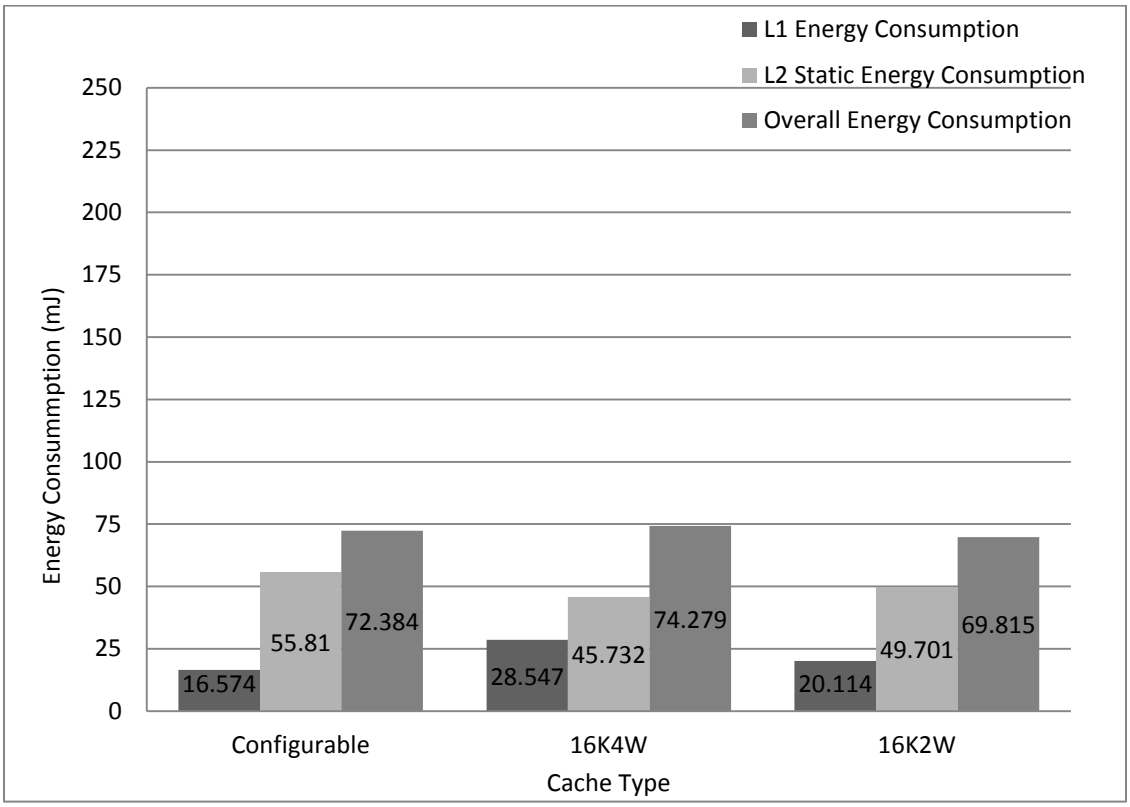

Figure 4.13 Energy statistics for {*blackscholes* (2 threads), *vips* (2 threads)}

Table 4.15 Results comparison for {*blackscholes* (2 threads), *vips* (2 threads)}

Cache Type	Cycles	IPC	L1 Energy Consumption (mJ)	L2 Static Energy Consumption (mJ)	Overall Energy Consumption (mJ)
Configurable	50768546	3.939	16.574	55.81	72.384
16K 4W	41600723	4.808	28.547	45.732	74.279
16K 2W	45211604	4.424	20.114	49.701	69.815
% of 4W	122.04%	81.93%	58.06%	122.04%	97.45%
% of 2W	112.29%	89.04%	82.40%	112.29%	103.68%

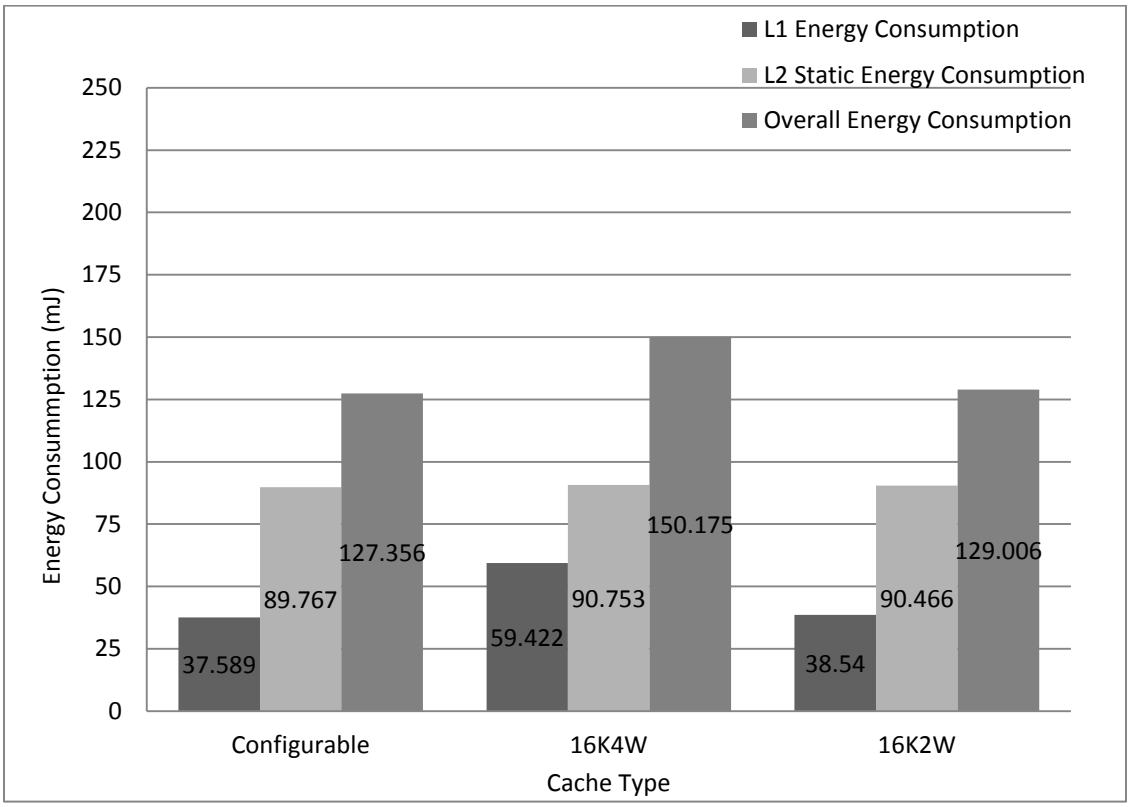

Figure 4.14 Energy statistics for {*ferret* (3 threads), *patricia* (1 thread)}

Table 4.16 Results comparison for {*ferret* (3 threads), *patricia* (1 thread)}

Cache Type	Cycles	IPC	L1 Energy Consumption (mJ)	L2 Static Energy Consumption (mJ)	Overall Energy Consumption (mJ)
Configurable	81658608	2.449	37.589	89.767	127.356
16K 4W	82555635	2.423	59.422	90.753	150.175
16K 2W	82294149	2.43	38.54	90.466	129.006
% of 4W	98.91%	101.07%	63.26%	98.91%	84.81%
% of 2W	99.23%	100.78%	97.53%	99.23%	98.72%

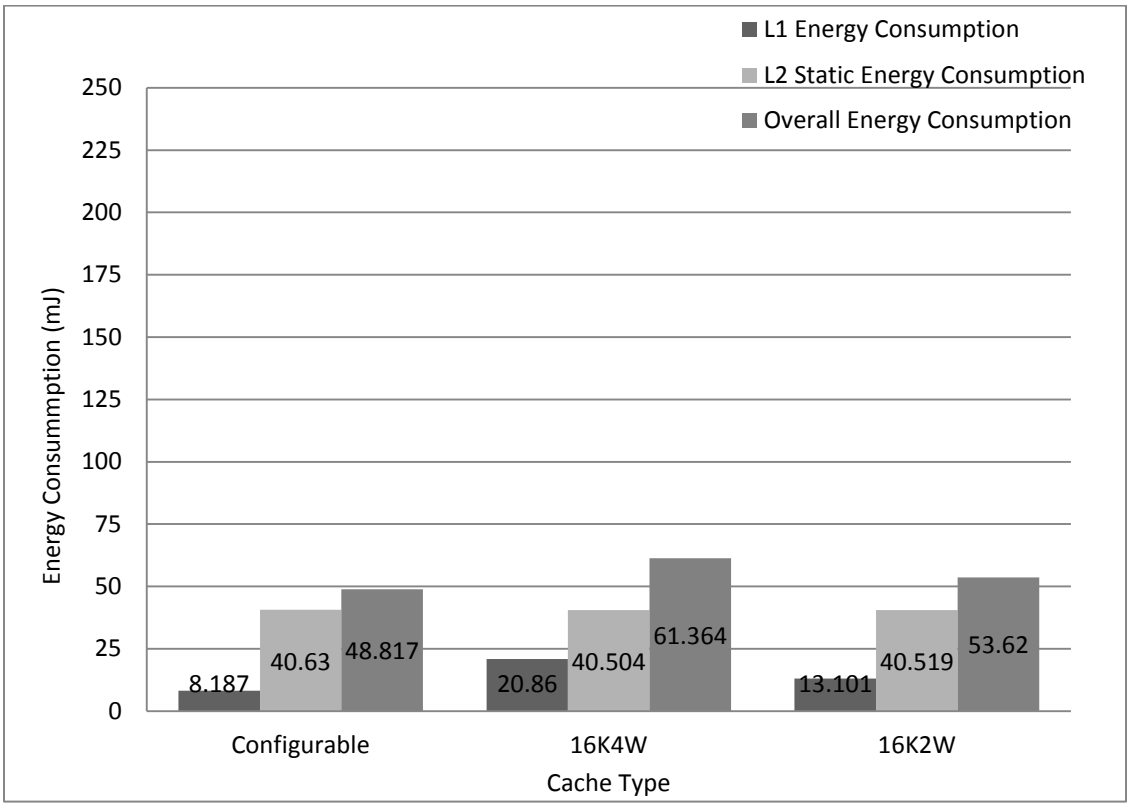

Figure 4.15 Energy statistics for {*bodytrack* (4 threads)}

Table 4.17 Results comparison for {*bodytrack* (4 threads)}

Cache Type	Cycles	IPC	L1 Energy Consumption (mJ)	L2 Static Energy Consumption (mJ)	Overall Energy Consumption (mJ)
Configurable	36959998	5.411	8.187	40.63	48.817
16K 4W	36845663	5.428	20.86	40.504	61.364
16K 2W	36858811	5.426	13.101	40.519	53.62
% of 4W	100.31%	99.69%	39.25%	100.31%	79.55%
% of 2W	100.27%	99.72%	62.49%	100.27%	91.04%

4.4 Analysis of results

Table 4.18 Comparison of overall results

	Cycles	IPC	L1 Energy Consumption	L2 Static Energy Consumption	Overall Energy Consumption
% of 4W	103.42%	96.99%	51.97%	103.42%	83.08%
% of 2W	101.70%	98.41%	77.54%	101.70%	94.58%

From Table 4.18, we can observe that relative to a regular 16KB 4-way L1 cache, on average an energy savings of 16.92 % is observed with a configurable 16KB 4-way L1 cache with only a 3.01% reduction in IPC. Compared to a 16KB 2-way L1 cache, average energy savings of 5.42 % are observed with only 1.59% degradation in the overall IPC. The energy consumption overhead due to the additional monitoring hardware is negligible compared to the overall energy consumption.

For some benchmarks, notably {*blackscholes* (2 threads), *vips* (2 threads)}, the savings in dynamic energy are not worth the corresponding large reduction in performance and the corresponding increase in static energy (see Table 4.15). This behavior occurs since in this case, our chosen tuning interval is long enough that it is unable to detect the change in program phase. A variable tuning interval that adjusts to the program phase would better accommodate such a case.

It is important to point out that savings on L1-cache energy alone are huge – 48.03 % on average relative to a regular 16KB 4-way L1 cache and 22.46 % relative to a 16KB 2-way L1 cache. The increase in L2 static energy due to the extra cycles consumed reduces the overall improvement. If we had considered application of techniques like drowsy caches [18] to the L2 cache, far more overall energy savings would have been noticed.

4.5 Conclusion and future work

The proposed heuristic combined with a configurable cache is successful in achieving substantial savings in energy consumption relative to a regular cache. A variable tuning interval that adjusts according to the program phase could be beneficial in better regulating the corresponding reduction in performance. For our future work, we would like to examine the effect of a variable tuning interval on the performance and energy consumption.

REFERENCES

[1] "Cortex-A9 Processor - ARM," ARM, [Online]. Available: http://www.arm.com/products/processors/cortex-a/cortex-a9.php. [Accessed March 2012].

[2] B. Jacob, S. Ng and D. Wang, Memory Systems: Cache, DRAM, Disk, 1st ed., San Francisco, CA: Morgan Kaufmann Publishers Inc., 2007.

[3] A. Malik, B. Moyer and D. Cermak, "A low power unified cache architecture providing power and performance flexibility," in *Proceedings of the 2000 international symposium on Low power electronics and design*, Rapallo, Italy, 2000.

[4] K. Inoue, T. Ishihara and K. Murakami, "Way-predicting set-associative cache for high performance and low energy consumption," in *Proceedings of the 1999 international symposium on Low power electronics and design*, San Diego, California, 1999.

[5] W. A. Wulf and A. M. Sally, "Hitting the Memory Wall : Implications of the Obvious," *SIGARCH Comput. Archit. News,* vol. 23, no. 1, pp. 20-24, March 1995.

[6] C. Zhang, F. Vahid and W. Najjar, "A highly configurable cache for low energy embedded systems," *ACM Trans. Embed. Comput. Syst.,* vol. 4, no. 2, pp. 363-387, May 2005.

[7] S. Thoziyoor, N. Muralimanohar, J. H. Ahn and N. Jouppi, "CACTI 5.3 (rev 174)," HP Labs, [Online]. Available: http://quid.hpl.hp.com:9081/cacti/.

[8] M. D. Hill, "A Case for Direct-Mapped Caches," *Computer,* vol. 21, no. 12, pp. 25-40, December 1988.

[9] J. L. Hennessy and D. A. Patterson, Computer Architecture, Fourth Edition: A Quantitative Approach, San Francisco, CA: Morgan Kaufmann Publishers Inc., 2006.

[10] D. H. Albonesi, "Selective cache ways: on-demand cache resource allocation," in *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, Haifa, Israel, 1999.

[11] J.-L. Baer and W.-H. Wang, "On the inclusion properties for multi-level cache hierarchies," *SIGARCH Comput. Archit. News,* vol. 16, no. 2, pp. 73-80, May 1988.

[12] G. E. Suh, L. Rudolph and S. Devadas, "Dynamic Partitioning of Shared Cache Memory," *J. Supercomput.,* vol. 28, no. 1, pp. 7-26, April 2004.

[13] M. Weiser, B. Welch, A. Demers and S. Shenker, "Scheduling for reduced CPU energy," in *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, Monterey, California, 1994.

[14] H. Aydi, P. Mejía-Alvarez, D. Mossé and R. Melhem, "Dynamic and Aggressive Scheduling Techniques for Power-Aware Real-Time Systems," in *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, 2001.

[15] W. Wang, S. Ranka and P. Mishra, "A General Algorithm for Energy-Aware Dynamic Reconfiguration in Multitasking Systems," in *VLSI Design (VLSI Design), 2011 24th International Conference on*, 2011.

[16] 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 *Proceedings of the 2000 international symposium on Low power electronics and design*, Rapallo, Italy, 2000.

[17] S. Kaxiras, Z. Hu and M. Martonosi, "Cache decay: exploiting generational behavior to reduce cache leakage power," in *Proceedings of 28th Annual International Symposium on Computer Architecture*, 2001.

[18] K. Flautner, N. S. Kim, S. Martin, D. Blaauw and T. Mudge, "Drowsy caches: simple techniques for reducing leakage power," in *Proceedings of 29th Annual International Symposium on Computer Architecture*, 2002.

[19] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan and D. Tullsen, "Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, 2003.

[20] W. Wang, P. Mishra and S. Ranka, "Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems," in *Proceedings of the 48th Design Automation Conference*, San Diego, California, 2011.

[21] C. Zhang, F. Vahid and R. Lysecky, "A self-tuning cache architecture for embedded systems," *ACM Trans. Embed. Comput. Syst.,* vol. 3, no. 2, pp. 407-425, May 2004.

[22] A. Gordon-Ross and F. Vahid, "A self-tuning configurable cache," in *Proceedings of the 44th annual Design Automation Conference*, San Diego, California, 2007.

[23] T. Sherwood, E. Perelman, G. Hamerly and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, San Jose, California, 2002.

[24] Advanced Micro Devices, "Memory Coherence and Protocol," in *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, vol. 2, 2011, pp. 169-172.

[25] J. Handy, The cache memory book: the authoritative reference on cache design, 2nd ed., Orlando, FL, USA: Academic Press, Inc., 1998.

[26] R. L. Mattson, J. Gecsei, D. R. Slutz and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Syst. J.,* vol. 9, no. 2, pp. 78-117, June 1970.

[27] M. D. Hill and A. J. Smith, "Evaluating Associativity in CPU Caches," *IEEE Trans. Comput.,* vol. 38, no. 12, pp. 1612-1630, December 1989.

[28] G. E. Suh, S. Devadas and L. Rudolph, "A New Memory Monitoring Scheme for Memory-

Aware Scheduling and Partitioning," in *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, 2002.

[29] R. Ubal, J. Sahuquillo, S. Petit and P. Lopez, "Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors," in *Proceedings of the 19th International Symposium on Computer Architecture and High Performance Computing*, 2007.

[30] C. Bienia, "Benchmarking Modern Multiprocessors," 2011.

[31] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Proceedings of the 22nd annual international symposium on Computer architecture*, S. Margherita Ligure, Italy, 1995.

[32] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *IEEE International Workshop on Workload Characterization*, 2001.

BIOGRAPHICAL INFORMATION

Gaurav Puri received his Bachelor of Technology degree in Computer Science and Engineering from Amritsar College of Engineering and Technology, Amritsar, India in 2010. His research interests include computer architecture, parallel programming and code optimization.