# Frequent Value Locality And Its Applications to Energy Efficient Memory Design

by

Jun Yang

---

A Dissertation Submitted to the Faculty of the

Department of Computer Science

In Partial Fulfillment of the Requirements
For the Degree of

Doctor of Philosophy

In the Graduate College

The University of Arizona

2 0 0 2

## Statement by Author

# Acknowledgements

I want to express my gratefulness to my adviser, Professor Rajiv Gupta. He is such a rare researcher who pursues his goal constantly and tirelessly. As a student, I am indebted most to his hard-to-find spirit. During the development of this dissertation, Professor Gupta provided me unlimited help and guidance. He has led me through the darkness and pointed me at the right direction to keep me away from confusion. And he is always there to help, no matter how far he goes and how long he goes. Without those, I would never complete this thesis on time. I gained precious experience from him which I could never get anywhere else.

I also wish to give my deepest appreciation to the care, support, encouragement and love from my mother and father. They have taught me how to live a life with difficulties, how to face ups and downs, how to make decisions and how to stay up with happiness. They are my lifetime teachers. I am thankful to them for giving me their knowledge, wisdom and their love.

Thanks also go to Professor Richard Snodgrass and Professor Christian Collberg. They have provided me scientific discussions and comments that helped me develop a refined manuscript. I also appreciate all those who helped answer my questions, who reviewed my papers and provided insightful comments. You helped me conduct better research during the development of this dissertation.

Last but not least, I want to thank my husband Youtao Zhang for also being my career mate. He has impacted me in many ways including the time management and the creativeness in conducting research. It is always pleasant to discuss difficult problems with him during late nights when both of us enjoy staying in our office. I wish him continuous success in his professional career.

To my dear mother Dekun Sun, my dear father Hongsheng Yang and
my dear husband Youtao Zhang

# Table of Contents

Table of Contents—*Continued*

Table of Contents—*Continued*

# List of Figures

List of Figures—*Continued*

List of Figures—*Continued*

# List of Tables

"Power, not manufacturability, is the challenge we face. Our industry needs to rise to the occasion and deliver innovative solutions that break through the power wall so we continue to deliver value to end users."

— Pat Gelsinge

*Chief Technology Office, Intel*

# Abstract

The need in low power processor design is growing due to the reliability problem for high frequency, high temperature processor chips and the expanding market for battery powered mobile devices. The memory hierarchy is a known source of significant power consumption. This dissertation develops low power techniques for two parts in the memory hierarchy, namely the data cache and the off-chip data bus. The proposed techniques are based on new observations of the memory residing frequent values.

The study on memory values shows that a small set of frequent values occupy a substantial fraction of memory spaces allocated to an executing program. Those values remain fairly stable over a program run. Moreover, the frequent values are distributed in the memory quite uniformly and periodically. Techniques in identifying the set of frequent values through software method and hardware methods are developed. Those techniques are adopted in the low power applications for the data cache and data bus.

A conventional data cache is redesigned into *frequent value cache* (FVC) so that power consumption is reduced for every access of frequent values. However, this comes with a cost of extra cycles for nonfrequent value accesses. To overcome the loss in speed, a load marking technique is developed so that for a substantial number of nonfrequent value accesses there is no degradation in speed. Experimental results of the FVC design show an energy reduction of 28.8% in L1 data cache is achieved.

On the off-chip data bus, an *FV encoding* technique is developed exploring frequent values. The encoding scheme reduces the total bus switching by using "one-hot" codes for frequent values. Variations of the FV encoding technique are also designed to achieve maximum switching reduction across different configurations and different benchmarks. The FV encoding technique can reduce the total number of bus switch-

ing counts 1.5 to 4 times more than that is achieved by other data bus encoding schemes.

In addition to the frequent value based cache design, a *cache access limiting* mechanism is developed to achieve low power from a different angle. A subset of cache accesses is removed by *reusing* their results in history. The reuse hardware is fine tuned to keep the overhead minimum while achieving low power in the data cache. The reuse hardware for the data cache can achieve 11% net cache energy saving.

# Chapter 1
# Introduction

## 1.1 Motivation for Low Power Processor Design

Recently, power consumption has become one of the biggest challenges in high-performance desktop systems. This is because the drive toward increasing levels of performance has pushed clock frequencies higher and has increased the processor complexity. Both increases come at a cost of high power consumption. The costs associated with packaging, cooling and power delivery have thus jumped to the forefront in the microprocessor industry.

To get an idea of the trends in power consumption of today's processor consider the following table taken from the power study on Alpha processors [65]. The Alpha processor family is multi-issue, out-of-order execution high performance processor. We can see clearly the drastic growth of the power and its density as well. This increase would also negatively impact the processor reliability if the power dissipation keeps increasing at this rate. Even though reducing the supply voltage is well known as an efficient way of controlling power consumption, its benefits are more than offset by the increased complexity and frequency. This calls for creative architecture solutions that can focus on high level trade offs between power and performance.

| Alpha Model | Power(W) | Frequency(MHz) | Die size($mm^2$) | Voltage(V) |
|---|---|---|---|---|
| 21064 | 30 | 200 | 234 | 3.3 |
| 21164 | 50 | 300 | 299 | 3.3 |
| 21264 | 90 | 575 | 314 | 2.2 |
| 21364 | >100 | >1000 | 340 | 1.5 |

Table 1.1. Power Trends for Compaq Alpha

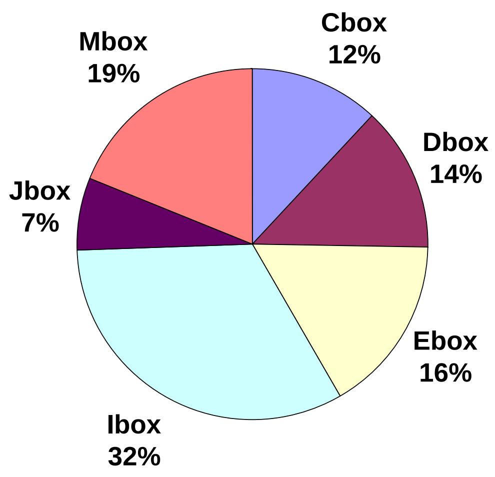
The need to limit the power consumption is also crucial for portable computer platforms such as cellular phones, palm handhelds and pocket PCs because those devices are battery powered. Given the type of applications being written for mobile devices, there is an increasing demand for delivering high quality multimedia output. Since the advances in battery technology are limited, designing low power processors that can operate with a light weight battery for long duration is imperative. Table 1.2 shows the trends in power consumption for typical embedded processors: ARM7 to ARM10 [3]. They are simpler designed in-order execution pipelined processors. We can see from the table that the range of power consumption is only the order of MilliWatt. The reason for this big difference is due to much simpler architecture design. Future embedded processors will have more complex structure such as deeper pipeline length and branch prediction. The designs will resemble high performance processors but under different constraints. Therefore, limiting the power consumption is also becoming more and more important for embedded processors.

| ARM Model | Power(mW/MHz) | Frequency(MHz) | Die size($mm^2$) | Voltage(V) |
|---|---|---|---|---|
| ARM720T | 0.2 | <100 | 1.8 | 0.9 |
| ARM920T | 0.35 | <230 | 6.0 | 0.9 |
| ARM1020E | 0.8 | <375 | 6.9 | 0.9 |

TABLE 1.2. Power Trends for ARM Family (0.13$\mu$m technology)

Let us look at the power distribution of the Alpha 21264 and ARM 920T. Figure 1.1 is taken from the power study in Alpha processors [65] and Figure 1.2 is taken from a tutorial on low-power processor [54].

In the Alpha 21264, the EBox and IBox represent the integer and floating point execution processor core. The summed power expenditure of those two is 47%, much higher than the ARM processor core (25%). This is because the 21264 has complex four-issue out-of-order speculative execution pipelines while the ARM 920T has only a single-issue in-order execution pipeline. The more complex the design the higher

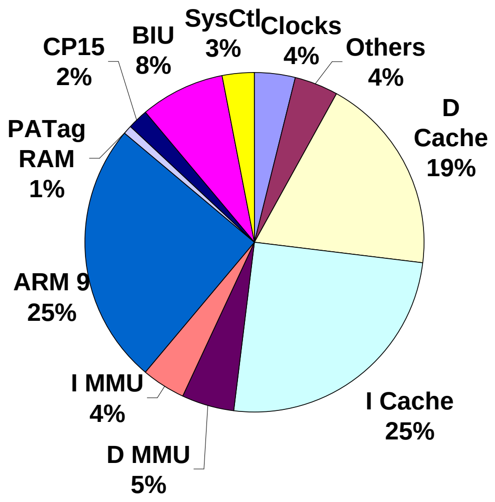


- CBox: Bus Interface Unit, Data and Control Buses
- DBox: Data Cache
- EBox: Integer Units
- IBox: Integer Mapper and Queue, FP Mapper and Queue, Instruction Data Path
- JBox: Instruction Cache
- MBox: Memory Controller (Load/Store Queues, TLB etc.)

FIGURE 1.1. Power Distribution for Alpha 21264

power it consumes. The IMMU, DMMU, PATag RAM and the CP15 in ARM 920T contribute in various memory requests and handlings, similar to the function of the MBox of the 21264. The total power of those components is 12%, less than the 19% for MBox in 21264. One reason for that is the ARM uses the virtual address caches to partially avoid the address translation. The BIU and SysCtl in ARM 920T is comparable to the CBox of 21264. The former has total 11% of power consumption and the latter has 12%.

From the power distribution in both figures, we can see that the execution core and the cache memory system take up most power out of the total. This thesis focuses



- D Cache: Data Cache
- I Cache: Instruction Cache
- D MMU: Data Memory Management Unit
- I MMU: Instruction Memory Management Unit
- ARM 9: ARM execution core
- PATag: Physical Address Tag RAM
- CP15: Control Coprocessor
- BIU: Bus Interface Unit
- SysCtl: System Controller
- Clocks: Clock driver and network

FIGURE 1.2. Power Distribution for ARM920T

on reducing the power in the cache memory system. It is notable that caches consume significant amount of power in both processors, totaling up to 44% for ARM 920T and 21% for Alpha 21264. The instruction and data caches in ARM 920T amount for larger portion of power because they adopt higher associativity to amend the low hit rate due to smaller cache sizes.

The off-chip buses are also a significant source of power loss, although not dominant. They are usually very wide—the standard PC memory bus includes 64 data lines and 32 address lines. Each has capacitance that is orders of magnitude higher than the internal bus and requires substantial driving power. It is not unusual for a chip to expend 15–20% of its power on these off-chip drivers [42].

Overall, the memory system is critical in processor design because it affects both the performance and the power consumption. Usually there is a trade off between performance and the power. The modern designs tend to enlarge the cache sizes at different level to maximize the performance, driven by the increasing speed gap between CPU and memory. Larger cache sizes and higher clock frequency will in turn increase the power consumption of the memory system in the future processors.

## 1.2 Thesis Contributions

The previous section discussed the power distribution for two representative processors. Even though the two processors have completely different designs, they both exhibit a common feature that the memory hierarchy is one of the dominant sources of power dissipation. This includes the on-chip level one caches, the off-chip buses, and the off-chip lower level cache/memory hierarchies. This thesis investigates revolutionary low power solutions targeting different points in the memory hierarchy.

Figure 1.3 gives the high level picture of the major three low power solutions developed for a typical microprocessor. The first reuse mechanism is applied to all the memory instructions. A memory instruction is considered reusable if for read

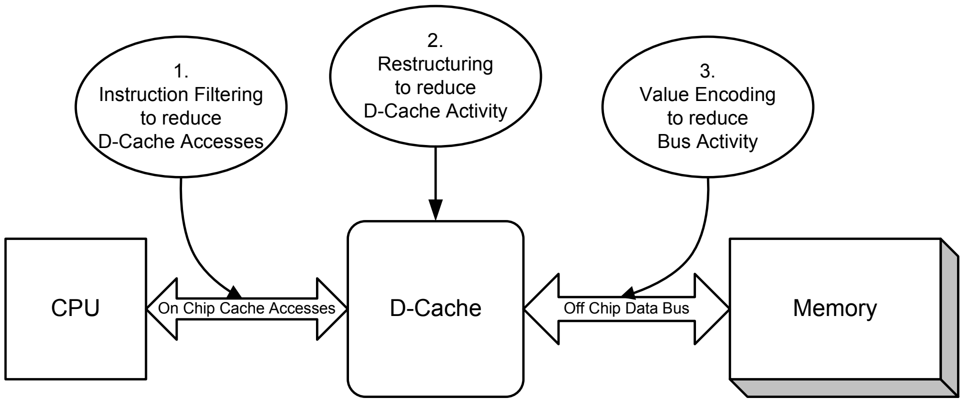


FIGURE 1.3. Low power designs included in this thesis

operations we can obtain its result without accessing the cache, or for write operations accessing the cache is unnecessary. The reuse checking is performed just before the instruction accesses the data cache. Upon successful reuse, it turns back to prepare for committing, saving one access to the data cache.

The data cache restructuring is tailored for the data array inside the D-cache. The array is partitioned into two sub-arrays, one with a shorter word width and the other with a longer word width. Data accesses are categorized so that a majority of them can be satisfied by only activating the shorter word width array and the rest of the accesses need both but at longer time. This design trades a little performance for big power savings since accessing the full data array consumes more energy than accessing a partial array.

The encoding algorithm is applied on the off-chip data bus as shown in Figure 1.3. It encodes the values streams evicted from the on-chip data cache and the value streams transferred from the off-chip memory. The algorithm exploits temporal locality of the values appeared on the data bus and transforms the values into codes so that the number of switching wires [1] between neighboring values are reduced as much as possible. Experiments showed that the proposed algorithm outperforms other

[1] wires that change state from last cycle to current cycle, i.e. switch from low state to high state or high state to low state

existing encoding schemes while maintaining low overhead in encoder and decoder.

The second and the third low power solutions are based on the *frequent value phenomenon* that is discovered throughout the memory hierarchy. It is a characteristic of the values that appear in the memory that are relevant to a program. The essential idea is that a small set of distinct values happen very frequently in memory. Similarly, many values transfered across the bus occur repeatedly also. We called this the *frequent value phenomenon* and studied its interesting properties to better apply it to our low power designs.

In summary, the major contributions in this thesis are as follows.

1. A memory instruction reuse mechanism is developed to limit accesses to the L1 data cache, reducing the total energy of the data cache with only small overhead.

2. A restructuring of the L1 data cache scheme is proposed. With the newly designed cache, the accesses that are due to frequent values spend only one third or less of the energy than usual. This new design is also applicable to higher level cache or memory in the hierarchy. The redesigned cache slows down the processor by no more than 4%.

3. An efficient encoding algorithm is developed for off-chip data buses. This encoding algorithm is able to reduce the switching activity on the data bus by around 30%.

## 1.3 Thesis Organization

In Chapter 2, the background of the low power research in different types of processors is first briefly introduced. Then the related research in low power cache and memory, existing bus encoding research and the previous work in computation reuse is discussed.

Starting from Chapter 3 is the body of the thesis. First is the elaboration on the discovery of the frequent value phenomenon. Three distinct properties of frequent values are established through extensive experimental results. Chapter 4 provides technical methods in identifying the frequent values in various applications. Three methods in accordance with different application scenarios are proposed. Those methods contribute in the low power cache and bus designs in later chapters.

Chapter 5 illustrates the frequent value cache design specifics including constraints we endeavor to meet and the difficulties we solve. Chapter 6 develops the bus encoding algorithm to its fullest extent. A number of variations are included to reduce both the switching activity and the coder overhead.

In Chapter 7, the memory instruction reuse techniques is described. It first presents the reuse opportunities that exist in programs and then develops algorithms to catch the potential possibilities. It also includes methods to fine tune the reuse hardware to achieve low overhead.

# Chapter 2
# Background and Related Work

A great amount of research has been performed in the design of energy efficient systems. The techniques developed can be broadly categorized into *circuit level* techniques and *architecture level* techniques. These two types of techniques are complementary and therefore can be combined to maximize energy savings. In this chapter, some common methods used in circuit level low power design are first summarized. Then some important architecture level techniques are discussed. Various existing low power techniques in cache design, bus encoding, and computation reuse are discussed respectively. Finally, the simulation environments and metrics used in low power research including the environment used in this thesis are described.

## 2.1 Circuit and Logic Level Techniques

The low level power optimization methods scale technology related parameters in different ways. The major parameters determining power consumption are: supply voltage, operating frequency, effective capacitance and switching activity. Equation 2.1 gives the first order approximation of power consumption at CMOS circuitry level [42].

$$P \approx A \times C \times V^2 \times f \tag{2.1}$$

The above model measures the *dynamic power* consumption caused by the charging and discharging of the capacitive load on gate outputs. Dynamic power is the dominating part of overall power consumption in current technologies. In the above equation, $A$ is the number of switching of the gates, $C$ is the total effective capacitance seen at the gate outputs, $V$ is the supply voltage and $f$ is the operation frequency of the system. There have been many approaches proposed to reduce the value of these different factors or their combinations in this equation in order to lower $P$.

Besides *dynamic power*, the *static power* is also consumed even when there is no activity. It is the product of the supply voltage and the *leakage current* which is independent of clock rate and is present once the processor is powered on. Even though static power is not a dominant factor, under certain circumstances the leakage current could go up, increasing the static power. Since the dynamic power is the major portion in the total power consumption, some typical approaches in reducing the dynamic power are summarized and limitations are discussed next.

### Voltage Scaling

Equation 2.1 shows that there is a quadratic relationship between power and supply voltage. Reducing $V$ is therefore an effective way to reduce power. A large body of research that has been devoted to this technique [14].

However, voltage scaling has its own trade off. Reducing the supply voltage causes the circuit to run slower [6]. This will eventually make the applications run for a longer time, which in turn consumes more energy since energy is the product of power and time. Moreover, scaling supply voltage will increase *leakage current* and *leakage power* which is another factor to diminish the advantages. High leakage current also makes it difficult to design dynamic circuits, caches, sense-amps, etc. Because of the fundamental limitations stated above, voltage scaling in future techniques will have only marginal practical impact.

### Reducing Frequency

Another way to reduce the power is by reducing the clock frequency. The power will decrease almost linearly in operating frequency. It has also been found that a lower discharging rate will help maximize the total amount of battery energy. Therefore, lowering the clock frequency could prolong the time between battery recharges [41]. However, programs run slower with lower frequency and the total energy consumed may or may not decrease. The following equation explains the reason.

$$E = P \times T = P \times I \times CPI \ / \ f \tag{2.2}$$

In the above equation, $E$ stands for energy, $P$ represents power, $T$ is time, $I$ is the total number of dynamic instructions of a program, $CPI$ means the average number of cycles needed by each instruction and $f$ is the clock frequency. It is clear from the equation that $E$ depends on both $P$ and $T$. $T$ is proportional to $I \times CPI$ and the inverse of $f$. Given a program and an architecture, the product of $I$ and $CPI$ remains same under different $f$. Therefore, lowering $f$ will increase $T$ but decrease $P$. Consequently, the variation in $E$ is difficult to predict if simply the $f$ is reduced.

### Reducing Capacitance

The third parameter in 2.1 is the effective total capacitance. Current methods to reduce $C$ is through downsizing the transistor, reducing the number of fan-out gates, and decreasing the wire capacitance. However, transistor scaling is limited by device physics and silicon-compatible material constraints [64]. The wire capacitance is very difficult to compute due to the layout and the cross-talk between close by wires [6].

Next we will see how reducing the activities can have significant impact on lowering power consumption.

### Reducing Switching Activities

The fourth factor is the dynamic switching activity. Minimizing this factor is effective in power reduction when the chip technology and the supply voltage is set. Reducing switching activities is also a very flexible in design as the architects have the most design space at different levels to reduce useless switching. The idea here is to minimize activities of certain functional units or chip areas that are not performing useful computation. Typical techniques at the hardware level that have appeared in the literature are the following.

1. **Clock Gating** The power consumed by the clock network, which includes the clock generator, the clock drivers, the clock distribution tree, the latches and the clock loading due to all clocked elements, is more than 40% in high-performance processors [62]. Clock gating, for this reason, has been widely employed to turn off those parts of the clock tree to latches or flip-flops that are not being used in each cycle [9, 25, 27]. It can be implemented by adding special "enable/disable" signal gates to the clock network. And the low area and performance overhead is paid off by the significant amount of power reduction.

   Unfortunately, there are a number of issues that must be considered in clock gating. The most important concern is that the disabled block may not power up in time, or that modified clocks may generate glitches. Other issues such as clock skews and high transition current all make clock gating more difficult to design new CPUs.

2. **Sleep (Standby) Modes** Many state-of-art processors have built-in sleep (standby) modes. Typically, the clock is stopped for all but certain sections of the processor after the default period of inactivity. The processor does not perform any work or performs very little work while asleep resulting in significant power savings. However, there is a long latency for the system to wake up so it is profitable to put the system into sleep mode only when it is expected to sleep for a relatively long time.

## 2.2 Architecture Level Techniques

Most architecture techniques trade off a little performance for lower power consumption, achieving overall energy reduction (see equation 2.2. Typical approaches include using simpler or smaller functional units, turning off part of functional units when they are not in use, removing certain redundancy and reusing computation results. Below, some novel techniques that are applied to different architecture components are briefly summarized.

### 2.2.1 Low Power Designs for Processor Core

For most high performance processors such as Alphas and Pentiums, the bulk of power goes into the pipeline issue logic. Various techniques have been proposed targeting this high power component. Since the primary goal of those processors is high performance, most techniques reduce the issue logic activities without much degradation in instruction throughput. Bahar *et al.* developed the Pipeline Balancing algorithm to dynamically adjust issue width on demand [5]. The algorithm is based on the observation that application programs do not always execute at their peak IPC. Therefore, the issue logic does not need to operate always at full width. In another design [22], it was observed that the issue logic wasted energy in trying to wake up empty instruction queue slots and already ready for execution instructions. Instead of limiting the searching width, they proposed to dynamically resize the instruction queue. This approach cuts off the power in wake-up activities that were performed beyond the dynamic queue length. In addition to resizing the issue width and instruction queue length, a more aggressive method in tuning resource sizes was developed [46]. In that method, the power sinks on data path as instruction queue, reorder buffer and load store queue were all allowed to vary sizes according to the needs of the executing program.

There are other techniques in reducing the power of the pipeline. For example, compressing the significant bits of values flowing through the pipeline can be used to reduce the ALU, register files and internal latches activities [13]. The technique is even extended to instruction and data caches as well.

Next, some representative low power solutions for caches the have appeared in the literature are described.

### 2.2.2 Energy Efficient Cache Designs

**A Conventional Cache.** Figure 2.1 plots the typical $m$-way set associative cache structure. It consists of *Tag arrays*, *Data arrays*, some comparators, multiplexers and internal latches. The major power spent in the cache is the tag comparison and data reads and writes. The tag comparison involves reading $m$ tags from the *Tag arrays* and performing $m$ comparisons. Data reads and writes involve activating $m$ cache lines from the *Data arrays* and selecting the right word in the right line.

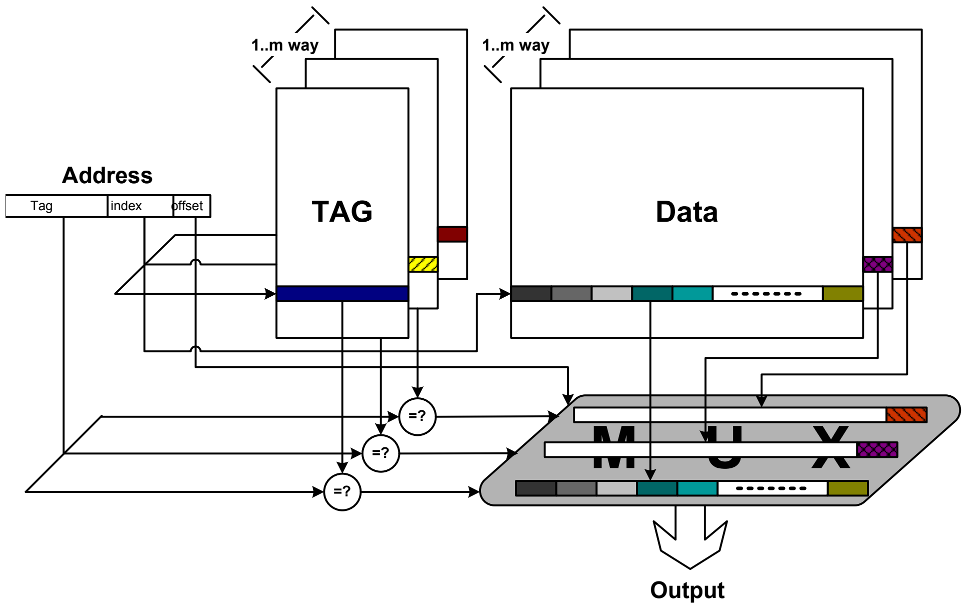


FIGURE 2.1. The structure of a general cache (derived from the standard set-associative cache structure [29])

**Energy Efficient Designs.** The problem of reducing the energy consumption of the conventional cache can be explored from different angles and in different cache components.

1. **Reducing Power for Set Associative Caches.** Traditional caches are designed for maximum performance. For example, a set associative cache outperforms a direct mapped cache because it decreases cache misses by probing multiple entries of the cache in parallel. However, a cache hit appears in only one entry, meaning that the energy spent in probing other entries are wasted. The Way-Prediction and Selective Direct-Mapped schemes were proposed to solve this problem [48]. Alternatively, certain cache ways can be disabled when the program has modest cache activity [2]. Even though the performance degraded due to less number of available cache ways, an overall energy saving was still achieved.

2. **Reducing Power for Tag Path.** The tag path of the cache consumes relatively high power. It mainly comes from the comparators that try to determine a hit or a miss. For this reason, methods were proposed to reduce the number of tag checks [45, 67]. One approach is that the compiler determines those loads and stores that are guaranteed to access the same cache line [67]. Those instructions can directly access the data arrays without tag checks. Since the instruction addresses are usually sequential, such a removal can be applied to the instruction cache effectively [45]. This technique can be implemented purely at the hardware level.

3. **Reducing Power for Data Arrays.** To diminish the energy spent in data arrays, one can either reduce the dynamic energy or the static energy. Dynamic energy is consumed when a cache line is driven for reading or writing data. It can be cut down by limiting the length of a line that needs to be driven. More specifically, only one word in a line, indicated by the offset field in the address, needs to be driven for reads and writes [24]. This involves wiring changes to the data arrays. The scheme saves energy in driving the other words that are discarded later in the same line.

Static energy is always consumed even when there is no activity. Kaxiras *et. al.* and Flautner *et. al.* introduced techniques to shut off the cache lines or bring them to a *drowsy state* [21, 35]. The lines being shut down are those that will not likely be accessed in the near future. Both techniques significantly reduced the static cache energy.

4. **Reducing Power Based on Values.** There are some designs that are based on the contents stored in the cache. Villa *et al.* found that data caches contain a lot of zeroes [63]. They introduced a compression method for zeroes inside the cache such that they can be represented by only a single bit. Reading or writing a single bit is much cheaper than 32 bits in energy consumption.

5. **Other Techniques.** Miscellaneous techniques including specializing caches [31], sequentializing cache accesses [28] code compression for instruction caches [38] are all valuable.

This thesis presents two power reduction techniques. The first technique focuses on reducing the power spent by data arrays. The average cache access power is reduced through data array restructuring. The second technique focuses on limiting the number of accesses to the cache through the reuse of instructions.

### 2.2.3 Bus Encoding Techniques

Existing bus encoding algorithms can be categorized into *address* bus encoding, *data* bus encoding, and *general purpose* bus encoding. There are some very good encoding algorithms for address buses, especially instruction address buses, because most of the address stream is sequential. However, a similar regularity does not exist in data buses since it is assumed that the value stream on the data buses is randomly distributed. Thus, the encoding algorithms for the data bus are mostly limited to statistic investigation. The general purpose encoding algorithms do not result in significant switching reduction. This is because the data value streams and address streams exhibit very different characteristics. It is therefore difficult to develop a general algorithm that is effective for both.

**General Purpose Encoding.** A very simple encoding algorithm called Bus-Invert was proposed by Stan *et al.* [59]. In this scheme, either the original address or its binary inverse is sent on the bus depending on the Hamming distance between the current address and the previous transmitted value. The rule is to always send the value that would cause the number of switching that is less than half of the total bus width. This scheme is applicable to both address and data buses and is adopted in many other encoding schemes because of its simplicity. Ramprasad *et al.* developed a framework [50] for generic encoder-decoder architecture. They also proposed an adaptive method that requires huge hardware overhead.

**Address Bus Encoding.** Gray coding [61] has been proposed to minimize the switching on the instruction address bus. The encoding scheme ensures that when the address is sequential, there is only one switch between two consecutive address words. T0-C coding was developed by Aghaghiri *et al.* [1]. It freezes the bus when the addresses are sequential. The bus transmits values normally when the address is non-sequential. The Working-Zone-Encoding [43] is developed based on memory reference locality. The memory regions being referenced by a program are divided into working zones. Instead of transmitting a sequence of complete addresses that exhibit locality, in this technique, the offset of current reference with respect to the previous reference to the same *working zone* is sent over the bus, along with an identifier of that zone.

**Data Bus Encoding.** As mentioned earlier, there are not many encoding schemes for the data buses. Even though the generic methods can be adopted, the achievable switching reduction is modest. Benini *et. al.* presented the *adaptive* encoding in which new codes are generated based on the past $N$ data samples [7]. This mechanism has huge hardware overhead so they developed further techniques in scaling down the sampling and encoding sizes.

This thesis introduces a new encoding scheme for data buses that is based on observations of the data streams sent on the bus. The frequent value characteristics are utilized into the encoding algorithms. The algorithm achieves a significant reduction in switching counts.

### 2.2.4 Computation Reuse

The reuse mechanism has been exploited in many papers [4, 18, 30, 53, 57]. It is based on the empirical observations that many instructions, and groups of instructions, having the same inputs and outputs are executed repeatedly. Those instructions can be identified either dynamically [4, 30, 53, 57] or statically [18]. In dynamical instruction reuse, the inputs and outputs of the instructions are memorized in some hardware together with necessary tags. On successful reuse test, the result can be obtained directly from the hardware instead of the functional units [57] and [4]. More aggressively, the reusability was extended to block level [30]. There the authors exploited the inputs of a basic block and reuse the result of the block. This approach incurs expensive hardware overhead since all the block inputs and outputs need to be saved.

This thesis exploited a reuse technique for memory instructions. It aims at getting the results at an early stage in the pipeline so that the dependent instructions can read the value earlier. Consequently, an access to the cache can be saved to reduce the cache power dissipation.

## 2.3 Simulation Tools

For computer architecture researchers, realizing novel designs in hardware is too expensive and time consuming, especially in an early stages of development. Therefore, most research relies on simulation tools that run at a tractable amounts of time for real sized programs, and provide reasonable accuracy. Most of all simulation tools are easily extensible.

The SimpleScalar tool set [11] is a widely used simulator in the modern processor architecture research community. The tool set simulates a slightly simpler MIPS-IV architecture and provides from an extremely simple and fast functional simulator to a detailed out-of-order issue processor simulator. The tool set contains a GCC based compiler and utilities that help generate MIPS object code. The advantage of SimpleScalar is that it is fast, flexible and efficient. Written in C code, the tool allows users to easily incorporate new designs into the simulator within a reasonable amount of time.

The success of SimpleScalar has made its power evaluation extension easy. SimplePower [75] is the in-order 5-stage SimpleScalar simulator augmented with an energy estimation tool. It uses transition sensitive energy models to estimate the energy spent by processor components based on their states transition. SimplePower's limitation is that no energy estimation is available for out-of-order superscalar processors. The problem is solved in Wattch [10] which can model energy for different types of complex processors simulated by SimpleScalar. Wattch models the dynamic energy consumption of each major processor component and computes the accumulative energy every cycle. Though fast and simple, Wattch has been criticized for its large error in computing energy consumption for typical processors. Other processor energy simulators such as TEM$^2$P$^2$EST [19] and AccuPower [47] are more accurate but are not available for public use yet.

The study of the memory hierarchy calls for similar cache and memory simulators.

Some early cache simulation tools such as CacheProf [55] and Dinero IV [20] are effective functional simulators that produce mainly cache references, hits and misses information. Those tools are trace-driven for fast running time. They do not provide cache timing information which is desired for performance analysis. The newer version of SimpleScalar now has its own cache simulator of up to two levels. Cache and memory hit/miss latencies are calculated so that the performance impact of memory hierarchy can be easily studied.

As the interest in energy efficient memory hierarchy designs increased, energy models for cache and memory also emerged. The CAPE [34] tool used an analytical model to estimate the power dissipation in caches. This model is then adopted in SimplePower with enhancements that includes the off chip memory energy as well. By far, the most popular cache timing and power tool is the CACTI series [66]. CACTI 1.0 [32] models access time for non-fully associative caches. CACTI 2.0 added [52] modeling support for fully-associative caches, a power model, technology scaling, multiported caches, and improved tag comparison circuits, as well as other improvements to CACTI 1.0. CACTI 3.0 [56] includes modeling support for the area and aspect ratio of caches, caches with independently addressed banks, reduced sense-amp power dissipation, and other improvements to CACTI 2.0. The Wattch [10] simulator incorporated CACTI 2.0 into its cache energy models. CACTI has been extended to XCACTI by Renau *et. al.* in [31]. The XCACTI provides energy measurement for not only read operations, but also write operations, write back and line fill operations on cache misses.

In this thesis, a cycle level simulator FAST developed by Onder *et. al.* [44] is used for processor cores. FAST can automatically generate simulators from an architecture description language (ADL). It currently supports the MIPS ISA which is also written in ADL. Three types of simulators are used: (1) a simple fast functional simulator that simply executes instructions one by one as if they are going through a single staged pipeline (Figure 2.2); (2) an in-order processor simulator which simulates the

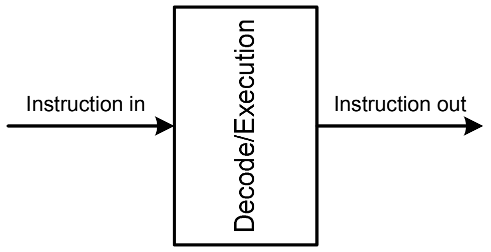


FIGURE 2.2. A Simple One-Stage Simulator.

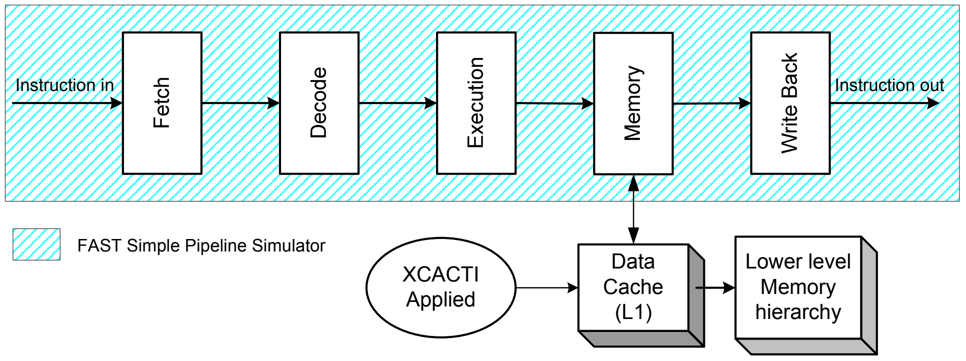


FIGURE 2.3. A Pipelined Simulator.

standard five stages pipeline (Figure 2.3) and (3) an out-of-order superscalar simulator (Figure 2.4). The first simulator is used mainly to generate trace information such as the memory instruction sequence for studying the cache access behavior. The second simulator is used in investigating embedded processor designs and the third is for exploring high performance superscalar processor designs. The research work contained in this thesis contributes to the FAST system in that the memory hierarchies are added to the previous processor cores. The memory extension can simulate as many levels of cache as possible, either unified or split, from direct-mapped to fully associative cache. Moreover, the memory system can output accurate latency information as well as energy consumption statistics which is obtained through plugging in the XCACTI energy model (Figure 2.3 and 2.4). In designing new energy efficient caches, the XCACTI tool is also modeled such that it complies with the modified caches. The enhanced FAST system allows us to investigate the memory hierarchy impacts on both performance and energy consumption. We also extracted energy models from Wattch for non-cache structures such as array structure in modeling the energy in hardwares such as a simple indexing table.

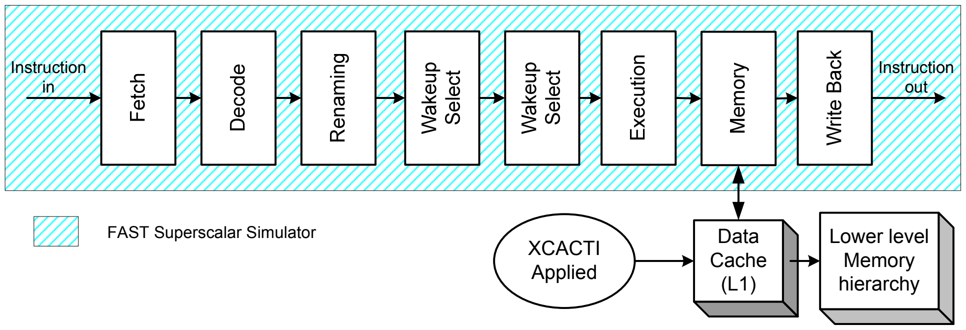


FIGURE 2.4. A Superscalar Simulator.

## 2.4 Energy Measurement Metrics

Using correct metrics in experimental evaluation is crucial and has been studied by Gonzalez and Horowitz [25]. Earlier researchers have used only the *energy* metric in evaluating new designs. Unfortunately, this is misleading since the energy can be reduced dramatically by slowing down the processor or cutting down the supply voltage as we have discussed earlier. Therefore, Gonzalez and Horowitz proposed an *energy*delay* metric to combine both factors. The *delay* factor is the total execution time of a program, usually measured in terms of number of cycles. The *energy*delay* metric better describes the improvement or deterioration of new designs that trade performance for energy. Since then, this metric is widely used in the low power research community.

The following chapters contain the body of this thesis starting with our frequent value observations which is described next.

# Chapter 3 Frequent Value Phenomenon

Recent research has demonstrated that values produced by executing instructions exhibit a high degree of value locality, that is, multiple executions of the same instruction often produce the same value [23, 40]. Value locality has been exploited in the design of value reuse and prediction mechanisms for superscalar processors.

In this chapter another kind of locality is identified, termed *frequent value locality*; this is also quite prevalent in programs. The first aspect of the frequent value locality is that if the values involved in memory accesses are tracked, it can be observed that at any given point in the program's execution, a small number of distinct values occupy a large fraction of these referenced locations. In fact it is observed that on average in fifteen of the `Spec95` [1] [58]. programs, eight distinct values occupy 48% of all allocated memory locations throughout the execution of the program. The second aspect of this phenomenon is that the set of frequent values remains quite stable throughout the execution of the program. The third and final aspect of frequent value locality is that frequent values are scattered fairly uniformly throughout the memory.

## 3.1 Characteristics of Frequent Values

The *frequent value locality* phenomenon characterizes the behavior of values being held in live memory locations of running programs. The following three properties of the values characterize frequent value locality. These properties are demonstrated

[1] The Spec95 benchmark suites were released by the Standard Performance Evaluation Corp. (SPEC) on August 21, 1995. The suites provided the worldwide standard for measuring and comparing computer performance across different hardware platforms. The Spec95 comprises two sets (or suites) of benchmarks: CINT95 for compute-intensive integer performance (8 programs) and CFP95 for compute-intensive floating point performance (10 programs). The Spec benchmarks were selected from existing application and benchmark source code running across multiple platforms.

by analyzing the behavior of 15 `Spec95` benchmarks when run on *reference inputs.* The left 3 benchmarks do not go through the FAST simulator due to incompatible libraries, therefore are not experimented in this experiments.

### 3.1.1 Property I: Frequent Value Occurrences

*A small number of frequently occurring values, called* ***frequent values****, occupy a substantial fraction of memory locations allocated to an executing program.*

To establish the above property we ran the benchmarks and examined the values in memory locations every 10 million instructions and averaged the frequencies of the values over the entire set of collected samples. During each sampling point, the entire memory space was scanned through and every distinct value was ranked according to its occurrence frequency. The memory locations that were considered at a given point included those that were of interest to the program. In particular, the currently allocated stack and heap memory locations were considered. After the completion of the program's execution, for each encountered value, its average frequency across all sampling points was computed. The resulting average frequencies of all values were sorted in descending order. Values at the top of the list are more frequent than the values that appear later in the list. A significant amount of time collecting this data was needed as a program run typically involved execution of several billion instructions.

Figure 3.1 shows that 12 out of 15 benchmarks exhibit this property and on an average around 48% of memory locations are occupied by the top eight frequently occurring values in the 15 `Spec95` benchmarks that were used in this study. The top 8 frequent values are listed in Table 3.1. Examination of these values shows that there is a mix of small values (that can be represented using 16 bits) and large values (which require more than 16 bits). While the same small values (e.g., zero) are often observed across different programs, the same is not true for large values. This is

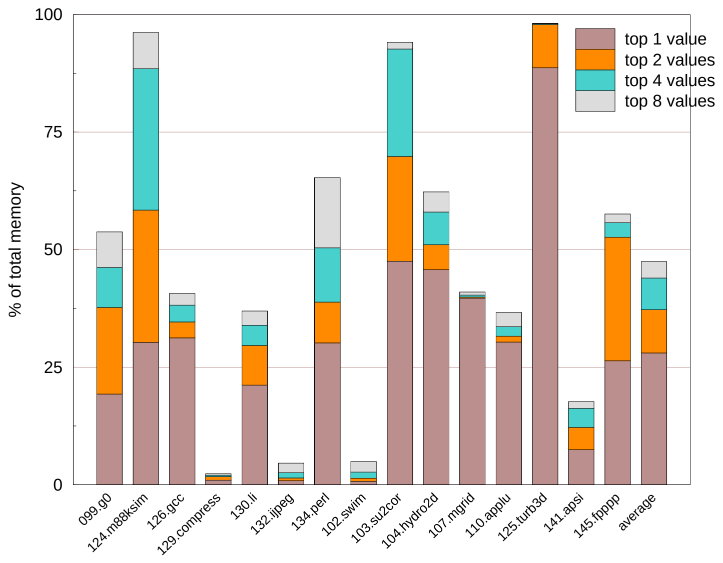


FIGURE 3.1. Amount of memory occupied by top 8 frequent values.

because the large values are often memory addresses or string constants. Figure 3.2 shows what fraction of locations were occupied by small frequent values and large frequent values. In some programs the large values occupy a substantial number of locations.

| Benchmark | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 099.go | 0 | 351a | 4 | 1 | 2 | 3 | 349 | 1c1 |
| 124.m88ksim | 0 | 4022ada0 | 1c | 40229030 | 4022a610 | 60d12 | 8048bf7d | 1db82340 |
| 126.gcc | 0 | e7 | 403 | 80004 | 40252734 | 10001 | 20 | 1b |
| 129.compress | 0 | fffffff | 65687420 | 20656874 | 61687420 | 90a0a0a | 68742065 | 20656820 |
| 130.li | 0 | 3 | 1 | 4 | 6 | 1000000 | 5 | 40280df4 |
| 132.ijpeg | 0 | 1 | ffff | 10000 | ffff0000 | fffffff | 10001 | 1ffff |
| 134.perl | “xxxx” | “x x ” | 0 | “ xx ” | “x xx” | “xx x” | “ x x” | “xx” |
| 102.swim | 47435000 | 47435001 | 47434fff | 47435002 | 47434ffe | 47435003 | 47434ffd | 47435004 |
| 103.su2cor | 0 | 3fe00000 | 40040000 | 807bcdaf | 3fd5f8e1 | 40290000 | 3fec71bc | 390cf5ba |
| 104.hydro2d | 0 | 3feccccc | cccccccc | 3fe33333 | “3333” | 3ff00000 | “0000” | bc400000 |
| 107.mgrid | 0 | 80000000 | 3c300000 | bc300000 | 3c400000 | bc400000 | bc200000 | 40000000 |
| 110.applu | 0 | 2752547 | 4189374c | bfe16c8b | 43958106 | 3f7b089a | 3f90e560 | 80000000 |
| 125.turb3d | 0 | 80000000 | “ ” | 3ff00000 | “::::” | 1 | 6 | 3bc79ca1 |
| 141.apsi | 0 | 3fb99be4 | d443f3ee | d443f3ef | d443f3f0 | 3fb99999 | 9999999a | “ ” |
| 145.fpppp | 9999999a | 3fc99999 | 0 | 33333333 | 3fd33333 | 47ae147c | 3fb47ae1 | 3fa47ae1 |

TABLE 3.1. Frequently occurring values ordered by decreasing frequency.

We have identified several reasons for the frequent value phenomenon. The first reason is that programs usually contain many constants such as character constants

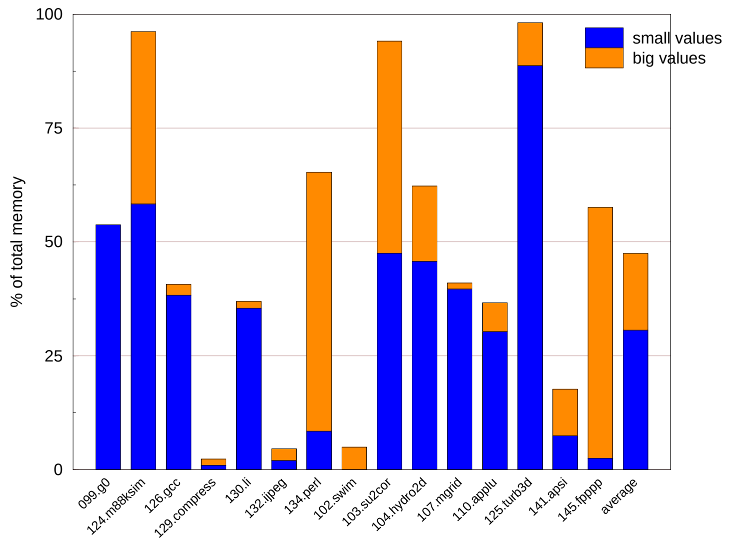


FIGURE 3.2. Amount of memory occupied by small vs. big values.

and NULL pointer value, or, near constant values such as boolean variables which toggle between 0 and 1. Our study shows that they not only exist in register but also in memory as well, even when the program is compiled at `-O3` optimization level. The second reason is that many programs contain dynamically allocated data structures such as hash tables or binary trees. Accessing to different nodes frequently requires to start from the same head of the bucket or the root of the tree. The third reason is that the frequent values sometimes come from the inputs of a program such as a text file input of a compiler. There are only limited keywords for a language that repeatedly occur in the file. And the compiler usually loads the bulk of the file into the memory before compiling.

Currently, the frequent values are observed in the 32-bit machine applications. If the word width is increased to 64 bits, the frequent values will not be affected greatly for the following reasons. First, the small program constants will be represented in 64 bits by simply having sign extension in the high ordered bits. Second, the string constants will also be frequent except that they have different lengths. More specifically, if a frequent string is shorter than 32 bits it will still be frequent in a

64-bit machines with the same length because values are word-aligned. If a frequent string required multiple 32-bit words (each 32-bit substring must be frequent also), it is then represented in half number of 64 bits in the 64-bit machine. If the frequent values are memory addresses, their binary representations will be different in two different types of machines. However, the existence of frequent values is a property of a program. Those 32-bit frequent memory addresses will still be frequent in a 64-bit machine except that the binaries are in different forms.

### 3.1.2 Property II: Frequent Value Stability

*The set of frequently occurring values remains fairly stable over a program run which implies that frequent values can be identified and exploited during a program run.*

To observe this property the occurrences of frequent values throughout the program execution were studied. The graphs in Figure 3.3 and 3.4 show the behaviors of the benchmarks over their entire execution. To generate these graphs, the top ten frequent values were first found using the methodology already described above. Next the programs were run again and during these runs, at each sampling point, for each of the top ten frequent values, the number of memory words that contain the frequent value is noted down.

In the graphs the X axis represents time and the Y axis represents the memory occurrence characteristic of the various frequent values. To make the graphs more readable we did not plot these graphs for entire execution of the program but instead carefully reduced the duration to a period in the middle of program's execution. However, when the scope of the data presented is narrowed, the data for the entire execution is first examined and we only narrowed the duration for which the data is presented if the behavior of the program was similar for the remainder of the execution. In these graphs the top-most line represents the total number of allocated memory locations. The subsequent curves give us an idea of how many locations

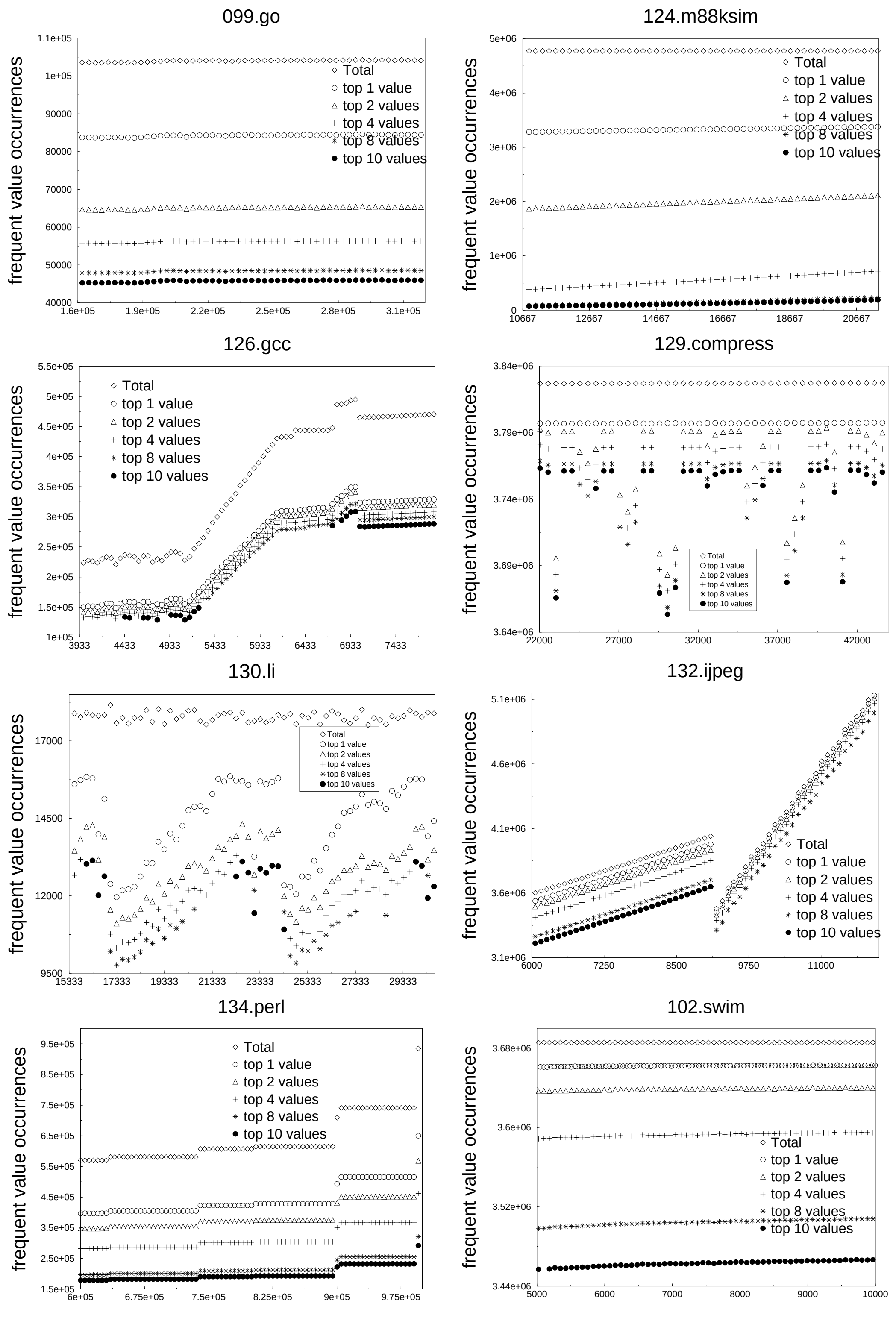


FIGURE 3.3. Stability of frequent values over program execution.

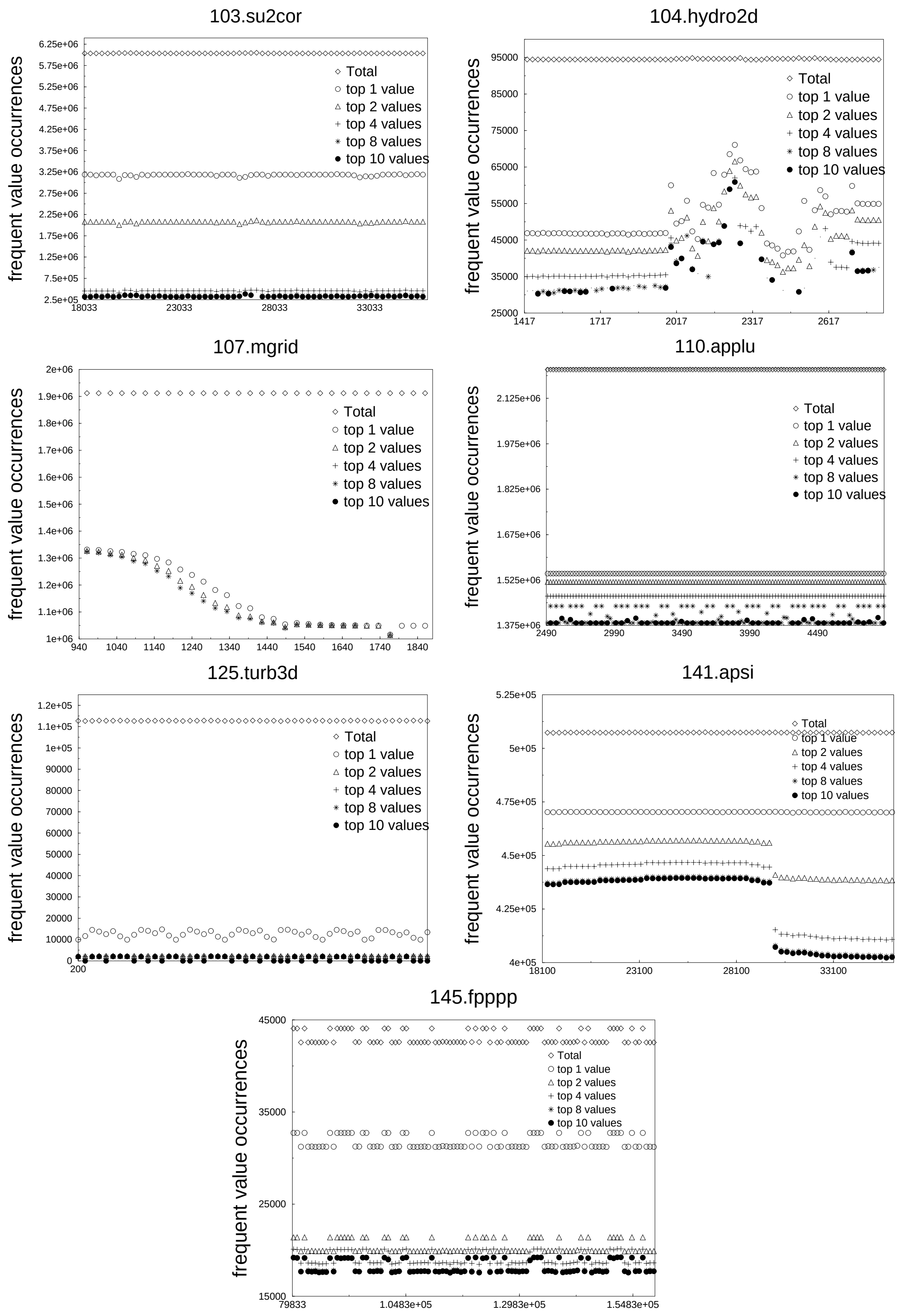


FIGURE 3.4. Stability of frequent values over program execution.

correspond to the top ten most frequently occurring values. The difference between the first (top most) and second curve is the number of locations with the top most frequent value. The difference between the first and third curve is the number of locations containing to the top two frequent values and so on. From the results it can be seen that the fraction of allocated locations occupied by a given number of frequent values remains fairly stable throughout the program execution. This is because the same values continue to occur frequently over the entire execution of the program.

### 3.1.3 Property III: Frequent Value Distribution

*The frequent values are distributed fairly uniformly throughout memory which implies that no matter which part of memory is accessed, it is likely to encounter these values.*

To establish the above property, the distribution of frequently occurring values in memory is plotted as shown in Figure 3.5 and 3.6. The data in these graphs represents the snapshot of memory at a point when the programs were nearly half way through their execution. The referenced memory was broken into blocks of 800 consecutive locations each and the percentage of frequent values in each block of 800 locations was plotted as a point in the graph. A threshold of top 8 frequent values is selected in these graphs. As can be seen, for nearly all of the programs the frequent values are scattered across the memory and for many programs the distribution of frequent values across the memory is quite uniform.

## 3.2 Frequent Value Locality vs. Value Locality

Although the frequent value locality phenomenon is related to recently discovered concepts of value locality and value prediction [40], there are important differences between them. Value locality addresses the likelihood of a previously-seen value recurring repeatedly within a storage location. The authors limited their study to only the general-purpose or floating-point registers that are targets of integer, floating-point

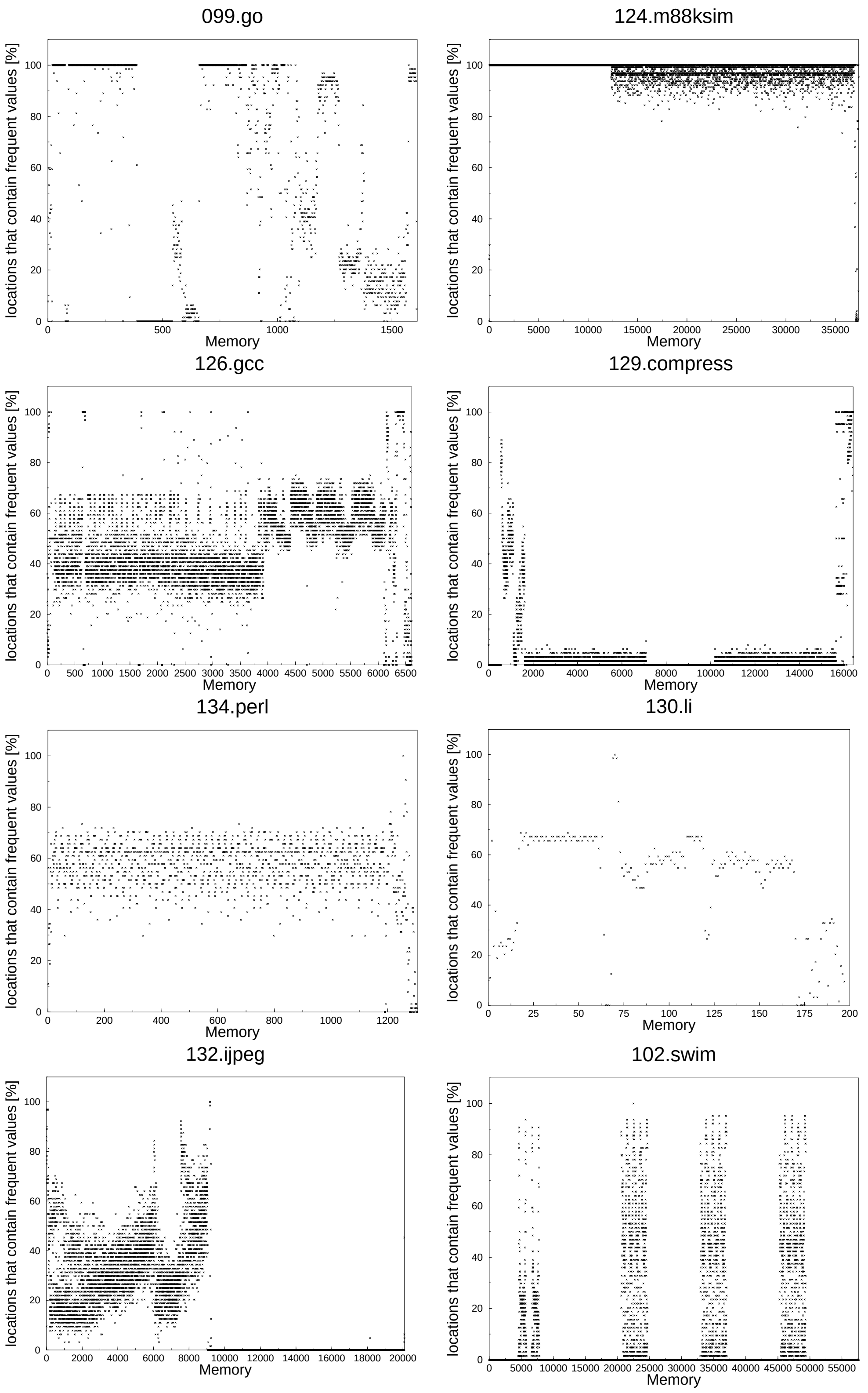


FIGURE 3.5. Distribution of frequent values across memory (heap and stack).

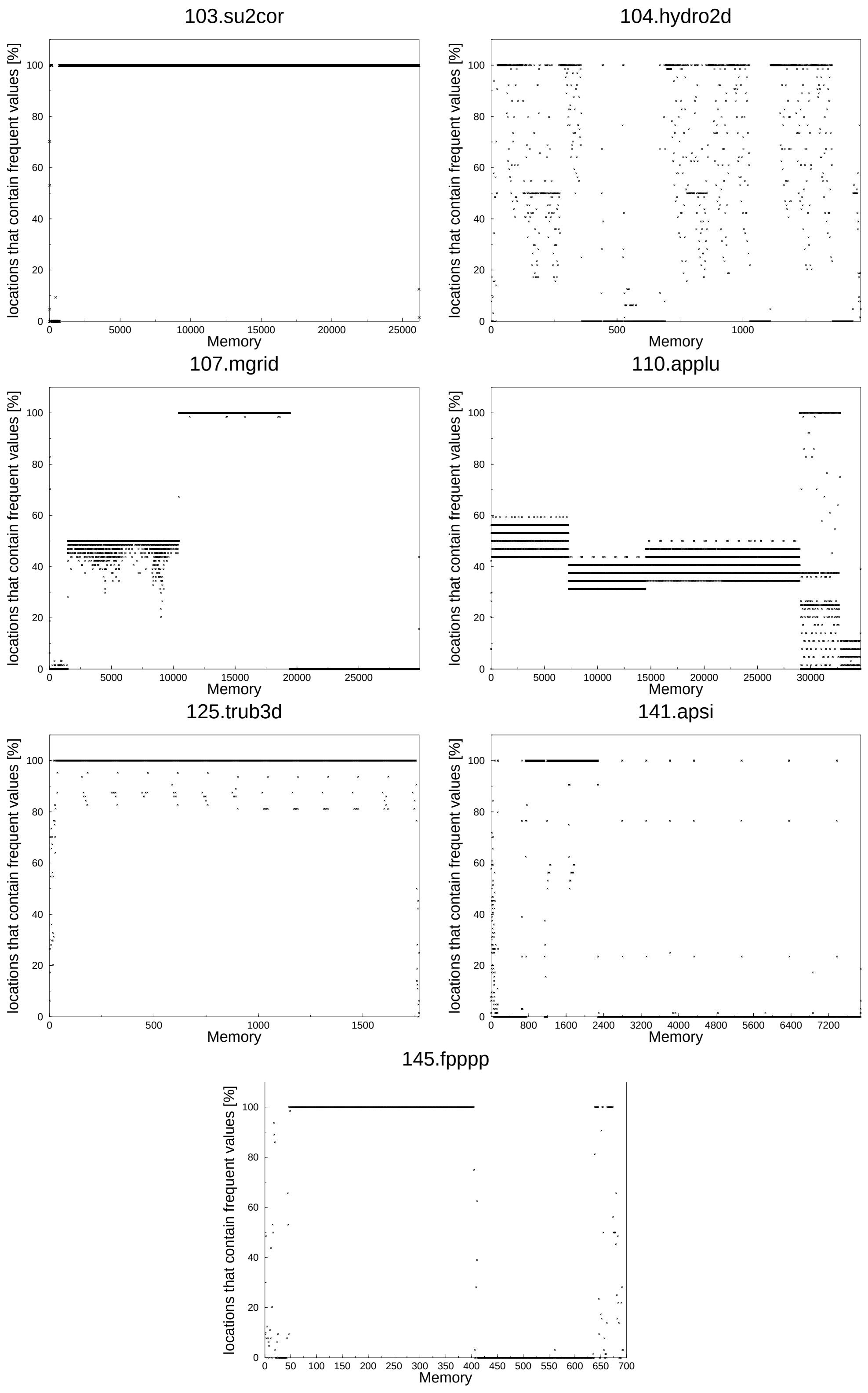


FIGURE 3.6. Distribution of frequent values across memory (heap and stack).

and memory instructions. The study is a **register** value locality and characterizes only the values encountered during multiple executions of specific instructions.

Unlike the study on only streams of instructions, our frequent value locality characterizes the behavior of values present across the **memory** allocated to the program, throughout the execution of the program. Substantial number of experiments were conducted and interesting spatial and temporal properties of frequent values were discovered. The work presents a thorough insight of frequent values that are useful in exploring memory value related applications. Register value locality is exploited for carrying out value prediction and speculative execution of instructions to speed up a program's execution. Frequent value locality can be exploited in designing the memory hierarchy to achieve better power or performance behavior.

## 3.3 Summary

In this chapter, the frequent value phenomenon has been demonstrated by characterizing it into three distinct properties. The claims are supported through elaborative experimental data obtained from running 15 `Spec95` benchmarks. The chapter first showed the existence of the frequent values across most of the programs tested. Then the stability of the frequent values during the execution time line was illustrated. Finally, the distribution of the top frequent values in the memory spaces at the mid point of execution was shown.

Though the set of ideal frequent values have been presented in this chapter, it is important to develop application driven techniques in finding those frequent values. And we will introduce those techniques in the next chapter.

# Chapter 4 Identifying Frequent Values

## 4.1 Frequent Value Identification Introduction

The data in Figure 3.1 showed that the large frequent values always vary from program to program and small frequent values can also differ across programs. Since there is no universal set of frequent values, methods for identifying these values must be developed. Before describing the different methods for finding frequent values, it is useful to understand the nature of applications that will make use of these methods.

While it has been discussed the frequent value locality in context of memory contents of a program over its entire execution, the observations have much broader implications. Given that frequent values were observed across the memory, it is also expected that these values would be frequently encountered at all points in the memory hierarchy, for example in the on chip data cache, on the data bus that brings data into the on chip cache, and of course in the main memory itself. At different points in the memory hierarchy at which frequent value locality is being exploited, different types of frequent value finding methods may be appropriate.

In this chapter a number of different approaches ranging from software profiling techniques to hardware profiling techniques that can be used to find frequent values are discussed. Different methods are suitable for different applications depending on the constraints under which the application must operate. This chapter describes the following three scenarios for finding frequent values and evaluates their effectiveness:

– *Find Once for a Given Program.* This method finds a *fixed frequent value set* through a *profiling run* which is then used by the application in all later *execution runs.* This is a purely software based approach. Thus once the values are known, they must be communicated to any hardware based application either through compiler generated code or operating system support. Moreover, if the frequent value set is sensitive to the program input, i.e., the frequent value sets obtained from different inputs differ greatly, this approach will cause deterioration in designs that employ one fixed set of frequent values.

– *Find Once Per Run of the Program.* This method finds a *fixed frequent value set* during each execution run of the program. The set of values is found through *limited online profiling* during the initial execution of the program after which the values are fixed and profiling ceases. These values are then used by the application during for remainder of the execution. In other words the *fixed frequent value set* is found during each execution and therefore the frequent value set being sensitive to program input is not a problem for this method. This approach uses *specialized hardware* for finding the values. Therefore no compiler or operating system support is required to communicate the values to the hardware.

– *Continuously Changing During Program Run.* This method maintains a *changing frequent value set* by carrying out *continuous profiling* of the program during each execution run. Moreover profiling is carried out by specialized hardware. Under this method an application can benefit from adaptation of the frequent value set during a given run.

The two low power applications that are considered later in this thesis, and the manner in which they fit into the above scenarios, are briefly described below to further motivate the need for algorithms that fit the above scenarios. As can be seen, these two applications operate at different points along the memory hierarchy.

– *Low Power Frequent Value Cache:* The design of a low power data cache is presented, which stores frequent values in encoded form to reduce the dynamic activity in the data cache. This application must use a *fixed set of frequent values* because the encoding must remain fixed for the duration of the program. This is because a change in encoding would require at a minimum flushing the cache. The situation for context switch will be discussed in Section 5.5.2. Moreover, here it is more interesting to find frequently occurring values in the data stream between the CPU and the data cache which will be referred to as the *frequently accessed values.*

– *Frequent Value Encoding for Low Power Data Bus:* The design of a bus encoding technique is described, which is aimed at reducing the switching activity on the external data bus of the CPU. This application can take advantage of a *continuously changing set of frequent values* since the encoding is localized to the data bus. In other words, no other part of the system has to be aware that encoding is being carried prior to sending a value across the data bus and decoding is performed immediately after receiving the value at the other end of the data bus. Moreover, here it is more interesting to find frequently occurring values in the data stream between the on-chip cache and off-chip memory which will be referred to as *frequently transferred values.*

Note that since the data streams relevant to the above applications flow across different points in the memory hierarchy (between CPU and on-chip data cache and between on-chip cache and off-chip memory), they correspond to values stored in a program's allocated memory. Therefore it is expected that these data streams exhibit frequent value locality. In other words the presence of *frequently accessed values* and *frequently transferred values* should be true.

Given the above applications it is clear that the first two scenarios for finding frequent values are relevant for finding frequently *accessed* values while the third scenario can be used for finding frequently *transferred* values. Therefore in the remainder of this section after describing the algorithms for finding frequent values under the three scenarios, we evaluate them in the appropriate context of frequently accessed values or frequently transferred values. All evaluations in this chapter are based upon 15 programs from the `Spec95` benchmark suite which were run on the *reference inputs*, unless stated otherwise.

## 4.2 Software Method – Find once for a given program

The method for finding frequent values under this scenario is simple but time consuming. Since this process is performed only once for a given program, it can be justified that one spends a significant amount of time on finding frequent values. I instrument the program to intercept all data values involved in load and store instructions as these are the values that constitute the data stream between the CPU and the data cache. A hash table is maintained in which all the encountered values along with their frequencies are stored. The hash table size is not allowed to grow beyond an upper limit which was 300 MB in the implementation. For some programs the table size was large enough too hold all values encountered during the execution but for others the size was not the case. When the hash table reached its limit, two-thirds of the least frequently occurring values are removed from it and then continued processing future accessed values. The values discarded have a maximum occurrence count of 200 which is less than $10^{-4}\%$ of total accesses at the time. Therefore it is highly unlikely that any frequent values would be discarded.

The results of implementing this method and applying it to `Spec95` programs are described next. Consider the data in Figure 4.1 which shows what percentage of total accesses involve *frequently accessed values* – a maximum set size of 128 values is considered during program runs based on reference inputs. As can be seen, the data stream of accessed values contains frequently occurring values on average 128 values account for over 50% of all accesses.

The data presented in Figure 4.1 is ideal data since in collecting the above data

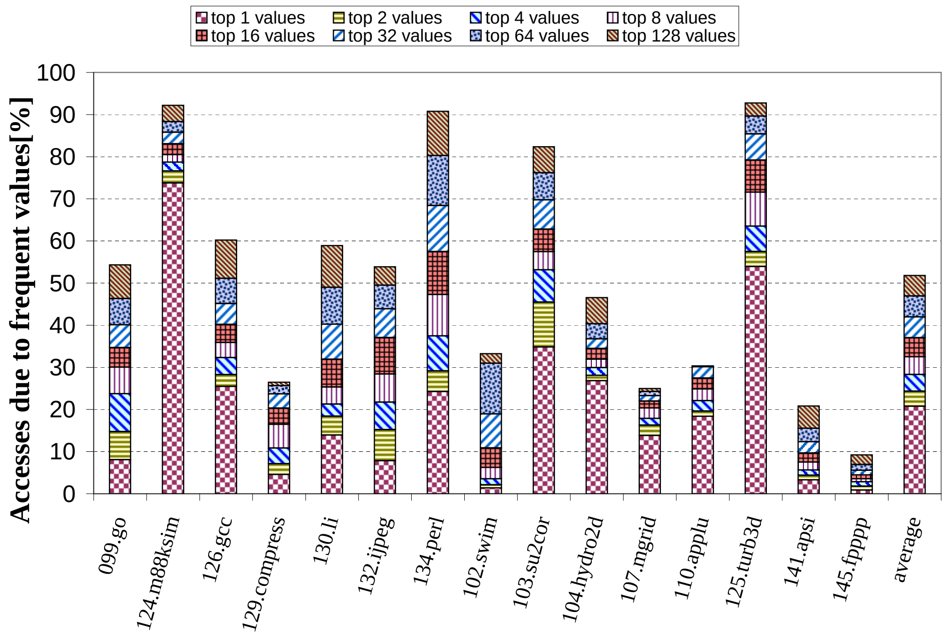


FIGURE 4.1. Access percentage attributed to top 128 frequent values.

both the profiling runs and the execution runs were carried out using the same inputs (reference inputs). Since the frequently accessed values will be found by running the program once on some input and used later during program runs on other inputs, it is interesting to see how much is lost due to the sensitivity of frequent values to program inputs. Therefore an experiment was carried out in which the profiling run on *training inputs* was used to identify frequently accessed values. Then accesses to these values were measured during program run on reference inputs. The results are shown in Figure 4.2 and 4.3. For a varying number of frequent values the accesses to frequent values as a percentage of total accesses during program run on reference inputs is plotted. One curve is based upon use of frequent values found from the profiling run on training inputs and for comparison the other ideal curve used the reference inputs during the profiling run. As can be seen, for most programs this approach does quite well as the train curve is close to the ideal curve.

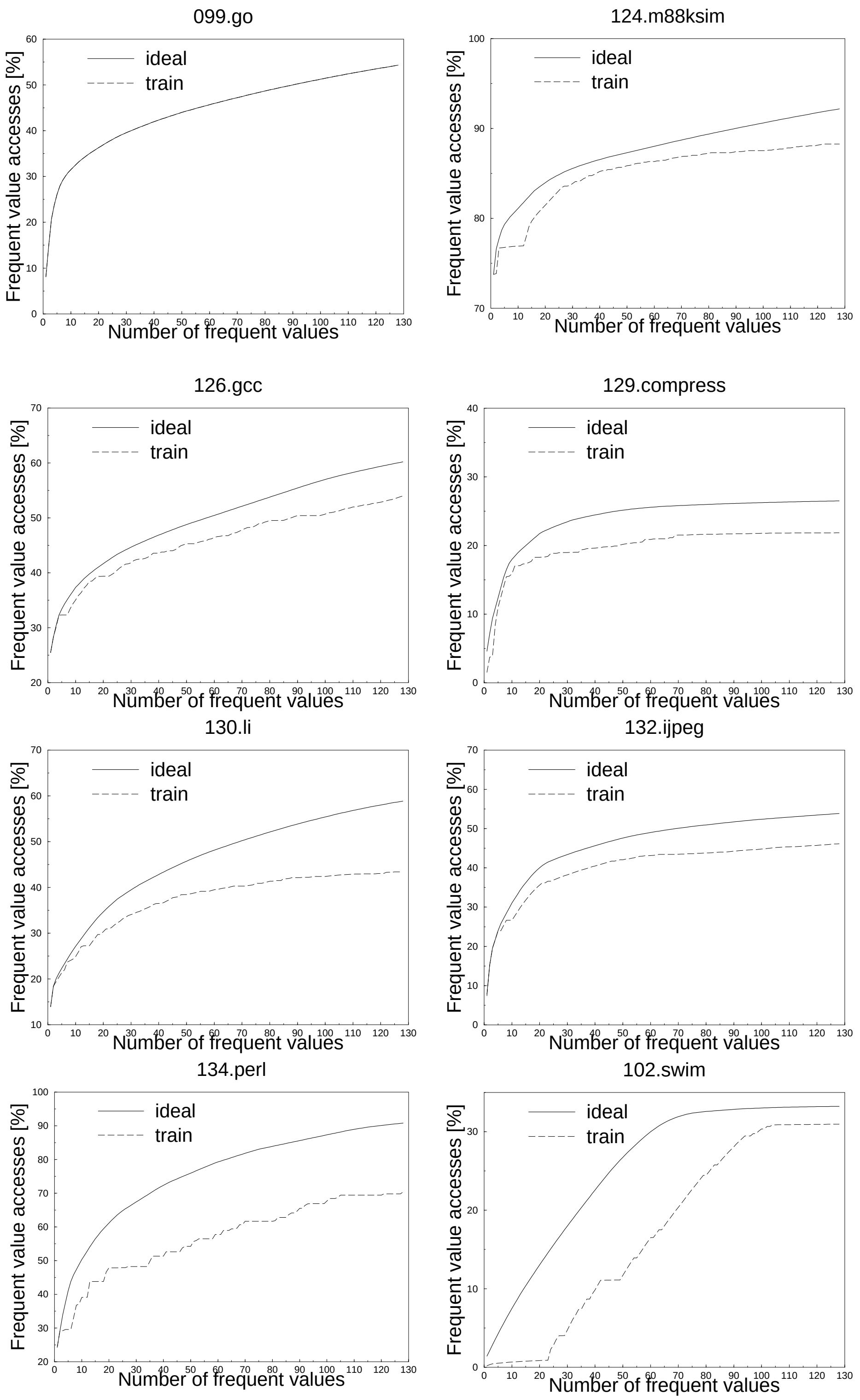


FIGURE 4.2. Finding frequent values in profiling run using training inputs for use in later execution runs. The training inputs are designed to generate feedback data in the Spec benchmarks.

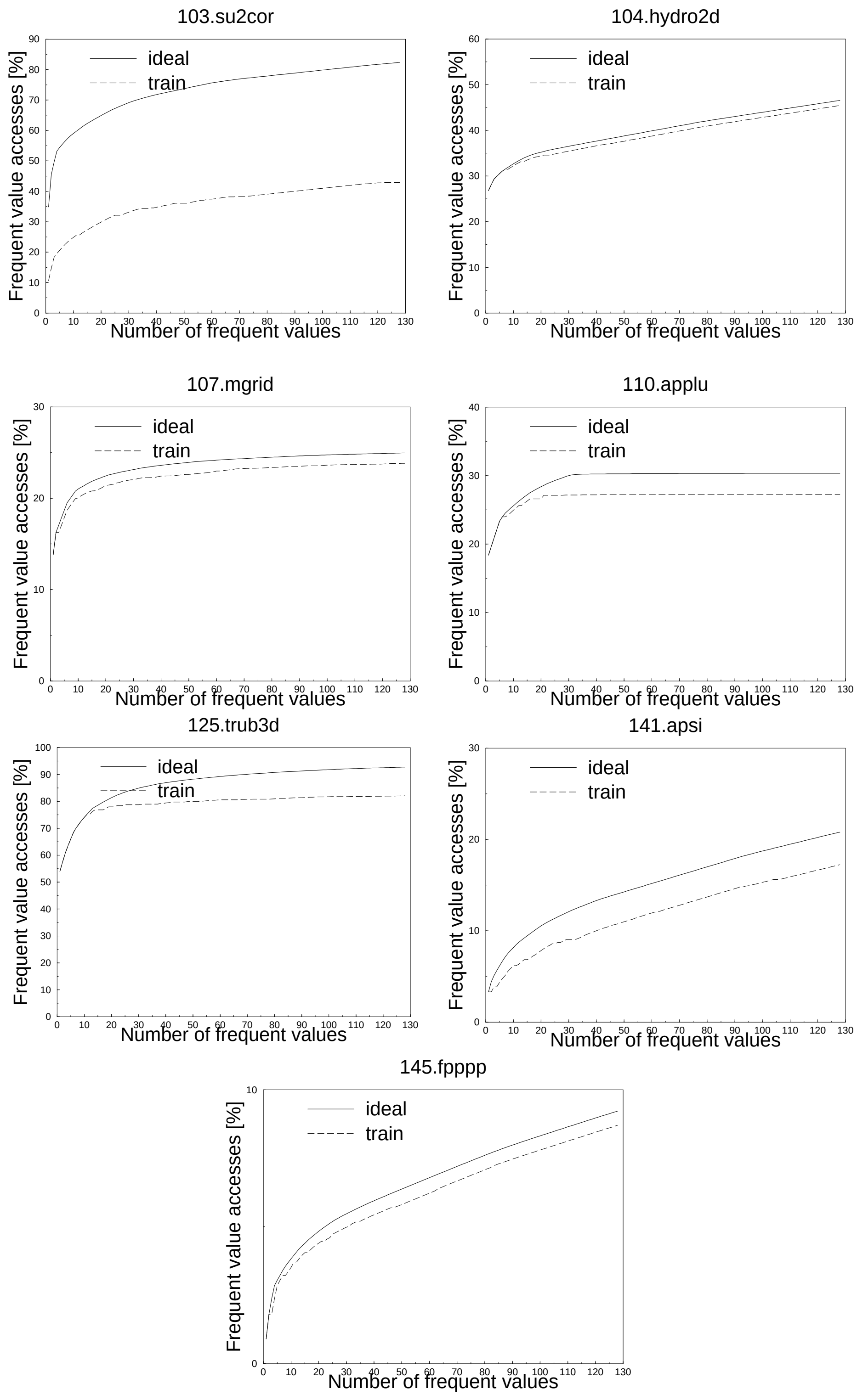


FIGURE 4.3. Finding frequent values in profiling run for use in later execution runs.

## 4.3 Hardware Method I – Find once per run of the program

As mentioned earlier, the algorithm for finding frequently accessed values during each program run is meant for implementation in hardware. Therefore I use a small table of frequent values in this method. To find the top $n$ values, the table contains $2n$ entries, each having a *value* field and a *counter* field as shown in Figure 4.4. The *value* field stores the data value encountered during monitoring and the *counter* field contains a $c$ bit saturating counter.

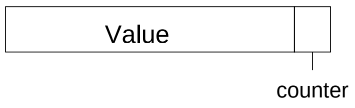


FIGURE 4.4. Value table entry.

The algorithm for finding the frequent values using the value table is given in pseudo code in Figure 4.5. Each time a data value is involved in an access by the CPU, the table values are updated as follows. If the value is already present in entry $i$, then the counter at entry $i$ is incremented by one. When the counter saturates, the entry $i$ is *swapped* with entry $i-1$. The purpose of this activity is to let frequent values gradually percolate to the top part of the table. When a new value is encountered, and there is no free entry in the table, a victim entry in the table needs to be selected to free up the space. An entry is freed from the bottom half of the table with the smallest counter value because the bottom part is expected to contain values seen less often in comparison to values in the top half of the table.

Our method is inspired by the conventional software *value profiling* technique in [12]. However, value profiling technique does not use *swapping*. It simply maintains frequency counts for values in the table and periodically clears half of the table to allow new values to enter into the table. When the half of the table is cleared, the values are sorted according to their associated frequency counts and half of the values with counts lower than the other half are removed. The sorting operation makes this

```
void UpdateValueTable($v$){
    search the table for a match of the Value field and $v$
    if (Table[$i$].Value == $v$) then{
        Table[$i$].counter++;
        if (Table[$i$].counter saturates)
            swap Table[$i$] and Table[$i - 1$]
    }
    if ($v$ is not in the table){ /* Insert $v$ into table */
        find from the lower half of the table an entry $j$ with the smallest counter
        replace table[$j$].Value with $v$
        clear table[$j$].counter
    }
}
```

FIGURE 4.5. Algorithm for finding the frequent values using the value table.

existing technique unsuitable for hardware implementation. Our algorithm does not require sorting. Instead it uses *swapping* to approximate the effect of sorting. The swapping process approximately sorts the list such that bottom half contains less frequently seen values. When replacing a value from the bottom half, the counter value is used to free up an entry corresponding to a less frequently seen value from among the values in the bottom half of the table.

The approach to approximating sorting is very effective in practice as the experiments comparing *conventional value profiling* with the proposed *hardware value profiling* show in Figure 4.6. The two algorithms are compared by comparing the quality of their frequently accessed value sets, which is expressed in terms of percentage of cache accesses that can be attributed to the values in the set. In the experiments the swapping interval is varied by varying the counter width $c$ from 1 bit to 3 bits. A longer interval means frequent values climb up in the table at a slower pace and a shorter interval leads to faster convergence but may cause excessive swapping between two entries which already contain frequent values. From Figure 4.6 it is interesting to see that a counter lengths of 1 and 2 bits give nearly the same results—on average

41% of the cache accesses are coming from top 128 frequent values captured by using either 1 bit or 2 bits counter; however, a 3 bit counter degrades a little—the top 128 frequent values account for about 36% of the cache accesses. This is because the interval between swaps is longer causing a slower pace for frequent values to move up. Therefore in the rest of the experiments, a 2 bit counter is chosen which achieves similar results as a 1 bit counter without introducing unnecessary swaps. When compared with value profiling technique [12], the proposed algorithm produces nearly the same results as conventional algorithm and in many cases performs even better (e.g., for `129.compress`, `132.ijpeg`, `124.m88ksim`, `102.swim` and `103.su2cor`).

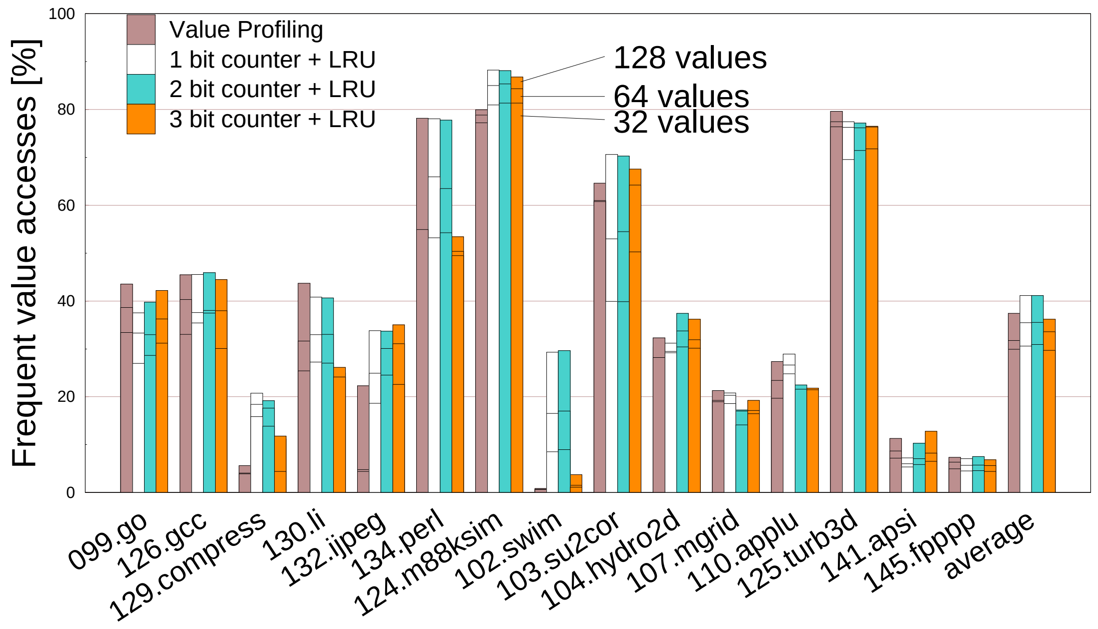


FIGURE 4.6. Comparison of value profiling technique and the proposed hardware method for capturing 32, 64 and 128 frequent values.

Next the effectiveness of this method in finding frequently accessed values is shown. The effectiveness of this algorithm depends on the degree of profiling. One can expect that a greater amount of profiling will usually be more effective. However, the more profiling the program does, the less time it has left for exploiting the frequently accessed values. Therefore in the experiments the amount of profiling was varied from 1 million to 800 million instructions for most programs of moderate size. The results are presented in Figure 4.7 and 4.8. For each benchmark a set of curves corresponding to different profiling levels specified in terms of number of instructions is presented. In parenthesis the fraction of total program execution spent on profiling is shown. Even though in the experiments I varied the profiling levels between 1 and 800 million instructions, in these plots the profiling periods displayed were selected to show interesting areas of the graph. In some cases many short profiling intervals are shown while in others more longer profiling intervals are shown.

To obtain the results, different sized frequent value tables, from 2 to 256 entries for capturing 1 to 128 values, were implemented into our architecture simulator for profiling. Every so often the values captured in each table were recorded for examination of how the frequent value set is developed as the a program runs. In the end, the quality of the frequent values recorded at each stage is measured in terms of the percentage of cache accesses they contribute. Those data were then presented in graphic forms as shown in Figure 4.7 and 4.8.

The results imply that the programs can be divided into two categories. For many of the programs the degree of profiling makes only a small difference (e.g., `124.m88ksim`). In other words the frequently accessed values can be identified using a small amount of profiling and greater amounts of profiling are not necessary. The reason for this behavior is that usually a very small subset of frequently accessed values account for most of the frequent value accesses and these values are so frequent that they are seen immediately as execution begins. For example, looking at Figure 4.1, it can be seen that in the case of `124.m88ksim` the top 128 values account for 92% of all accesses; however, the topmost value alone accounts for 74% of all accesses. For other programs increasing the profiling interval beyond a certain threshold makes a significant difference (e.g., `129.compress`). This is because for these programs typically a larger number of frequent values need to identified accurately because they

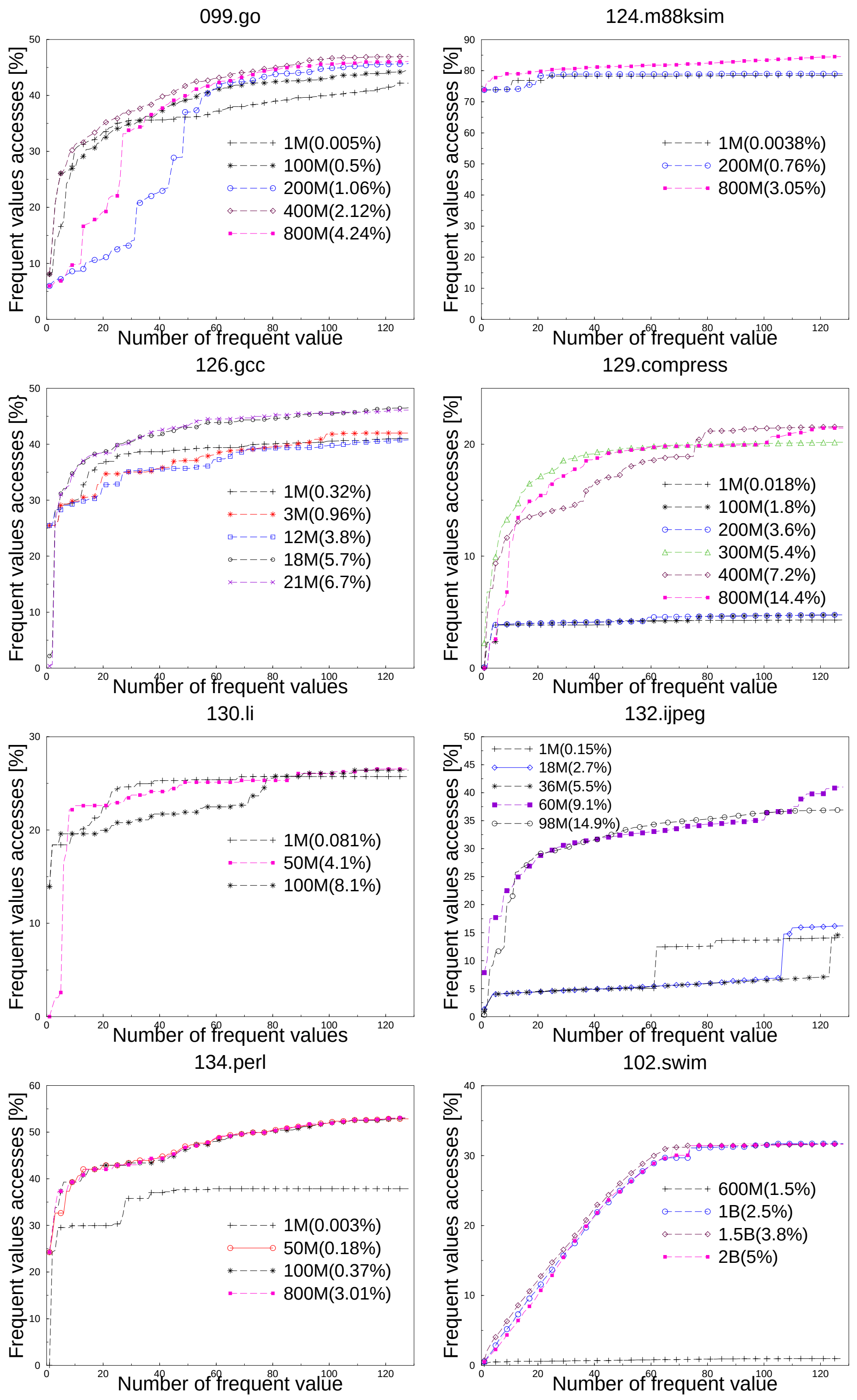


FIGURE 4.7. Finding and using frequent values in each execution run.

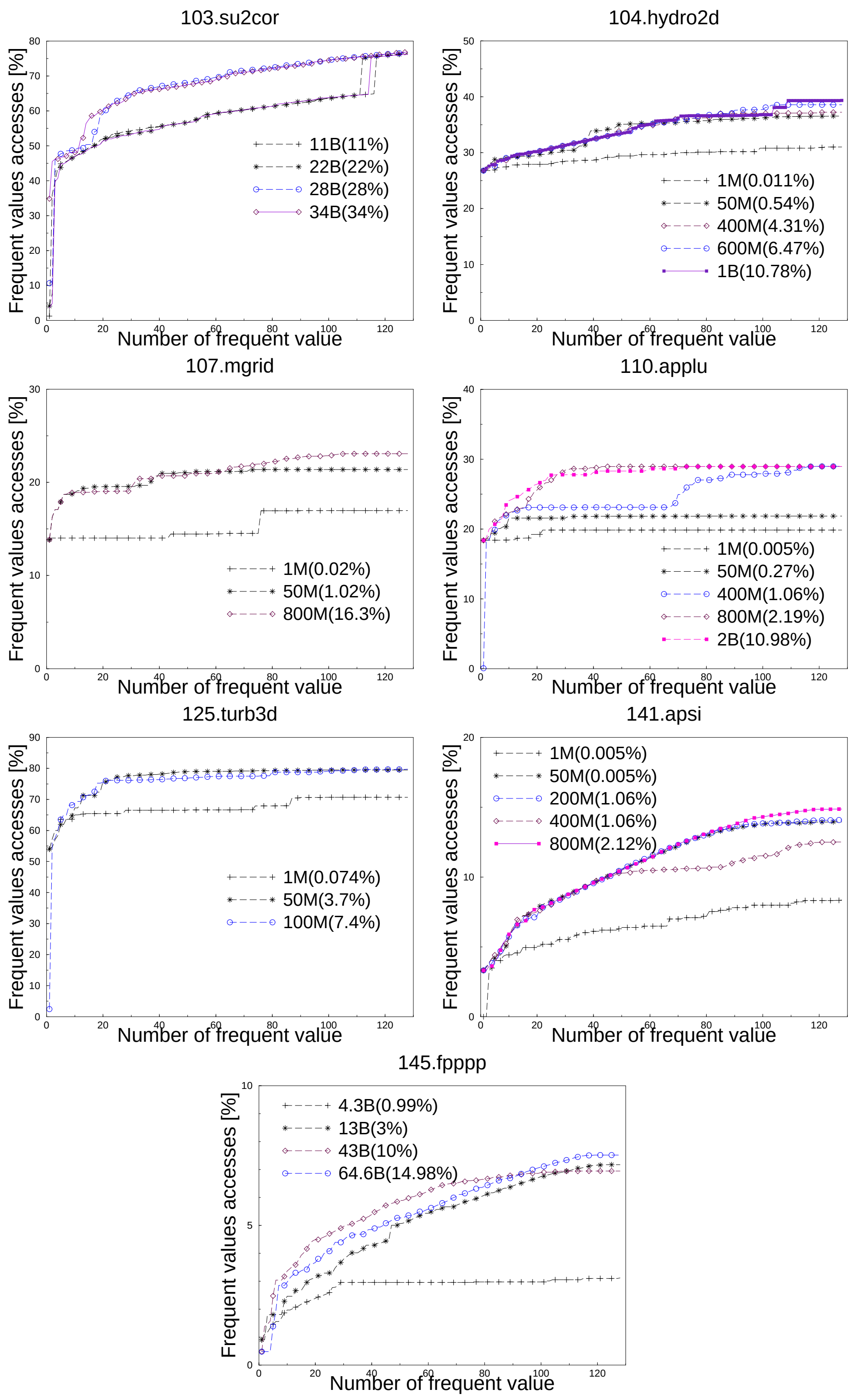


FIGURE 4.8. Finding and using frequent values in each execution run.

all account for a significant number of accesses. The larger the number of important frequent values, the longer it may take to find them as some of these values may show up a bit later in the execution. For example, Figure 4.1 shows that in the case of `129.compress`, the top 128 values account for nearly 27% of all accesses, but the top most value accounts for only 5% of accesses. In fact, to get close to 27% of accesses it is important to accurately identify the top 32 frequently accessed values for `129.compress`.

## 4.4 Hardware Method II – Adaptive FV Finder

Let us now consider the hardware algorithm for maintaining a continuously changing set of frequent values. A table with as many entries as the number of frequent values that are to be identified is maintained. The LRU replacement policy is used for filling and updating the frequent value table. To gain time ordering information, a *reference bit* and a $n$-bit timestamp for each value recorded in the FV Finder is used. The reference bit is set when the value appears at the input. At regular intervals, the reference bit is shifted right into the high-order bit position of the $n$-bit timestamp causing all bits in the timestamp also to be shifted right and the lowest-order bit in the timestamp being discarded. This operation is performed for all entries in the table and at the same time all the reference bits are reset. Thus, the timestamp keeps the history of value occurrences for the last $n$ time periods. For example, the timestamp of 000 means this value did not appear during the last three time intervals, timestamp 100 means it was just seen in the last interval, and the timestamp 000 with reference bit set means it is encountered in the current time slot. When an entry is required and a value is to be evicted, the entry that is selected is the one with the smallest timestamp and clear reference bit. The new value is put in with a fresh reference bit and timestamp (all 0's) in this selected entry. Figure 4.9 shows the algorithm in pseudo code for the hardware, assuming that there are $N$ entries for the reference bit array, $ref[]$, and the timestamp array, $ts[]$. Figure 4.10 is the pseudo code for replacing an old value with a new value in the FV Finder.

```
void UpdateFVFinderStatus() {
    clock_tick −−;
    if (clock_tick == 0) {
        for ($i = 0$; $i < N$; $i + +$) {
            $ts[i] = ts[i] >> 1$;
            the highest-order bit of $ts[i] = ref[i]$;
        }
        clock_tick = update_period;
    }
}
```

FIGURE 4.9. LRU algorithm for the adaptive frequent value finder.

Figure 4.11 gives an example to illustrate the above algorithm using a sequence of data values shown in the table labeled with `Time` and `Value`. The `Value` row shows the sequence of values coming at clock time indicated by the `Time` row. For simplicity, it is assumed that there are only 4 entries in the coder (FV Finder) each having a 3-bit timestamp and a 1-bit reference bit. Initially the coder is empty. After $t7$, the contents of the coder along with the reference bit and timestamp are shown in (a). Suppose that the period of the LRU updating is 8 clock ticks. Therefore, at time $t8$, the timestamps and the reference bits need to be updated. The results are shown in (b). At $t9$, a new value $0x40457f80$ comes in, but all the entries in the coder are filled. A victim needs to be selected and replaced by the new value. (b) shows that the value $-1$ has the smallest timestamp 000 and a clear reference bit, therefore it is chosen as the victim. After the replacement, the new value is inserted into the 2nd entry and its timestamp is cleared and the reference bit is set as shown in (c). At $t10$, another new value 7 comes in replacing the value $0xae2$ because it has the smallest timestamp and its reference bit is 0. The resulting coder, timestamp and reference bit are shown in (d). The values in the frequent value table, together with the timestamp, give an idea on what values most recently occurred and therefore might be seen again soon.

```
void Insert($v$) {
    if ( there are still empty entry #$e$ in the table ) {
        insert $v$ to the empty entry $e$;
        $ts[e] = 000$;
        $ref[e] = 1$;
        return;
    }
    /* Find the entry with the smallest timestamp and clear reference bit */
    min = 1111;
    ind = -1;
    for ($i = 0$; $i < N$; $i++$) {
        temp = $ref[i]$ concatinated with $ts[i]$;
        if ( temp < min ) {
            min = temp;
            ind = $i$;
        }
    }
    insert $v$ into the table entry at index ind;
    $ts$[ind] = 000;
    $ref$[ind] = 1;
}
```

FIGURE 4.10. Inserting new values. Assume a 3-bit timestamp and 1 reference bit.

| Time: | t1 | t2 | t3 | t4 | t5 | t6 | t7 | t8 | t9 | t10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Value: | 0 | −1 | 1 | 0xae2 | 0 | 0 | 0 | 1 | 0x40457f80 | 7 |

| Coder | ref. | ts |
|---|---|---|
| 0 | 1 | 101 |
| −1 | 0 | 001 |
| 1 | 1 | 010 |
| 0xae2 | 0 | 010 |

(a)

| Coder | ref. | ts |
|---|---|---|
| 0 | 0 | 110 |
| −1 | 0 | 000 |
| 1 | 0 | 101 |
| 0xae2 | 0 | 001 |

(b)

| Coder | ref. | ts |
|---|---|---|
| 0 | 0 | 110 |
| 0x40457f80 | 1 | 000 |
| 1 | 0 | 101 |
| 0xae2 | 0 | 001 |

(c)

| Coder | ref. | ts |
|---|---|---|
| 0 | 0 | 110 |
| 0x40457f80 | 1 | 000 |
| 1 | 0 | 101 |
| 7 | 1 | 000 |

(d)

(a) The contents of coder, reference bit(ref) and timestamp(ts) at t8.

(b) Updating ref. and ts: Shift ref. right to the highest order bit of ts, shifting rest of ts right by one bit.

(c) At t9, value −1 is replaced by new value 0x40457f80.

(d) At t10, value 0xae2 is replaced by new value 7.

FIGURE 4.11. Example of frequent value identification.

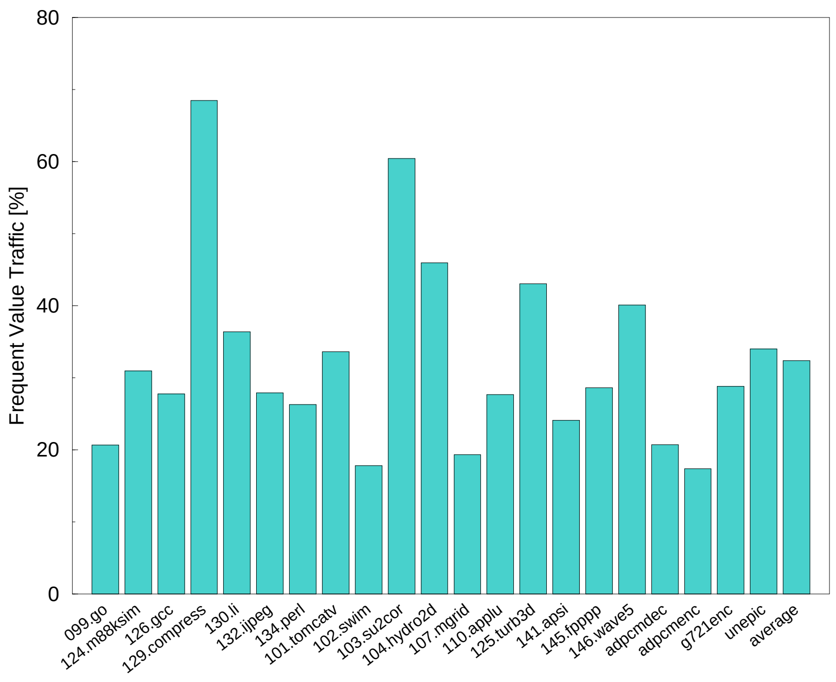


FIGURE 4.12. Data bus traffic due to 32 frequent values.

Since the above approach is used for the bus encoding application, I evaluated it in context of the data stream between on-chip cache and off-chip memory. I measured percentage of data traffic that could be attributed to the changing set of 32 frequently transferred values found using the above algorithm. Figure 4.12 shows the results of this experiment. On average, 32% of the traffic was attributed to the frequently transferred values. It is interesting to note that in the case of the `129.compress` benchmark, when a fixed set of frequently accessed values was used, they did not account for a substantial number of accesses; but when a changing set of frequently transferred values are used, they account for nearly 68% of the total traffic. Therefore, while for some benchmarks a fixed set of frequent values may be adequate, for others a changing set may provide better results.

## 4.5 Summary

In this chapter, one software and two hardware techniques for identifying frequent values efficiently have been introduced. The software technique is suitable for applications whose frequent values are not sensitive to program inputs. Therefore, finding them incurs a one-time overhead and future runs can benefit from them easily. The hardware techniques are transparent to the user and adapt to different programs.

Next, I will discuss one of the major applications that is based on the frequent value phenomenon in level one on-chip data caches. A novel energy efficient design that explores the frequent values inside the cache will be introduced.

# Chapter 5
# The Frequent Value Data Cache Design

On-chip cache memory consumes a significant amount of energy, which many researchers have paid attention to. A variety of cache designs are being developed to conserve energy [2, 13, 24, 36, 37, 49, 51, 74].

In this chapter a new approach is presented to reduce dynamic energy consumed by a data cache which is complimentary to many of the previously proposed solutions for the same problem. It presents the design and evaluation of an energy efficient L1 data cache, called the *frequent value cache* (FVC), which achieves energy savings through data compression. In the previous chapters, it has been shown that a small number of distinct frequent values often occupy a large portion of program memory and therefore account for a large portion of memory accesses. This chapter demonstrate how this *frequent value* phenomenon can be exploited in designing a cache that trades off performance with energy efficiency.

In the FVC, frequent values are represented by fewer than 32 bits as they are stored in encoded form while all other values are stored in unencoded form using all 32 bits of a word. The data array is partitioned into two arrays such that if a frequent value is accessed only the first shorter data array is accessed, while for nonfrequent values both data arrays must be accessed. Since frequent values are encountered quite often, this approach greatly reduces the energy consumed by the data cache. The reduction in energy is achieved at the cost of an additional cycle needed to access nonfrequent values. Therefore, FVC design represents a trade-off between lower dynamic energy consumption for frequent value accesses and higher access times for nonfrequent value accesses.

The detailed access time and energy models for FVC have been developed by modifying XCACTI [31]. The studies demonstrate that for a wide range of configurations for set-associative data caches – including varying associativity, cache size, line size, and number of frequent values – the FVC compares favorably with a conventional data cache design. The time spent on encoding and decoding of values does not impact the cache access time for associative caches.

Prior works propose other cache designs that exploit presence of frequent values [76, 69]. However, none of those designs are low power cache designs but rather they are aimed at improving the hit rate of the data cache. By storing data in compressed form greater amounts of data is held in the caches to increase the hit rate. The design in [76] provides an extra small regularly structured cache that stores the frequent values. In [69], the data cache stores compressed lines so that the effective capacity of the data cache is increased and the miss rate is also reduced. Both of the above designs increase the *cache cycle time*. In this chapter, the new design does not increase the *cache cycle time*. By restructuring the data array to take advantage of frequent values the *average* energy consumption for every cache access is reduced. Unlike the other designs, the miss rate from the new FVC stays the same as the original one. In [63] a cache energy reduction technique based upon encoding only zero value was introduced. Since FVC design considers a set of values for encoding, the amount of energy reduction achieved is significantly higher.

This chapter is organized as follows. Section 5.1 presents the FVC design. The FVC's access times are analyzed in Section 5.2 and energy consumption in Section 5.3. In Section 5.4, frequent value identification is discussed. Section 5.5 contains experimental results.

## 5.1 FV Cache Overview

From the perspective of the frequent value cache, data values are divided into two categories: a small number of *frequent values*, say $n$, that typically ranges from 4 values to 128 values, and all remaining values that are referred to as *nonfrequent values*. The frequent values are stored in encoded form and therefore can be represented in $log_2n$ number of bits, which ranges from 2 bits for 4 frequent values to 7 bits for 128 frequent values. The nonfrequent values are stored in unencoded form in 32 bit words. The set of frequent values remains fixed for a given program run.

The cache data array is partitioned into two data arrays as shown in Figure 5.1. The *low-bit array* contains the lower order $log_2n$ bits of each word and the *high-bit array* contains the remaining $32 - log_2n$ bits. Frequent values are stored in encoded form in the low-bit array while nonfrequent values are split into the lower order $log_2n$ bits and the higher order $32-log_2n$ bits using the space in both arrays. To distinguish between a code for a frequent value and a trailing part for a nonfrequent value in the low-bit array, an additional flag bit is needed corresponding to each word in a cache line. The extra bit was stored along with every word in the low-bit array so that the word width becomes $log_2n + 1$. The original frequent values are stored in an n-entry decoder indexed by the $log_2n$-bit frequent value code.

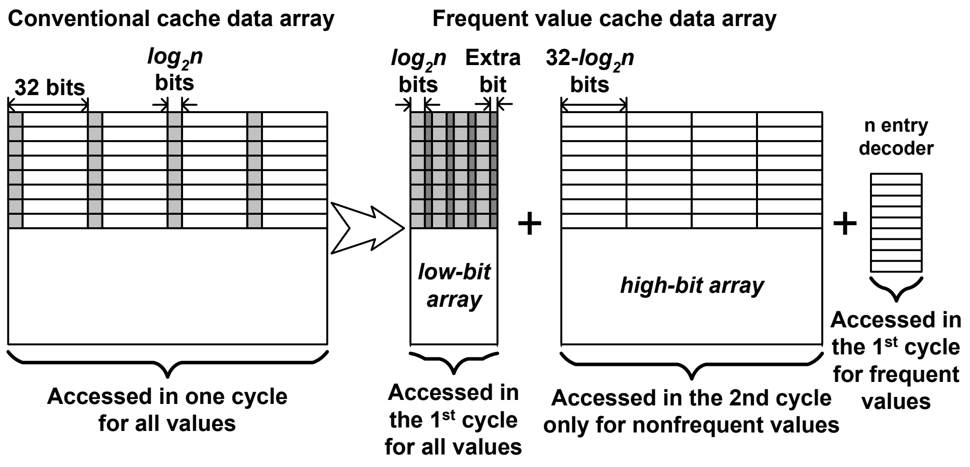

Conventional cache data array
Frequent value cache data array
32 bits
log2n bits
log2n bits
Extra bit
32-log2n bits
n entry decoder
low-bit array
high-bit array
Accessed in one cycle for all values
Accessed in the 1st cycle for all values
Accessed in the 2nd cycle only for nonfrequent values
Accessed in the 1st cycle for frequent values

FIGURE 5.1. An example of partitioning the data array of 32 bits per word (bpw), 4 words per line (wpl) into two arrays of $log_2n + 1$ bpw, 4 wpl and $32 - log_2n$ bpw, 4 wpl each. The n-entry decoder is used to store the original frequent values.

### 5.1.1 Accessing the FVC

When reading a word from the cache, initially it simply reads from the low-bit array. Since every word read out contains a flag bit, it is first examined to determine what comes next. If the flag is clear it means that the desired word is in unencoded form so the remaining bits should be read out from the high-bit array to form the original value. On the other hand, if the flag is set it means that the desired word is a frequent value and it is stored in encoded form. In this case it proceeds to decode the value. Since the access to the high-bit array is avoided, cache activity is significantly reduced.

Since the retrieval of $log_2 n$ bits from the low-bit array and that of $32 - log_2 n$ bits from the high-bit data array is serialized, it takes longer to read a nonfrequent value from the FVC than it would have taken to read the same value from a conventional data cache. Let us assume that on a hit it takes a single cycle to read a value from a conventional data cache. In contrast, for FVC a frequent value is read in one cycle while a nonfrequent value is read in two cycles. In other words, in the first cycle the $log_2 n$ bits from the low-bit array are accessed and if the value is a nonfrequent one, in the second cycle the remaining bits from the high-bit data array are accessed. Under this scenario reading nonfrequent values takes twice as long as that for frequent values. However, this is the worst case scenario as accessing most caches takes multiple cycles and consequently reading nonfrequent values does not have to double the total access time.

A write to the FVC is performed in a similar manner. Before a value is written, it is first encoded through an encoder. If encoding is successful, it means that the value is a frequent value and thus $log_2 n$ bit code is stored in the low-bit array and the flag bit is set. In this case, accessing the high-bit array is avoided. If the encoding fails, the value to be written is a nonfrequent value and thus both low-bit and high-bit data arrays are accessed as well as the flag bit is cleared.

### 5.1.2 The FVC design

Figure 5.2 shows a detailed FVC design. Instead of one data array in a conventional cache, there are two data arrays. Let us discuss in detail how the read/write operations as well as corresponding decode/encode operations are performed.

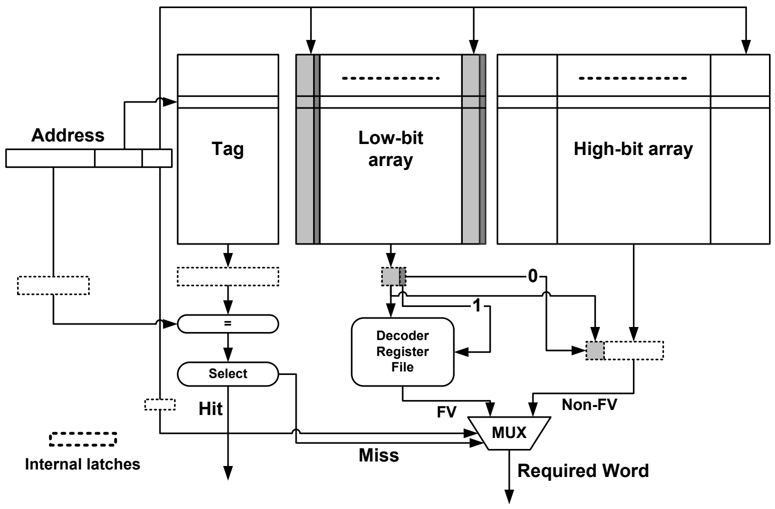


FIGURE 5.2. The FVC design.

During a *read operation* the low-bit array is first read out and the flag bit determines the next step. When the flag bit is set, the $log_2 n$ code is decoded. The decoder is in effect a multi-ported register file that contains frequent values. This *decoder register file* is shared among the multiple cache ways. The code is used to index the register file to retrieve the corresponding 32-bit frequent value. Reading a frequent value is done after the 32-bit flows through the output multiplexer. However, if the flag bit is clear (indicating a nonfrequent value), it turns off the decoder letting the partial data value flow to an internal latch where the remaining part of the value is filled from reading the high-bit array in the second cycle. The full value is obtained by concatenating the two parts completing an infrequent value read.

In a traditional cache implementation the entire line is first read out from the data array, then the desired word within the line is selected at the time it reaches the output multiplexer. If the same scheme is used in the FVC design, the decoding of frequent values cannot begin until the required word is selected out which is the very end of a cache access. Consequently decoding will increase the cache access time which is not desirable. Therefore the FVC adopts the subbanking scheme proposed by [24] in which the subbank containing the target word can be read independently. The width of each subbank is the physical word width and each subbank can be activated independently. This design facilitates the FVC implementation in that the decoding for frequent values can begin immediately after accessing the low-bit array since only the word at the desired location is read out instead of the whole line. The actual measurements of FVC access time can be found in section 5.2 and the subbanking scheme is used in the experiments as the baseline configuration.

During a *write operation* the value must be encoded. The encoding of a frequent value to be written is carried out before the cache access since the value to be written may be known as early as the decode/operand fetch stage. The encoding hardware for write accesses is shown in Figure 5.3. It is a CAM [1] that can match an incoming frequent value and output its location in binary form. The "CAM Array" shown in the figure stores the most often encountered 16 frequent values and the "RAM" stores the binary representation (ID) of each entry corresponding to the "CAM Array", that is, for 16 frequent values the ID needs 4 bits each.

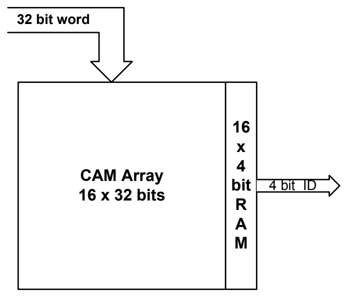


FIGURE 5.3. 16 Value FV CAM.

[1]CAM—Content Addressable Memory. A memory structure that can output a value's address based on the input value.

The frequent value encoder shown in Figure 5.3 is part of a slightly larger hardware unit that contains extra control information that is responsible for *finding frequent values*. During a given program run, frequent values are first found through this hardware by monitoring the values accessed for a certain amount of time after which the values are fixed and monitoring is stopped. For the remainder of the program's execution, these fixed frequent values are stored in encoded form. At this point, the values captured by the frequent value finder are sent to the *decoder register file*. When frequent value encoding/decoding is being carried out the contents of the *decoder register file* and the *encoding CAM* are the same. The details of the frequent value finder are discussed in Section 5.4.

The XCACTI [31] has been modified to incorporate a model of the above FVC design. The XCACTI tool adds the energy model for cache writes, write-backs and line-fills. This is what is needed in order to correctly measure the FVC energy. The baseline cache uses the subbanking technique presented in [24]. The major change in XCACTI to accommodate subbanking is the organization parameter $N_{dwl}$ which indicates how many times the data array has been split with vertical cut lines. In the FVC model $N_{dwl}$ is fixed to the number of words per line. This way the address decoder to the data array will automatically pick the right subbank to drive. For FVC design, accessing a frequent value differs from a baseline access in that the word width is narrowed down to code size plus one. Accessing an infrequent value increases each word width by one. Those will affect the delay in decoder, wordline, bitline, etc., since the number of bit columns, the wire lengths connecting different lines in a set and other parameters changed. For the *frequent value register file*, I extracted the data array access model from XCACTI and modified it so that it complies with register files. The method used is similar to the one used in [10]. I used those models to compare the access time and energy behavior of the FVC for a range of configurations for a conventional cache and FVC – including varying associativity, cache size, line size, and the number of frequent values. Next some details of this study are presented to demonstrate the feasibility and effectiveness of the FVC design. All of the data presented in this chapter is for 0.18 $\mu m$ technology.

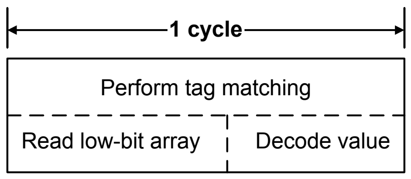
## 5.2 FVC Access Time

Let us study the timing behavior of an FVC and a conventional cache in greater detail and then see how their access times compare. In a conventional cache design the decoder drives both the tag array and the data array in parallel. The Tag array does the tag matching and the data array reads out the words from the intended locations. Typically, the tag matching takes longer than the data read. If the tag matching fails, the results of the data read are discarded; otherwise the data word is sent to the CPU. In case of FVC, while the tag is compared against the reference address tag, the data is read out of the low-bit data array. The single bit indicating whether the value is frequent or nonfrequent is also read and if the bit indicates that it is an encoded frequent value, the decoding of the value is then initiated. Therefore, since the read is completed before the results of the tag match are known, the decoding operation is carried out in parallel with tag matching.

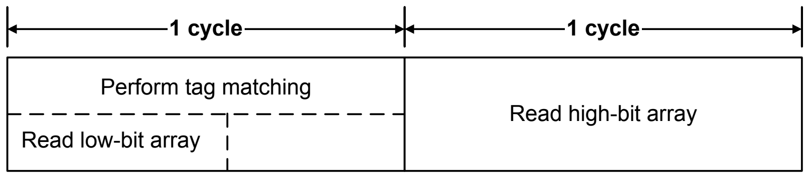
It should be ensured that the access time of the FVC is no greater than the access time of the baseline cache. Since the tag path is not changed, the tag matching time is the same for baseline and FVC caches. Therefore, in order for the access time of the FVC to be equal to that of the conventional cache, the sum of the times spent by the FVC on retrieving a frequent value from the low-bit data array ($T_{data\_path\_delay}$ without output multiplexer) and the decoding of the frequent value ($T_{decode}$) should be no greater than the time spent tag matching ($T_{tag\_path\_delay}$), that is:

$$T_{data\_path\_delay} + T_{decode} \leq T_{tag\_path\_delay}.$$

Figure 5.4(a) shows the timing behavior of the FVC for frequent and nonfrequent value reads assuming that the FVC satisfies the above constraint. It should be noted that no timing constraint arises due to encoding of values during write operations because the value being written is encoded before the cache access. Figure 5.4(b) depicts the situation for nonfrequent value reads. Since accessing the high-bit array is serialized with the low-bit array access, the total time in reading the data array can not be less than the tag matching time. Therefore it is carried in the following cycle.



(a) Frequent value read access



(b) Non frequent value read access

FIGURE 5.4. Read access operation.

Whether or not the above timing constraint is satisfied is determined by the configuration of the FVC. The $T_{decode}$ is a function of decoder size which in turn depends upon the number of frequent values used—the larger the number the longer the decoding time. The difference between the $T_{tag\_path\_delay}$ and the $T_{data\_path\_delay}$ is a function of cache size and associativity.

Table 5.1 shows the increase in access time, if any, that is caused due to the need to perform a decode operation upon a frequent value access. Recall that the access time is the critical path delay in the cache. The first number is the tag matching time while the second number is the increase in access time for the FVC design. In this study, I limit the variation of the cache organization parameters so that the cache data array is not chopped up into too many subarrays since it has already been

<u>Access times in nanoseconds.</u>

8 Kbyte Cache

| Ways → | 1 | | 2 | | 4 | | 8 | |
|---|---|---|---|---|---|---|---|---|
| L → | 8b | 16b | 8b | 16b | 8b | 16b | 8b | 16b |
| C (bits) ↓ | | | | | | | | |
| 2 | **4.38+0** | *2.43+0.01* | **2.85+0** | **2.26+0** | **2.36+0** | **2.11+0** | **2.22+0** | **1.99+0** |
| 3 | *4.38+0.01* | *2.43+0.08* | **2.85+0** | **2.26+0** | **2.36+0** | **2.11+0** | **2.22+0** | **1.99+0** |
| 4 | *4.38+0.07* | *2.43+0.15* | **2.85+0** | **2.26+0** | **2.36+0** | **2.11+0** | **2.22+0** | **1.99+0** |
| 5 | *4.38+0.14* | *2.43+0.21* | **2.85+0** | **2.26+0** | **2.36+0** | **2.11+0** | **2.22+0** | **1.99+0** |
| 6 | *4.38+0.24* | *2.43+0.31* | *2.85+0.03* | *2.26+0.06* | **2.36+0** | **2.11+0** | **2.22+0** | **1.99+0** |
| 7 | *4.38+0.32* | *2.43+0.40* | *2.85+0.11* | *2.26+0.14* | *2.36+0.04* | *2.11+0.04* | **2.22+0** | **1.99+0** |

16 Kbyte Cache

| Ways → | 1 | | 2 | | 4 | | 8 | |
|---|---|---|---|---|---|---|---|---|
| L → | 8b | 16b | 8b | 16b | 8b | 16b | 8b | 16b |
| C (bits) ↓ | | | | | | | | |
| 2 | **8.89+0** | *4.37+0.02* | **4.80+0** | **2.86+0** | **2.94+0** | **2.37+0** | **2.52+0** | **2.28+0** |
| 3 | *8.89+0.01* | *4.37+0.09* | **4.80+0** | **2.86+0** | **2.94+0** | **2.37+0** | **2.52+0** | **2.28+0** |
| 4 | *8.89+0.09* | *4.37+0.16* | **4.80+0** | **2.86+0** | **2.94+0** | **2.37+0** | **2.52+0** | **2.28+0** |
| 5 | *8.89+0.15* | *4.37+0.22* | **4.80+0** | **2.86+0** | **2.94+0** | **2.37+0** | **2.52+0** | **2.28+0** |
| 6 | *8.89+0.25* | *4.37+0.32* | *4.80+0.02* | *2.86+0.09* | *2.94+0.01* | *2.37+0.02* | **2.52+0** | **2.28+0** |
| 7 | *8.89+0.33* | *4.37+0.41* | *4.80+0.10* | *2.86+0.17* | *2.94+0.09* | *2.37+0.10* | **2.52+0** | **2.28+0** |

32 Kbyte Cache

| Ways → | 1 | | 2 | | 4 | | 8 | |
|---|---|---|---|---|---|---|---|---|
| L → | 16b | 32b | 16b | 32b | 16b | 32b | 16b | 32b |
| C (bits) ↓ | | | | | | | | |
| 2 | *8.89+0.01* | *4.37+0.09* | **4.79+0** | **2.87+0** | **2.96+0** | **2.43+0** | **2.56+0** | **2.41+0** |
| 3 | *8.89+0.08* | *4.37+0.16* | **4.79+0** | **2.87+0** | **2.96+0** | **2.43+0** | **2.56+0** | **2.41+0** |
| 4 | *8.89+0.16* | *4.37+0.24* | **4.79+0** | **2.87+0** | **2.96+0** | **2.43+0** | **2.56+0** | **2.41+0** |
| 5 | *8.89+0.22* | *4.37+0.30* | *4.79+0.01* | *2.87+0.06* | **2.96+0** | **2.43+0** | **2.56+0** | **2.41+0** |
| 6 | *8.89+0.32* | *4.37+0.40* | *4.79+0.11* | *2.87+0.16* | *2.96+0.06* | *2.43+0.04* | **2.56+0** | **2.41+0** |
| 7 | *8.89+0.41* | *4.37+0.49* | *4.79+0.19* | *2.87+0.25* | *2.96+0.14* | *2.43+0.13* | **2.56+0** | **2.41+0** |

64 Kbyte Cache

| Ways → | 1 | | 2 | | 4 | | 8 | |
|---|---|---|---|---|---|---|---|---|
| L → | 16b | 32b | 16b | 32b | 16b | 32b | 16b | 32b |
| C (bits) ↓ | | | | | | | | |
| 2 | *22.74+0.02* | *8.87+0.11* | **9.29+0** | **4.79+0** | **4.89+0** | **3.00+0** | **3.13+0** | **2.67+0** |
| 3 | *22.74+0.09* | *8.87+0.18* | **9.29+0** | **4.79+0** | **4.89+0** | **3.00+0** | **3.13+0** | **2.67+0** |
| 4 | *22.74+0.17* | *8.87+0.26* | **9.29+0** | *4.79+0.02* | **4.89+0** | **3.00+0** | **3.13+0** | **2.67+0** |
| 5 | *22.74+0.23* | *8.87+0.32* | *9.29+0.03* | *4.79+0.09* | **4.89+0** | *3.00+0.01* | **3.13+0** | **2.67+0** |
| 6 | *22.74+0.34* | *8.87+0.42* | *9.29+0.13* | *4.79+0.19* | *4.89+0.09* | *3.00+0.12* | **3.13+0** | **2.67+0** |
| 7 | *22.74+0.42* | *8.87+0.51* | *9.29+0.22* | *4.79+0.28* | *4.89+0.18* | *3.00+0.21* | *3.13+0.06* | **2.67+0** |

TABLE 5.1. Finding configurations with 1-cyle frequent value access and decode.

divided vertically into subarrays of one word width. As can be seen, for a wide range of configurations (shown in bold) there is no increase in access time as the second number is 0. This is because the decode time is quite small varying from 0.41ns for 2 bit decoding to 0.678ns for 7 bit decoding of frequent values. Since the tag path delay is greater than the data path delay, often there is enough time left over to carry out decoding. In many other cases while there is an increase in access time, the increase is very small. Finally, for direct mapped caches typically FVC design is not attractive as the increase in access time is significant. In summary, FVC is expected to perform well for cache configurations used in power efficient processors – high associativity and modest size.

## 5.3 FVC Dynamic Energy Model

A dynamic energy estimation model for the FVC cache has been developed. The energy estimation model is separated into two parts. In the first part energy estimates on a *per cache access* basis are computed. This computation is based on the detailed characteristics of the cache design shown in Figure 5.2 and depends on the cache configuration. The second part of the model estimates the *total cache energy* expended during a program run. Therefore it depends on runtime behavior of the program.

### 5.3.1 Per access energy

The energy calculations for the baseline cache is briefly described and it is compared with the energy calculations for the FVC. The equations for these calculations are presented in Table 5.2 to Table 5.4 where energy per access (EPA) is analyzed carefully. There are three main energy components for the conventional cache: energy consumed by the *data array*, energy consumed by the *tag array*, and energy needed to *drive the output multiplexer*. The energy consumed by the tag array is the same for both the baseline cache and the FVC, when they have the same associativity and capacity. Moreover, this energy must be spent for both frequent and nonfrequent value accesses. The same is true for the energy that is spent on driving the output multiplexer as in all cases output is 32 bits. The two main differences in energy computations are described next.

The first difference in energy computations arises in computing the data array energy associated with an access. One of the primary factor that determines the various components of the data array energy is the length of the word line. In case of the baseline cache, the wordline length, $wl$, is simply computed from the bits per word and the associativity. In case of a frequent value access in the FVC, $wl$ changes to $wl'$, where $wl'$ is less than $wl$ because the high-bit data array ($32 - log_2 n$ bits per word) is not accessed and it also offsets the energy spent on the extra flag bit. Similarly for a nonfrequent value access, $wl$ changes to $wl''$, where $wl''$ is greater than $wl$ since the flag bits need to be accessed irrespective of the frequent property of the value. Another major factor is the number of bitlines that need to be driven every time the data array is accessed. Similar to the reduction in word line length, the bitlines are reduced from 32 to $log_2 n + 1$ bits per word access. For example, if the codes for frequent values are only 4 bits, the amount of decrease in driving the bitlines is $(32-4-1)/32 \ = \ 84\%$. For nonfrequent values, there is one more bitline increase which is $1/32 \ = \ 3\%$.

The second difference in energy computation arises due to the decoding and encoding operations that must be carried out during read and write operations. Decoding is associated with every *frequent value read* access and each decoding is an access to the *decoder register file*. Reading nonfrequent values does not trigger decoding since it is blocked by the flag bit. Encoding is associated with *every* write access since the information of the value being encoded is not known a priori. The energy spent in encoding, $E_{fv\_encode}$, is the energy of a CAM access which differs greatly from the decoding energy, $E_{fv\_decode}$, and they are computed separately.

It is clear now that there is savings in energy when a frequent value is accessed

Energy spent on a single access on a hit in a baseline cache

$$\begin{aligned}
EPA_{cache} &= EPA_{data}(\mathbf{wl}) + EPA_{tag} + EPA_{output}(\mathbf{wl}) \\
EPA_{data} &= EPA_{d-decoder}(\mathbf{wl}) + EPA_{d-wordline}(\mathbf{wl}) + EPA_{d-bitline}(\mathbf{wl}) + \\
&\quad EPA_{d-sense-amp}(\mathbf{wl}) + EPA_{d-sense-ext-driver}(\mathbf{wl}) \\
EPA_{tag} &= EPA_{t-decoder} + EPA_{t-wordline} + EPA_{t-bitline} + EPA_{t-sense-amp} + \\
&\quad EPA_{compare} + EPA_{valid-driver} + EPA_{drive-mux} + EPA_{select-block} \\
where, & \\
&\quad EPA_{data}(\mathbf{wl})\ means\ EPA_{data}\ increases\ with\ \mathbf{wl} \\
\mathbf{wl} &= word\ line\ wire\ length\ in\ bits\ = 32 \times Associativity
\end{aligned}$$

TABLE 5.2. Per access energy model(1).

Energy spent on a single **frequent value** access on a **hit** in FVC

$$\begin{aligned}
EPA_{fv}^{Read} &= EPA_{fv} + E_{fv-decode} \\
EPA_{fv}^{Write} &= EPA_{fv} + E_{fv-encode} \\
EPA_{fv} &= EPA_{data}^{fv}(\mathbf{wl}') + EPA_{tag} + EPA_{output}(\mathbf{wl}) \\
EPA_{data}^{fv}(\mathbf{wl}') &= EPA_{d-decoder}(\mathbf{wl}') + EPA_{d-wordline}(\mathbf{wl}') + EPA_{d-bitline}(\mathbf{wl}') + \\
&\quad EPA_{d-sense-amp}(\mathbf{wl}') + EPA_{d-sense-ext-driver}(\mathbf{wl}') \\
where, & \\
\mathbf{wl}' &= FVC\ low - bit\ array\ word\ line\ wire\ length\ in\ bits \\
&= (log_2 n + 1) \times Associativity < \mathbf{wl}
\end{aligned}$$

TABLE 5.3. Per access energy model(2).

Energy spent on a single **nonfrequent value** access on a **hit** in FVC

$$\begin{aligned}
EPA_{nfv}^{Read} &= EPA_{nfv} \\
EPA_{nfv}^{Write} &= EPA_{nfv} + E_{fv-encode} \\
EPA_{nfv} &= EPA_{data}^{nfv}(\mathbf{wl}'') + EPA_{tag} + EPA_{output}(\mathbf{wl}) \\
where, & \\
(\mathbf{wl}'') &= Total\ FVC\ word\ line\ wire\ length\ in\ bits \\
&= 33 \times Associativity > \mathbf{wl}
\end{aligned}$$

TABLE 5.4. Per access energy model(3).

while there is an increase in energy consumed when a nonfrequent value is accessed. However, the savings are much greater than the increase. The per access energy characteristics of all of the configurations are also analyzed. In Figure 5.5 the percentage reductions in energy consumed on a frequent value read access are given ($L$ is the line size and $C$ is the number of bits for encoding frequent values). As can be seen from the results, the reduction per access ranges from 34% to 84%. For writes, it is a little bit lower. The reductions are higher for greater degree of associativity.

As expected, the reduction in per access energy is greater when fewer number of bits are used to encode frequent values. However, this does not imply that the least number of bits should be used to encode frequent values. When the cumulative cache energy is considered, this trend need not hold. This is because as the number of bits is increased a greater number of values can be encoded and therefore the fraction of total accesses that are frequent value accesses also increases.

The increase in per-access energy consumed for a nonfrequent value access is small in comparison to the reduction achieved for a frequent value. These increases are computed separately for read and write operations. This is because a write is more expensive as it requires an associative search in the CAM encoder. It is found that the increase in energy for a *nonfrequent value read* ranges from 0.0005% to 0.0678% for the configurations considered. This increase is mainly due to the access to the extra flag bit that is required in FVC. In contrast, the increase in energy used by a nonfrequent value write ranges from 0.776% to 4.68%. However, as can be seen, these increases are small in comparison to the energy savings that result during frequent value accesses.

### 5.3.2 Total Cache Energy

Given the per-access energy costs computed in Table 5.2, the equations shown in Figure 5.6 can now be derived for computing the total energy consumed by the cache

8 Kbyte Cache

| Ways → | 1 | | 2 | | 4 | | 8 | |
|---|---|---|---|---|---|---|---|---|
| L → | 8b | 16b | 8b | 16b | 8b | 16b | 8b | 16b |
| C (bits) ↓ | | | | | | | | |
| 2 | 42.6 | 55.5 | 60.3 | 66.9 | 70.6 | 74.9 | 78.0 | 80.1 |
| 3 | 41.9 | 54.3 | 58.3 | 64.7 | 68.2 | 72.5 | 75.4 | 77.5 |
| 4 | 39.5 | 51.5 | 56.1 | 62.2 | 65.7 | 69.8 | 72.7 | 74.8 |
| 5 | 38.4 | 49.6 | 53.6 | 59.4 | 63.0 | 66.9 | 70.0 | 71.9 |
| 6 | 35.6 | 46.1 | 50.9 | 56.3 | 60.1 | 63.9 | 67.1 | 68.9 |
| 7 | 34.1 | 43.6 | 47.9 | 52.9 | 57.0 | 60.6 | 64.0 | 65.7 |

16 Kbyte Cache

| Ways → | 1 | | 2 | | 4 | | 8 | |
|---|---|---|---|---|---|---|---|---|
| L → | 8b | 16b | 8b | 16b | 8b | 16b | 8b | 16b |
| C (bits) ↓ | | | | | | | | |
| 2 | 43.8 | 52.4 | 56.9 | 67.4 | 69.2 | 75.0 | 76.7 | 80.0 |
| 3 | 43.1 | 51.3 | 55.0 | 65.1 | 66.9 | 72.6 | 74.2 | 77.3 |
| 4 | 40.8 | 48.8 | 53.0 | 62.7 | 64.5 | 70.0 | 71.6 | 74.6 |
| 5 | 39.9 | 47.3 | 50.8 | 60.1 | 61.9 | 67.2 | 68.9 | 71.8 |
| 6 | 37.4 | 44.5 | 48.6 | 57.3 | 59.2 | 64.3 | 66.1 | 68.9 |
| 7 | 36.2 | 42.6 | 46.1 | 54.2 | 56.4 | 61.2 | 63.2 | 65.9 |

32 Kbyte Cache

| Ways → | 1 | | 2 | | 4 | | 8 | |
|---|---|---|---|---|---|---|---|---|
| L → | 16b | 32b | 16b | 32b | 16b | 32b | 16b | 32b |
| C (bits) ↓ | | | | | | | | |
| 2 | 53.4 | 63.8 | 65.0 | 75.1 | 74.9 | 79.6 | 79.8 | 82.6 |
| 3 | 52.3 | 62.2 | 62.9 | 72.6 | 72.4 | 77.0 | 77.2 | 80.0 |
| 4 | 49.8 | 59.5 | 60.6 | 70.0 | 69.9 | 74.3 | 74.5 | 77.3 |
| 5 | 48.5 | 57.6 | 58.2 | 67.2 | 67.2 | 71.5 | 71.8 | 74.5 |
| 6 | 45.8 | 54.6 | 55.7 | 64.3 | 64.4 | 68.5 | 69.0 | 71.6 |
| 7 | 44.3 | 52.4 | 53.1 | 61.3 | 61.5 | 65.5 | 66.1 | 68.6 |

64 Kbyte Cache

| Ways → | 1 | | 2 | | 4 | | 8 | |
|---|---|---|---|---|---|---|---|---|
| L → | 16b | 32b | 16b | 32b | 16b | 32b | 16b | 32b |
| C (bits) ↓ | | | | | | | | |
| 2 | 55.4 | 65.0 | 66.2 | 74.1 | 73.9 | 81.2 | 80.1 | 84.0 |
| 3 | 54.3 | 63.3 | 64.0 | 71.7 | 71.5 | 78.5 | 77.5 | 81.4 |
| 4 | 51.7 | 60.7 | 61.7 | 69.1 | 68.9 | 75.8 | 74.9 | 78.7 |
| 5 | 50.5 | 58.8 | 59.4 | 66.5 | 66.4 | 73.0 | 72.2 | 76.0 |
| 6 | 47.8 | 56.0 | 57.1 | 63.8 | 63.7 | 70.1 | 69.4 | 73.2 |
| 7 | 46.5 | 54.0 | 54.6 | 61.0 | 61.0 | 67.2 | 66.6 | 70.3 |

FIGURE 5.5. % energy savings for frequent value accesses.

$$
\begin{aligned}
E_{fvc} &= E_{Rhit} + E_{Whit} + E_{RWmiss} \\
E_{Rhit} &= \underbrace{H^{Read}_{fv} \times EPA^{Read}_{fv}}_{frequent\ value\ read} + \underbrace{H^{Read}_{nfv} \times EPA^{Read}_{nfv}}_{nonfrequent\ value\ read} \\
E_{Whit} &= \underbrace{H^{Write}_{fv} \times EPA^{Write}_{fv}}_{frequent\ value\ write} + \underbrace{H^{Write}_{nfv} \times EPA^{Write}_{nfv}}_{nonfrequent\ value\ write} \\
E_{RWmiss} &= E_{writeback} + E_{writein} \\
E_{writeback} &= \underbrace{M_{ndr-lines} \times EPA_{fv}}_{cache\ line\ not\ dirty} + \underbrace{M_{dr-lines} \times EPA_{nfv} + DR_{fv-values} \times E_{fv-decode}}_{cache\ line\ dirty} \\
E_{writein} &= (wpl \times E_{fv-encode} + EPA_{nfv}) \times (M_{dr-lines} + M_{ndr-lines})
\end{aligned}
$$

FIGURE 5.6. Total energy consumed by FVC.

in a given program run. These computations are based on the number of read and write accesses which are hits, number of references that are misses, and the memory update policy. It is assumed that the cache uses a writeback policy and a single dirty bit is maintained for each cache line to decide when the writeback is needed.

The total energy is divided into three parts: energy consumed by read hits ($E_{Rhit}$), write hits ($E_{Whit}$), and misses ($E_{RWmiss}$). Since I have already computed the energy per read/write access on a hit for a frequent/nonfrequent value ($EPA^{Read}_{fv}$, $EPA^{Write}_{fv}$, $EPA^{Read}_{nfv}$, $EPA^{Write}_{nfv}$, respectively), the computation of total read/write energy on hits for a program run can be easily computed by measuring the total number of read/write hits to frequent/nonfrequent values ($H^{Read}_{fv}$, $H^{Read}_{nfv}$, $H^{Write}_{fv}$, and $H^{Write}_{nfv}$), as shown in Figure 5.6.

The energy consumed by misses consists of two parts: energy for performing the write back operation ($E_{writeback}$) and energy consumed by line fill of the new line fetched from the L2 cache ($E_{linefill}$). The writeback energy is computed separately for dirty lines and non-dirty lines. If the cache line is not dirty, some price is still paid for reading it from the cache. This is because the reading of the low-bit array is carried out in parallel with tag matching. Therefore even if there is a miss, and the line is not dirty, the cache has already spent $EPA_{fv}$ energy in reading the portion of a word line. If the line is dirty, then the entire cache line is read first, including the portion in high bit array, and it expends $EPA_{nfv}$ energy. In addition, before performing the writeback operation, the frequent values in the line must be decoded. By measuring the number of misses of dirty lines ($M_{dr-lines}$), misses of non-dirty lines ($M_{ndr-lines}$), and number of frequent values in dirty lines ($DR_{fv-values}$), the total energy expended by the writeback operations performed in the program run can be easily computed.

It is assumed that the above process does not take more time than usual since preparing for writeback and reading new lines from lower level memory is usually carried out in parallel. The lower level memory access time is a lot longer than the higher level memory. Therefore there is enough time to prepare the lines for writing back. The energy for writing in the new line is simply the cost of updating the entire cache line ($EPA_{nfv}$) and encoding the values in the line. Note that even if a value is a nonfrequent value, energy is spent on the encoding operation because the way to determine that a value is nonfrequent is by attempting to encode it. Note that line fill does not take longer time than usual since if it was a read operation that initiated the line fill, the referenced value is passed directly to the CPU whenever it is read out from the lower level memory. Performing encoding is done only afterwords so this process does not affect program speed. On the other hand, if the line fill is initiated by a write operation, the instruction finishes execution once it has deposited its value into the cache. The actions inside the cache usually do not affect program speed unless there are back to back cache accesses.

## 5.4 Finding Frequent Values for FVC

The approach to finding frequent values is to provide a hardware FV Finder which monitors the values in initial stages of program execution and then uses the frequent values found using limited monitoring for the remainder of the program execution. Next I will introduce the new hardware FV Finder design and then I will study the impact of monitoring time on the quality of frequent values that are found.

### 5.4.1 The Design

The hardware FV Finder was briefly introduced in section 4.3. However, applying the finder into the FVC needs to be explained in details. I first repeat briefly the algorithm used in the FV finder and then see how it works in the FVC.

A small table of frequent values is used in this method. To find the top $n$ values, the table contains $2n$ entries each having a *value* field and a *counter* field as shown in Figure 5.7. The *value* field stores the data value encountered during monitoring and the *counter* field contains an $c$ bit saturating counter. Each time a data value is involved in an access by the CPU the table values are updated as follows. Suppose that a value already present in entry $i$, its counter is incremented by one. When the counter saturates, the entry $i$ is *swapped* with entry $i-1$. The purpose of this activity is to let frequent values gradually percolate to the top part of the table.

Value
counter

FIGURE 5.7. FV finder entry.

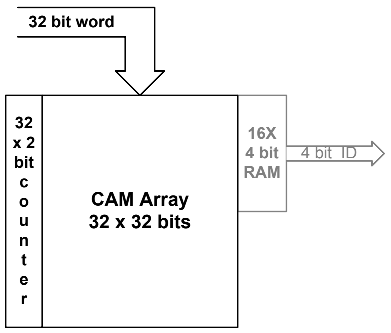


FIGURE 5.8. 32 entry FV finder and 16 FV encoder.

When a new value is encountered, and there is no free entry in the table, a victim entry needs to be selected to free up the space. An entry from the bottom half of the table with the smallest *counter* is selected since the bottom half contains less frequent values. When the counter field is short, selecting smallest counter is not expensive to implement either. I have experimented with the appropriate number of bits used in *counter* and found that the performance of a 2-bit counter is satisfactory. Therefore, selecting the smallest counter could be implemented as a *priority multiplexer* – selecting from the four states, $\{0, 1, 2, 3\}$, in descending priority. When there are multiple qualifying entries, an arbitrary one can be picked easily. After a certain amount of monitoring time, the frequent values can be obtained from the top half of the table.

### 5.4.2 Combining with the Encoder

Due to the nature of finding frequent values, the FV finder is implemented as a CAM. Section 5.1.2 has mentioned that this CAM can be combined with the encoder CAM. The complete hardware is used for different functionalities as follows. During the early period of program execution it acts as a *FV finder*. Later the monitoring stops and the hardware acts as a *frequent value encoder*. This is feasible since when the FV finder has identified the set of frequent values on the top half of the table, the lower half can be shut off and the top half can continue as the encoder (see Figure 5.8). The encoder needs the content of the *FV finder* and there is no need in replicating storage for frequent values. However, it is necessary to pass the captured frequent values to the decoder shown in Figure 5.2. Decoder initialization is a one time activity and can be done in a few cycles.

### 5.4.3 Study of Monitoring Time

The experiments described in Chapter 4 have shown the effectiveness of the FV finder. Here I compare the results with the *ideal set* which is obtained by software monitoring with unlimited resources through complete program run. The metric used for comparison is still the percentage of cache accesses involving the captured frequent values. The number of frequent values were varied from 1 to 128. The experimental results are presented in Figure 5.9 and 5.10.

In these figures, the legend shows the monitoring duration of the FV finder in terms of the number of instructions. In parenthesis the monitoring duration is given in terms of the percentage of the total executed instructions. Generally, the longer the FV finder runs, the greater is the number of frequent value cache accesses as the frequent value set is more accurately determined. The curve on the top represents the data for an ideal set of frequent values. If most of the curves are close to the ideal curve it means that the FV finder is very effective. From the graphs we can see that 8 out of the 11 tested benchmarks belong to this category. For these benchmarks the degree of monitoring has little impact on the quality of the frequent value set identified. This is because for these programs frequent values can be divided into two categories: a small number of values that are encountered extremely frequently throughout the program's execution and a much larger set of values that are encountered nearly equally frequently during the program's execution. A very small amount of monitoring is needed to identify values in the first category. A small amount of variation is due to the difference in the set of values identified from the second category as frequent values.

For the remaining three benchmarks – `130.li, 132.ijpeg and 134.perl`, the results from the FV finder differ from the ideal result greatly. There could be a number of reasons for this difference. First, the FV finder may not have run for enough time to capture the top frequent values as some of these values may show up later in the program execution. Second, the FV finder may not have enough entries to capture the accurate set of frequent values at any point. This could be verified by comparing the results with that obtained from software profiling where it is assumed that the FV finder has as many entries as needed. Figure 5.11 shows that for `130.li` and `134.perl`, with unlimited resources the top frequent values do appear as early as the first 5% of program execution time. Therefore, the inaccuracy in Figure 5.10

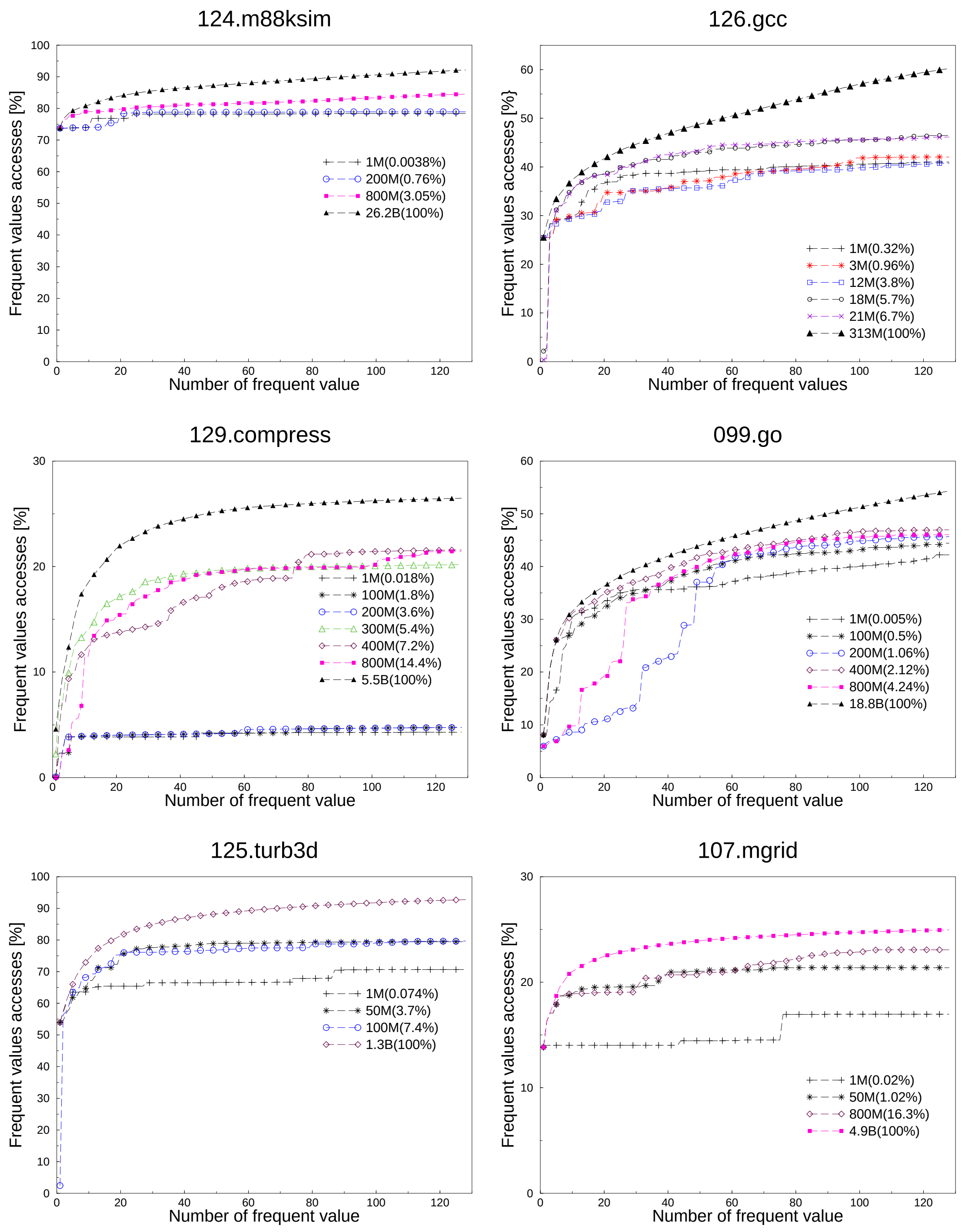


FIGURE 5.9. Comparing frequent value set identified by the FV finder with an ideal set.

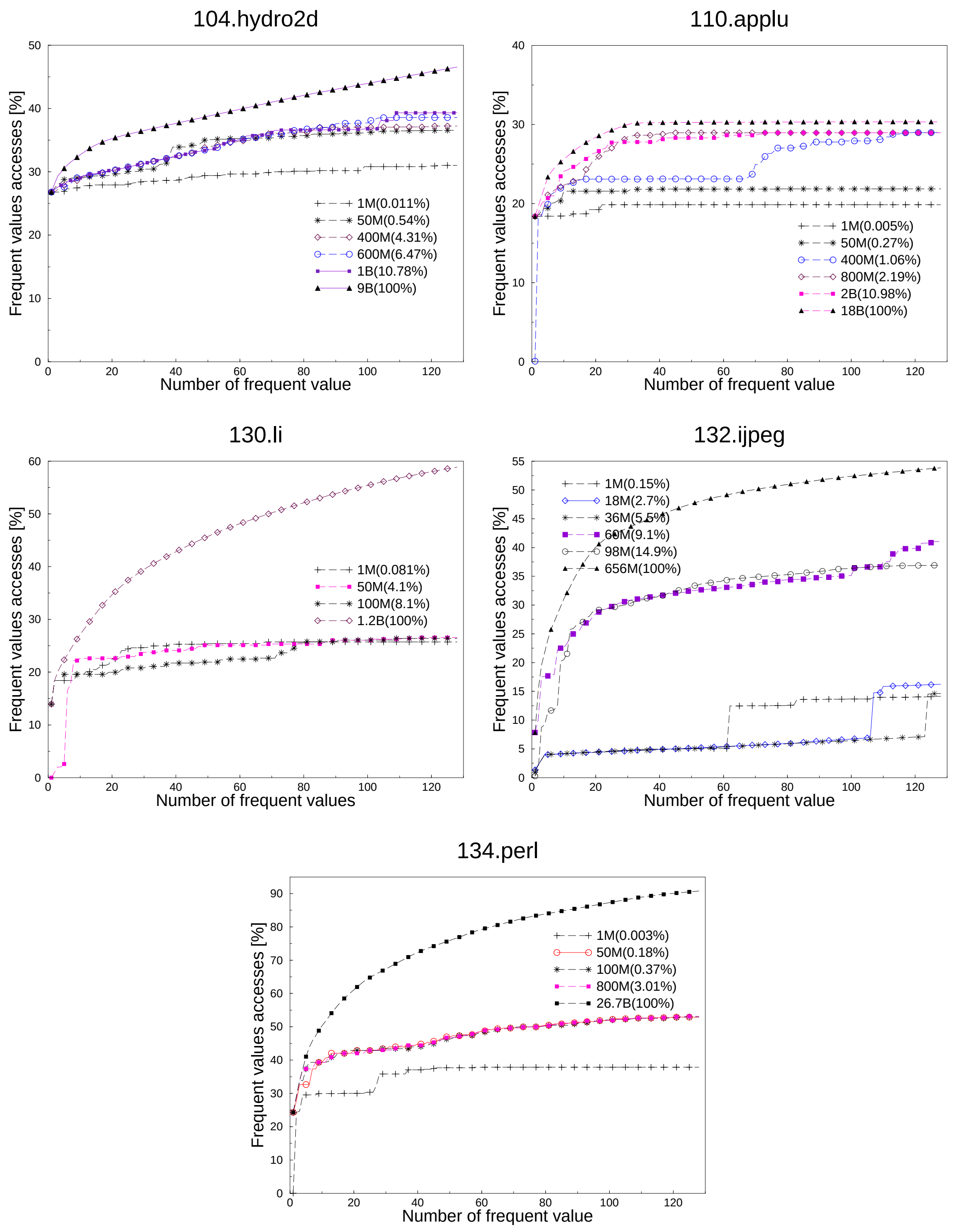


FIGURE 5.10. Comparing frequent value set identified by the FV finder with an ideal set.

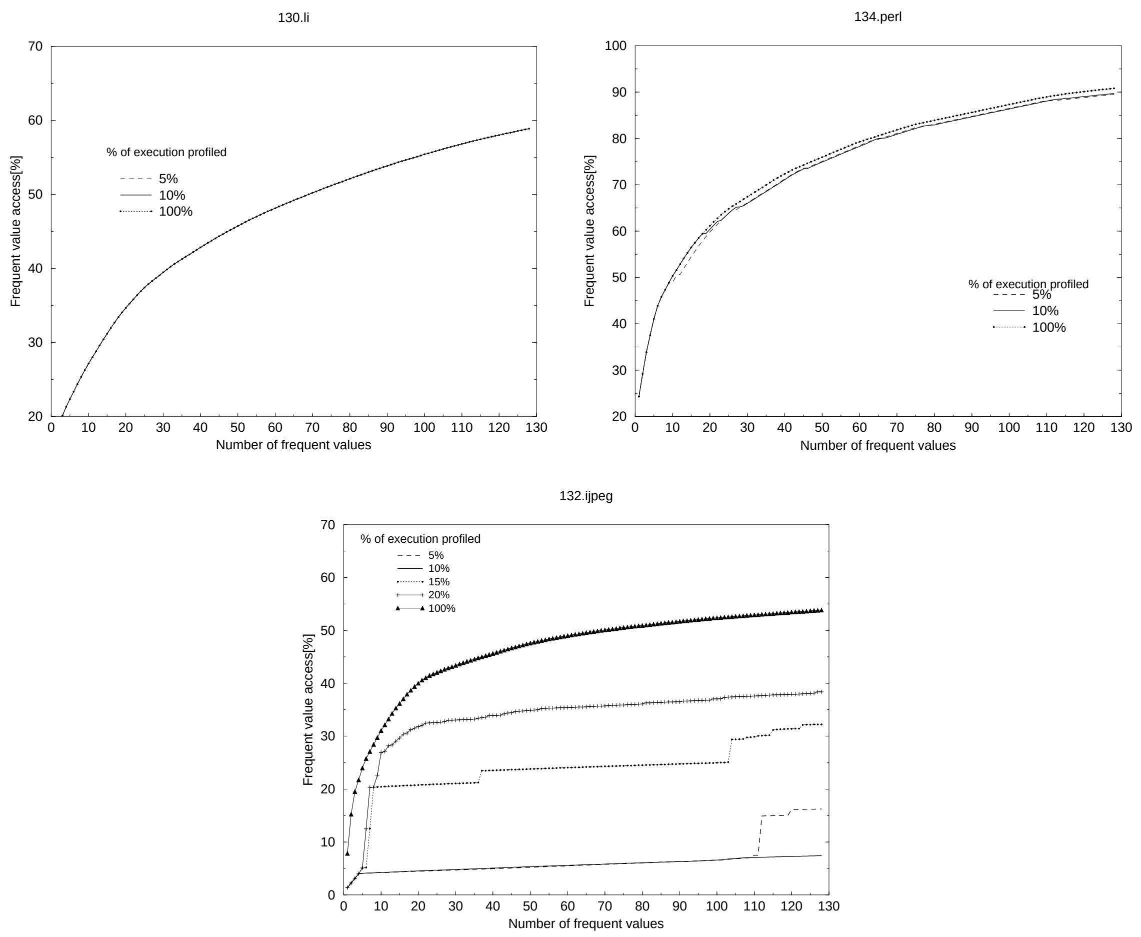


FIGURE 5.11. Comparing frequent value set identified using software method.

is probably due to the simple design of the FV finder. Figure 5.11 also shows that for `132.ijpeg`, even with infinite bookkeeping, the frequent values do not become stable not until about 15% of program has run. For this case, the FV finder only needs to run longer to capture the right set of the frequent values. Therefore, the greater the duration of monitoring, the better is the resulting set of identified frequent values.

Notice that in the above study the FV finder ran for the first few percent of program instructions which indicates that the number of memory accesses is even fewer. Since the only instructions that affect the energy consumption and the performance of the FV finder are memory instructions, I will run it for the first 5% of memory accesses in the experiments. This is reasonable even when there is no total instruction information since it is shown above that for most programs, running the FV finder for short amount of time is sufficiently good.

## 5.5 Experiments

### 5.5.1 Experimental Setup

Low power processors being developed at present (e.g., StrongArm), have simple in-order pipelines and reasonably small associative caches. The simplicity of these designs is the key to achieving low power. Therefore the FVC is also evaluated in that context.

The baseline processor supports a five stage in-order pipeline with the MIPS ISA. The simulator is based upon the system described in [44] into which the XCACTI 2.0 was integrated with modifications necessary for FVC. The XCACTI 2.0 is also modified to model energy on every write, energy in write back and line fill operation so that the formulas in total energy model (Figure 5.6) are accurately applied. All the benchmarks are compiled with *gcc* version 2.7.2 for a MIPS compatible processor using optimization level -*O3*. The cache parameters used in this study are given below. In addition, the L1 data cache latency is 1 cycle for the baseline and 1-2 cycles for the FVC corresponding to frequent-nonfrequent value accesses. The number of frequent values used was 16, 32 and 64. The L1 instruction cache latency is 1 cycle and L2 cache latencies are 6 cycles. The cache sizes used vary from small (8 Kbyte) to modest (64 Kbyte). The number of frequent values used are 16, 32, and 64. For these L1 cache configurations of FVC there is no increase in access times as the decoding of frequent values can be carried out according to the timing constraints presented earlier.

I ran the above configurations for the same set of benchmarks in section 5.4

| | Level | Size (KB) | Line Size (B) | Ways |
|---|---|---|---|---|
| Data Cache | L1 | 8 | 16 | 4 |
| | | 64 | 32 | 8 |
| | L2 | 128 | 64 | 8 |
| Instruction Cache | L1 | 8 | 16 | 4 |
| | | 64 | 32 | 8 |
| | L2 | 128 | 64 | 8 |

TABLE 5.5. Cache Configuration

using reference inputs. During the first 5% of the memory accesses the cache acts as a traditional cache and the FV finder captures the frequent values. After that, the cache acts as a FVC and the FV finder shuts down the lower half of the table and uses the content of the upper half for encoding. It is observed that the FV finder performs better than the study whose results were presented in the preceding section because it is now running for a longer duration of time. I further show increases in processor execution time delay, percent of decrease in cache energy alone, decrease in energy$\times$delay (the product of the first two metrics), and decrease in power (energy/delay).

### 5.5.2 Experimental Results

**FVC Hits Due to Frequent Values.** Figure 5.12 plots the percent of frequent value hits, including reading and writing of frequent values to the cache, in the 95% of the memory accesses performed after frequent values have been found. Bigger caches have more hits therefore a higher FV hit ratio. It is found that the results for `130.li` and `134.perl` improved by about 10% which means monitoring for longer time is beneficial. As expected, for the rest of the benchmarks there are slightly better frequent value hit rates than Figure 5.9–5.10.

**Cache Results.** The saving in cache energy, increase in program execution time, saving in energy delay product and power are shown in Figures 5.13–5.16. Generally,

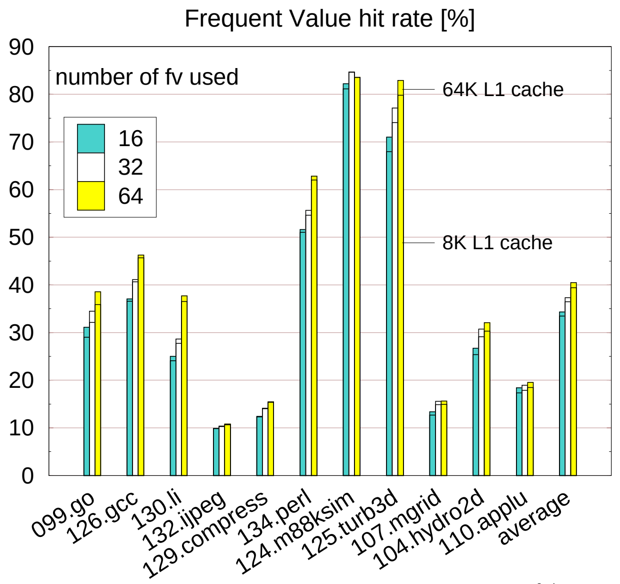


FIGURE 5.12. Cache hits that are FV over the 95% cache accesses.

programs with higher FV hit rates save more energy (see Figure 5.13) and slow down less (see Figure 5.14). As can be seen, on average this increase in execution time is 3.7%/ 3.5%/ 3.3% for 16/ 32/ 64 frequent values respectively for a 64Kb FVC. The savings in energy energy×delay, and power are substantial. For a 64Kb FVC, on an average, there is 26%/ 27.6%/ 28.8% reduction in cache energy, 23%/ 24.8%/ 26.1% reduction in $energy \times delay$, and 28.9%/ 30.3%/ 31.3% reduction in power for 16/32/64 frequent values, respectively. The savings are larger for larger number of frequent values and for larger sized cache. This means that the increased energy spent in coding and decoding for larger number of frequent values is offset by greater savings due to a greater number of frequent value accesses. The only benchmark that presents high slowdown (8∼9%) from Figure 5.14 is `107.mgrid`. First of all, it does not have abundant frequent value cache accesses from the study in Figure 5.10 so it cannot benefit much from the FVC design. Secondly, with the similar FV hit rates as `132.ijpeg` shown in Figure 5.12, the reason `107.mgrid` has more slowdown might be because the large amount of immediate data dependency between loads and the following instructions. The other two benchmarks `132.ijpeg` and `129.compress` present increase in energy×delay. This is also due to the low FV hit rates as it can be seen from figure 5.12. However, the increases are within only 1.75%.

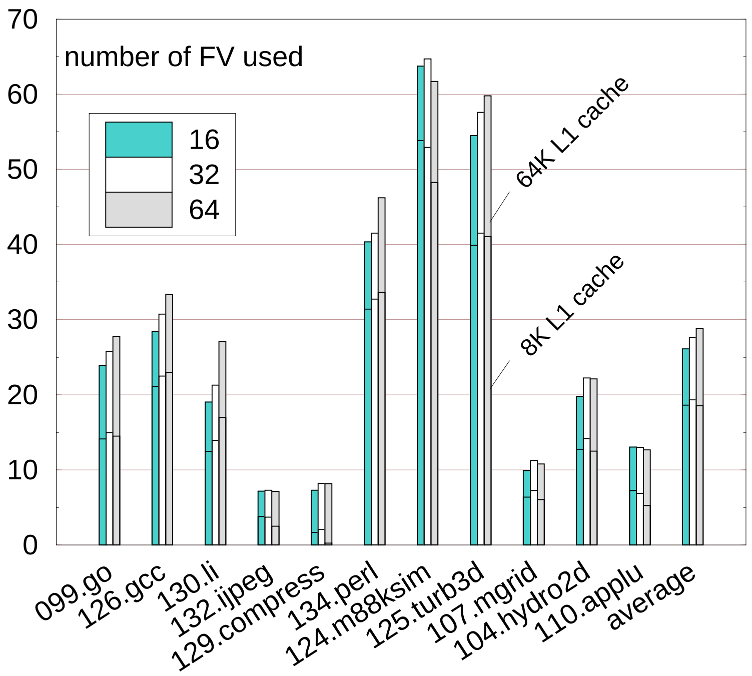


FIGURE 5.13. Cache Energy Reduction.

In another experiment I used Wattch [10] to compute the percentage of energy spent by the L1 data cache using the configuration in Table 5.5. For example, `gcc` uses 14.8% of the total energy in the L1 data cache, and the energy saving from using FVC is 33.4% (see Figure 5.13 for 64KB using 64 values) of the original L1 data cache, which corresponds to 4.9% savings of the total processor energy. The total execution time is increased by 3.3% which roughly means that besides L1 data cache, the energy spent by rest of the processor is increased by 2.8%. Therefore the net savings in energy was 2.1%. This one example shows that using FVC does provide net energy savings.

**FV Finder Energy.** I also used Wattch to estimate the energy spent for the FV finder. I applied their CAM model and 2-bit register for the counters. The energy for

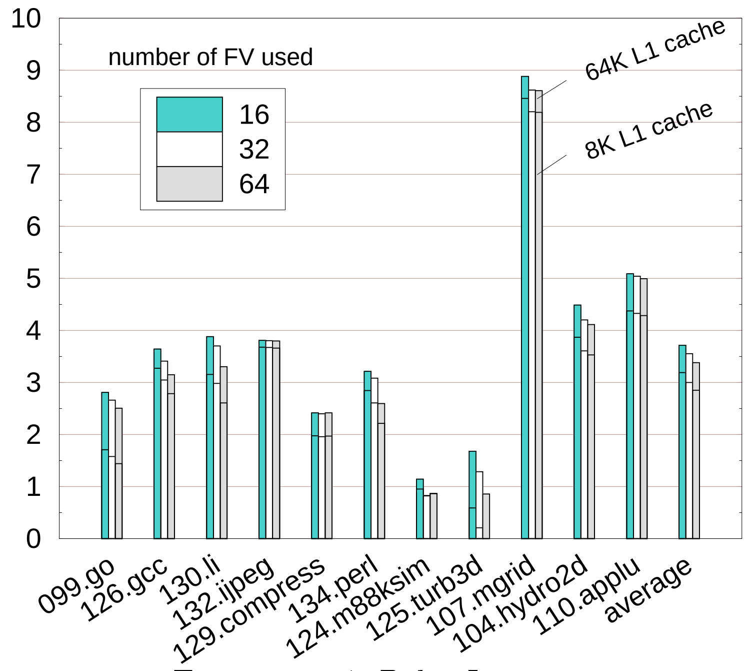


FIGURE 5.14. Delay Increase.

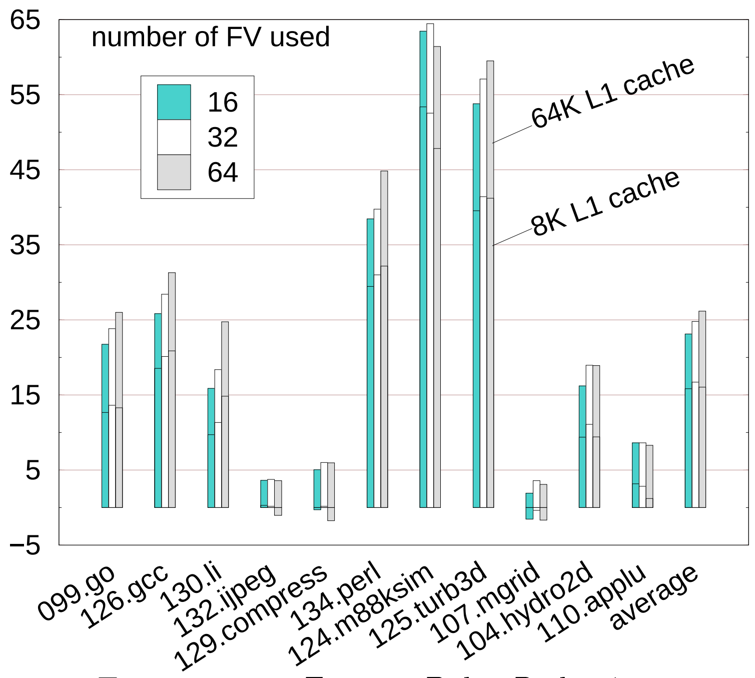


FIGURE 5.15. Energy×Delay Reduction.

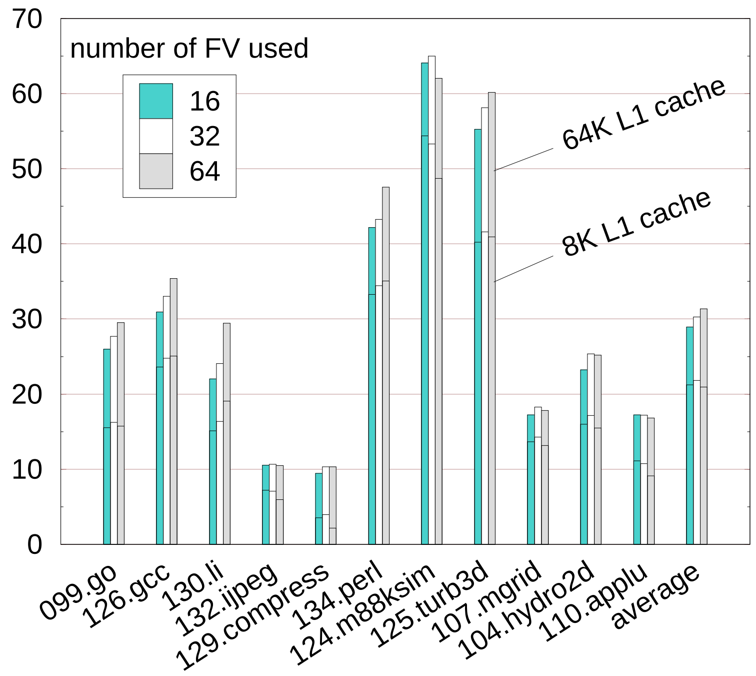


FIGURE 5.16. Power Reduction.

the FV finder ranges from 0.3% to 6.07% of the total data cache energy used by an 8K to 64K FVC. This is because the FV finder runs for only first 5% of the memory accesses therefore does not introduce much overhead.

**Discussion on Multiprogramming Environment.** So far the experiments are limited to a single program environment. In many situations however, a CPU runs multiple programs in a time sharing manner. Those programs will share the cache as well. To handle this multiprogrammed environment in the FVC design, a mechanism must be designed so that the frequent values of each program are not lost every time it is taken off the CPU. Fortunately, this is not difficult to solve. All the frequent values can be saved as the status of the program during context switch, just like saving the registers. In this way, all the frequent values are safely stored and do not need to be found again when the program obtains the CPU next time. During a context switch, the FVC needs to reset all the flag bits in the low bit array so that it starts fresh for the new program. Since the new program is likely to incur many cache misses after the context switch, the FVC can be rapidly tuned to full functionality since cache misses will force the linefill to perform encoding.

**Load marking for minimizing delay increase.** For the programs that have lower frequent value cache access rates, such as `107.mgrid`, the delay increases are higher than the other programs as shown in Figure 5.14. This is because most of the nonfrequent value accesses take two cycles, lengthening the average cache access time. To keep the number of cache access that require two cycles instead of one minimum, a *load marking* scheme is further developed as the following.

The loads that are involved in nonfrequent value accesses most of the time are marked. When these loads are encountered at runtime, the two data arrays are read simultaneously. Thus, this approach enables reading of nonfrequent values in one cycle by the marked loads. Therefore, the overall increase in delay is reduced. If only those loads that are rarely involved in frequent value accesses, the energy savings will be effected minimally.

The potential of the above idea is indicated by the study whose results are presented in Figure 5.17. For each statically distinct load, through profiling, we collected the percentage of dynamic accesses by the load that involve nonfrequent value accesses. Next we set a threshold value for the maximum percentage of nonfrequent value accesses allowed by the dynamic instances of a static load. This threshold is referred as the *nonfrequent value access threshold* (T). Each static load is classified as a *nonfrequent value load* if fewer than T% of its dynamic instances performing nonfrequent value accesses; otherwise it is classified as a *frequent value load.* In Figure 5.17, the values along x-axis represent different threshold values. The values along y-axis represent the number of load instances, as a percentage of total number of dynamic loads, that are due to the loads that are classified as *nonfrequent value loads* for a given threshold.

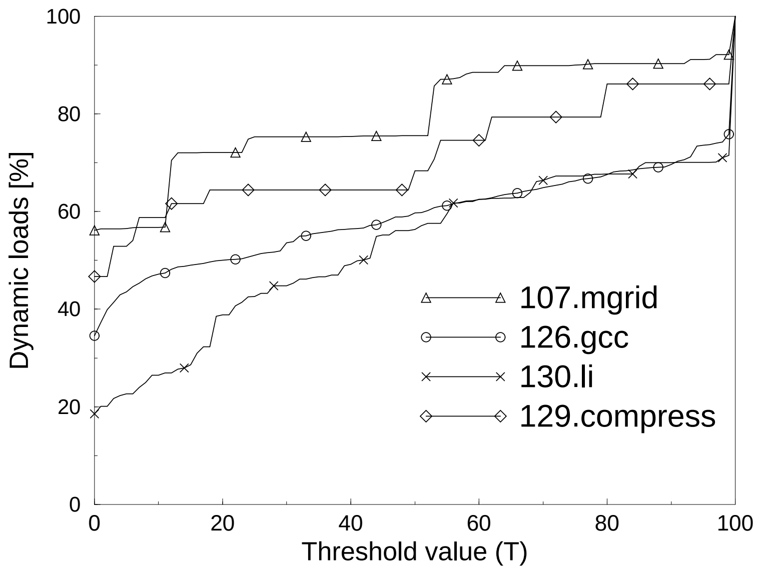


FIGURE 5.17. Load marking.

Two observations can be made from the data in Figure 5.17. First that most loads are involved in some nonfrequent value accesses. Second and even more important observation is that there are some static loads that always access nonfrequent values. They are chosen as *nonfrequent loads* and are marked in the program. By accessing both data arrays simultaneously for these loads, no reduction in data cache energy savings will occur. At the same time the increase in delay will be reduced as values will be obtained in a single cycle by these loads.

Experiments are rerun for some of the programs to see if there is a significant difference in the delay increase due to load marking. Figure 5.18 shows the results for the benchmarks been tested. The second column shows what percentage of the dynamic loads that arise from marked static loads. The third column shows the increase in delay before applying the load-marking scheme (taken from Figure 5.14). The last column shows the increase in delay after applying the load-marking scheme. In this experiment a 64K-byte FVC with 64 frequent values is used.

As can be seen, the results of load marking are quite positive. The benchmark having the highest slowdown (i.e., `107.mgrid`) without load marking now has a sig-

| Benchmarks | loads marked | Delay Increase | |
|---|---|---|---|
| | | before | after |
| `107.mgrid` | 56% | 8.6% | 5.1% |
| `129.compress` | 47% | 2.4% | 1.4% |
| `126.gcc` | 34% | 3.1% | 1.8% |
| `130.li` | 18% | 3.3% | 2.6% |

FIGURE 5.18. Effects of load-marking on delay.

nificantly lower increase in delay—5.1% as opposed to 8.6%. Other two programs also show significantly reduced delay increases of 1.8% and 2.6%. Note that the load marking scheme does not affect the energy consumption since all the marked loads consume the same amount of energy as they did before load marking. Thus, the load marking scheme is an effective way to reduce the FVC delay increase.

## 5.6 Summary

This chapter has introduced a frequent value cache design. The FVC stores frequent values in encoded form so that they only need a few bits instead of the full 32 bits in representation. With this encoding, the data array part of the cache is split into a smaller array and a larger array. The smaller array holds the short encoded form for frequent values and partial word of the same width for the nonfrequent values. The larger array holds the remaining part of the word for both frequent and nonfrequent values. A detailed access time and access energy study of FVCs was carried out with varying configurations. The hardware design of a frequent value finder was also developed and shown to be simple and effective. From additional experiments it can be concluded that for a wide range of configurations for set-associative data caches, the FVC compares favorably with a conventional data cache design. This is because the time spent on encoding and decoding of values does not impact the cache access time and the reduction in dynamic energy consumed by a *frequent value* read access is quite high.

# Chapter 6
# Frequent Value Data Bus Encoding Scheme

In CMOS circuits most power is dissipated as dynamic power for charging and discharging of internal node capacitances. Thus, researchers have investigated techniques for minimizing the number of transitions inside the circuits. The capacitances at I/O pins are orders of magnitude higher than internal capacitances. Thus, the power dissipated at the I/O pins is even greater than that dissipated at internal capacitances. Therefore techniques for minimizing switching at external address and data buses (see Figure 1.3), even at the expense of a slight increase in switching at internal capacitances, have been investigated for reducing power consumption [7, 8, 15, 17, 43, 50, 60, 61].

Many of the encoding schemes, such as the bus-invert coding [60], are general purpose and can be applied to both address and data buses. General purpose techniques can only provide modest reductions in switching activity. This is because the characteristics of values sent over data and address buses vary and thus using the same technique for both types of buses is not the most effective solution. To obtain greater reductions special characteristics must be identified for information transmitted over address and data buses. Using such a specialized approach significant success has resulted from research into minimizing switching at external address buses. In particular, the technique described in [43] is particularly effective as it reduces the address bus activity by as much as 66% for some benchmarks. The key to achieving such high reductions by this technique is its ability to exploit memory reference locality. The memory regions being referenced by a program are divided into working zones. Instead of transmitting a sequence of complete addresses that exhibit locality, in this technique, the offset of the current reference with respect to the previous reference to the same *working zone* is sent over the bus, along with an identifier of that zone. Since the offsets are quite small, in comparison to complete addresses, the "one-hot" encoding, which generates codes with zero's and a single one, can be used to transmit them and thus the number of switching transitions is greatly reduced.

The goal of this chapter is to develop a technique for data buses that is as effective as the above technique is for address buses [43]. The working zone technique is effective because it exploits the characteristic of address locality. Until now an effective specialized approach for a CPU's external data buses has been illusive. This is because no suitable characteristic for values transmitted over a data bus has been found. Unlike memory references that exhibit locality, the data values do not exhibit similar locality. In fact the values transmitted over the data bus may vary widely across the range of representable values.

The chapter is organized as follows. Section 6.1 presents the base FV encoding scheme in detail and shows the experimental results. Section 6.2 describes how to reduce encoder overhead. In Section 6.3 the effectiveness of the coders are discussed. And in Section 6.4 the encoding scheme for different processor configurations are tested. Section 6.5 compares the FV encoding algorithms with related research. Conclusions are given in section 6.6.

## 6.1 FV Data Bus Encoding

The design of the encoder and decoder used to reduce the switching activity on the data bus is now presented. The overall approach is as follows. The frequent values are transmitted over the bus in encoded form while the nonfrequent values are transmitted in their original unencoded form. The set of frequent values are kept in a table implemented as a CAM by both the encoder and the decoder. This table is searched and if the value to be transmitted is found in it, then the value is regarded as a frequent value which is then transmitted in encoded form. In order to ensure that the decoder can determine whether the transmitted value is in encoded form or not, an additional *control* signal must be sent from the encoder to the decoder in some situations. As will be described later in this section, the proposed method for maintaining frequent values is such that the contents of the frequent value tables at both the encoder and the decoder are always identical. In the remainder of this chapter the base encoding scheme is first described in detail and then some enhancements to this base scheme are described.

### 6.1.1 The Base FV Encoding Scheme

The method for encoding frequent values has the flavor of one-hot encoding with one important difference. The FV encoding scheme overcomes the major drawback of one-hot encoding in that it does not require $2^n$ wires, where $n$ is the number of bits representing the value, to transfer the data. Instead it achieves low switching activity by using the same number of wires as the data bus width. In the experiments 32-wire bus width is assumed.

The above goal is achieved as follows. The "hot" wire generated from the encoder is not used to represent the true decimal value being transfered but rather it indicates in which entry of the frequent value table in the encoder or decoder the frequent value can be found. In other words, if the $i^{th}$ entry in the frequent value table is found to contain the same value as the one being transmitted, then the $i^{th}$ output wire is set to 1 and all the remaining wires are set as 0. This is how a *one-hot code* is formed and sent over the data bus, completing the coding process (see Figure 6.1a). When the decoder receives the code from the bus, it reads out the value from the $i^{th}$ entry indicated by the code. It will be shown later how this method for maintaining the contents of the tables at the encoder and decoder ensures that the contents of the two tables are identical and thus the value is correctly decoded. Under the above scheme, if frequent values are transmitted back to back, then at most two bits switch while all other bits remain zero. This is how FV encoding reduces switching activity.

The nonfrequent values are transmitted in unencoded form. If a value to be transmitted is a nonfrequent value it cannot be found in the encoder CAM. Thus, the encoder does not attempt to generate a code. Instead, it simply passes the original value onto the data bus. When the decoder receives the value and finds more than one hot wire in it, it concludes that the transmitted value is not encoded (see Figure 6.1b).

It is possible that a nonfrequent value being transmitted in unencoded form contains a single high bit and all of its remaining bits are zeros. The FV encoding scheme ensures that the decoder does not erroneously decode this value by sending a single bit control signal from the encoder telling the decoder to skip decoding (see Figure 6.1c). The experimental results also include the switching overhead from sending the control signal.

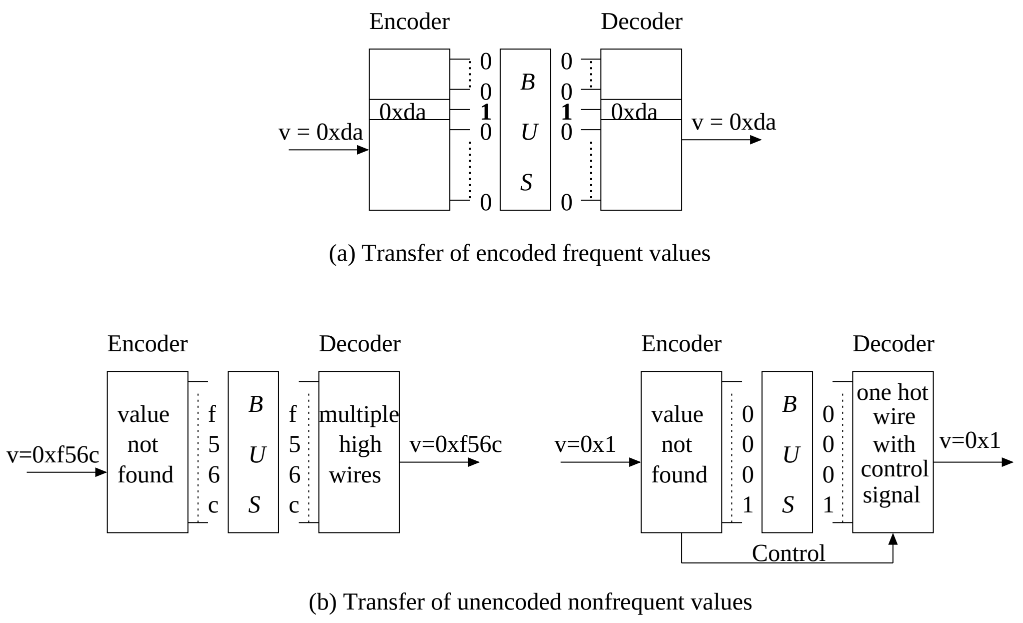


FIGURE 6.1. Encoding-decoding setup.

### 6.1.2 Keeping Encoder and Decoder FV Tables Consistent

Both the encoder and the decoder are effectively frequent value finders. Section 4.4 shows the algorithms they follow. It is extremely important to keep the sender side encoder and the receiver side decoder consistent all the time. The **same** replacement policy is used for both to assure they contain the same values. In more detail, if there are multiple entries that have the same timestamp, both the encoder and the decoder follow the same rule for picking up a victim, i.e., the first victim they encounter during searching. By doing so, it can be guaranteed that both sides contain not only the same values but also the same indexes for every value. The grounds for this to be true is that they have the same timestamp value and reference bit. This can be easily achieved by using the same time interval for updating the timestamp and the reference bit.

Next, the enhancements of the basic FV encoding schemes are introduced. The improved techniques can further reduce the bus switching activity greatly.

### 6.1.3 Enhancements of Base FV Encoding Scheme

**XORing Values.** The base encoding scheme reduces switching to at most 2 bits if a frequent value being transmitted is also preceded by a frequent value. While the base encoding scheme gives good performance when frequent values are encountered back to back, a pattern of intervening frequent and nonfrequent values is not favorable to the base scheme. In Figure 6.2 the percentage of traffic due to frequent values that are also preceded by frequent value transmissions is given. On average this number is 16%. From the data presented earlier in Figure 4.12 it can be seen that on average the frequent values account for 32% of the overall traffic. Therefore on an average 16% of transmitted values are frequent values that are preceded by nonfrequent values.

The base FV algorithm can also reduce switching between nonfrequent and frequent value transmissions using a decorrelator described in [8, 43]. If one takes the

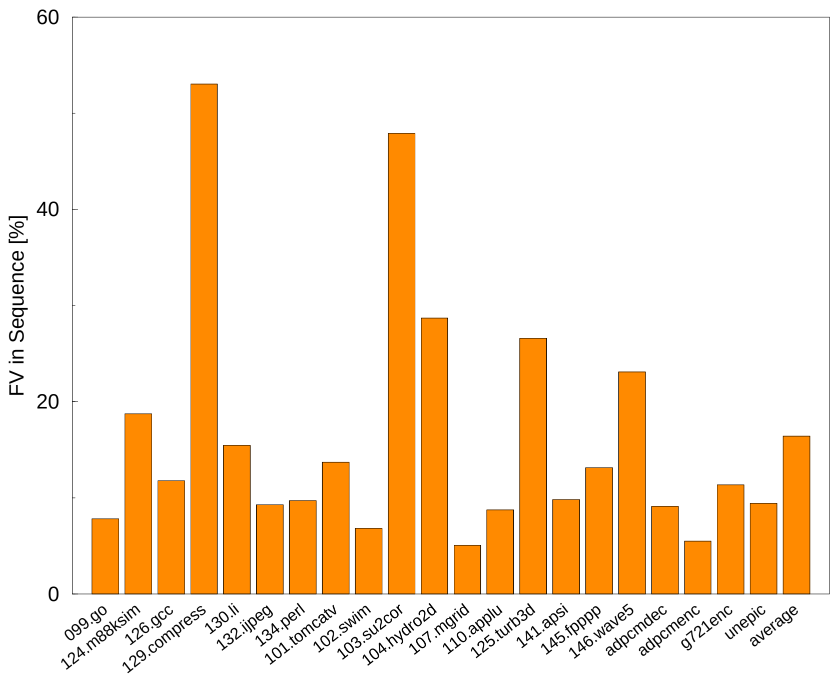


FIGURE 6.2. Occurrence of frequent values in sequence.

$XOR$ of the current value to be transmitted ($Code_n$) and the previously transmitted value ($Send_{n-1}$), then this has the effect of flipping only those wires of the bus that were low when $Send_{n-1}$ was sent and are high in $Code_n$. Therefore if $Code_n$ corresponds to a frequent value, it contains only one high bit and therefore no matter whether it is preceded by a frequent value or a nonfrequent value (i.e., $Send_{n-1}$ is frequent or nonfrequent) the switching activity is only 1 bit. In other words, transmission of a frequent value always results in switching of one bit. The combination of FV encoding and XORing current code with the previous value sent over the data bus is shown in Figure 6.3.

**Equality Test.** XORing the values can help reduce switching when different codes are to be transmitted in sequence. However, it also brings unnecessary switching when the same code is transferred repeatedly. For example, if a code with the $ith$ bit hot was transferred $n$ times continuously, the switching on bus will toggle $n$ times at the

$$Send_n = Send_{n-1} \oplus Code_n$$
$$Code_n = Send_n \oplus Send_{n-1}$$

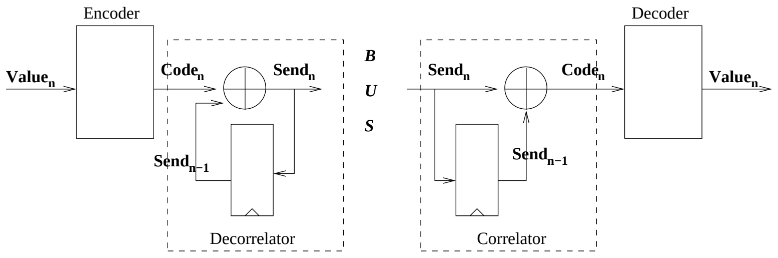


FIGURE 6.3. Reducing switching by XORing values.

*ith* wire. This increases switching since transferring the same code should not induce any switching while in the FV+XOR encoding, it does cause 1 switch, and eventually it can increase overall switching. Figure 6.4 shows how often this situation arises. It gives the percentage of traffic due to transmission of a code that is immediately followed by the same code. On average this situation accounts for 16% of the traffic. It is observed that this characteristic is observed at low levels in most benchmarks. However, for a few benchmarks (e.g., `compress` and `turb3d`) this situation is very common. As a result, for these benchmarks in particular, the switching caused due to repeated transmission of the same code should be avoided.

The additional switching can be removed easily as shown in Figure 6.5. The encoder can keep a register of the last value ($Value_{n-1}$) transferred and compare it with the current value ($Value_n$). If the two values match, the code for the last value ($Code_{n-1}$) on the bus can be sent again. The receiving side, without knowing the equality property of the current value, puts the code through the correlator. Since the code is the same as the last code, the correlator, namely XOR, will compute the result 0 as Code$_n$. There are now two cases where $Code_n$ can be 0: one is when $Send_{n-1}$ is sent twice back to back as it just explained; and the other is when $Value_n$ is 0 and 0 is not a frequent value and therefore not encoded. One can disambiguate the two cases by hardwiring an entry in the encoder/decoder to 0 to make 0 a permanent frequent

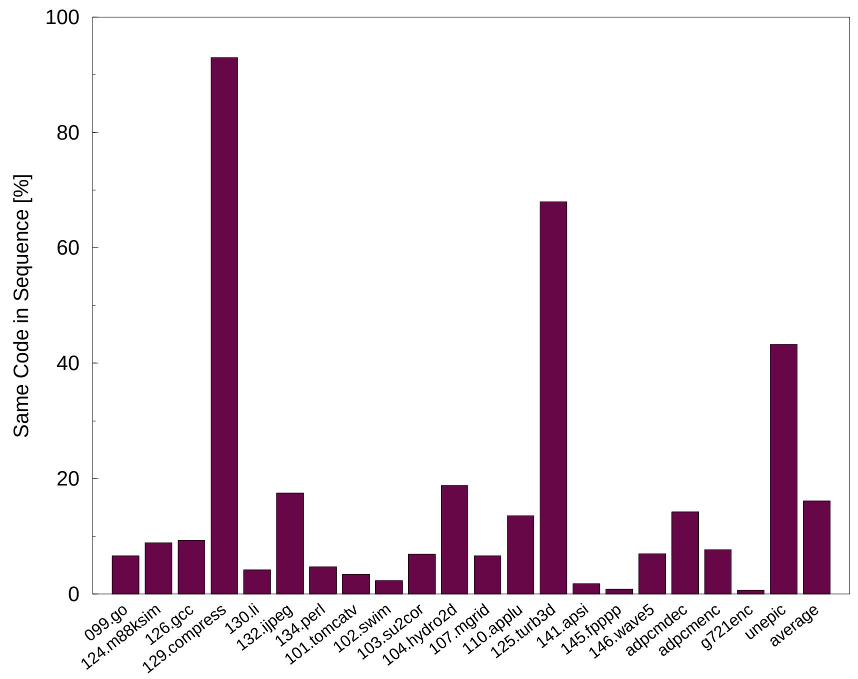


FIGURE 6.4. Transmission of identical code in sequence.

value which is therefore always transmitted in encoded form. This leaves only one possibility for $Code_n$ to be 0, which corresponds to the case when the same value is being transfered again and therefore the decoder can simply output the last value it produced. Note that in this process, the sending side did not initiate the activity of the encoder or the decorrelator and the receiving side used only the correlator. That way the energy spent in the encoder and the decoder is also saved.

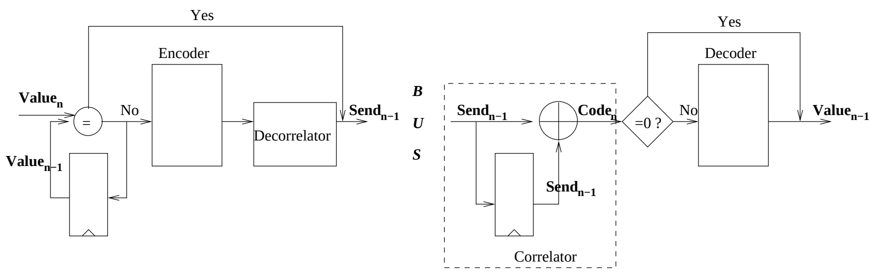


FIGURE 6.5. Dealing with equal code transfer.

**Hamming Distance Based Exclusion of Frequent Values.** So far in the discussions I have considered all encountered values as candidates for being frequent values. However, it should be noted that not all frequent values are equally effective in reducing switching activity. The impact of encoding a frequent value is proportional to the hamming distance between the unencoded frequent value and the corresponding one-hot code assigned to it. The hamming distance between a small unsigned value and a one-hot code is quite small. Therefore whether these small unsigned values are transmitted in unencoded form or in a one-hot encoded form, the number of switches that will occur will be very close. It is possible that by excluding such values from consideration during frequent value identification, better performance may be achieved. First, their exclusion will allow entries in the frequent value table to be used by other values which are not as frequent but have a greater hamming distance from the one-hot code they are assigned. Second, the encoding and decoding activity will be reduced because the frequent value table need not be accessed for these values at all.

**An Example.** Figure 6.1 illustrates how the FV encoding scheme and its enhancements are able to reduce the switching activity. It compares the switching activity for a sample sequence of values without encoding and with different levels of encoding. Here it is assumed that the initial value on the data bus is 0 which is followed by two frequent values, one nonfrequent value, and finally two more frequent values shown in the first column of the first table in Figure 6.1. All values are written in hexadecimal format. If no encoding is carried out the number of bit transitions for this sequence is 32, as shown in the first table. This number reduces to 9 when the frequent values are encoded using the base FV encoding scheme. The reduction arises due to transmission of one hot-codes as opposed to original values with large numbers of high bits. The application of XOR reduces bit transitions by one bit during transmission of second, third and fourth values. However, it also increase the bit transitions for

| **Transmitted Binary Value Without Encoding** | Switching | Comments |
|---|---|---|
| 0xff | 8 | frequent value |
| 0xfff | 4 | frequent value |
| 0x300 | 10 | |
| 0xfff | 10 | frequent value |
| 0xfff | 0 | frequent value |
| Total Switching | ***32*** | |

$\Downarrow$

| **Transmitted Binary Value With FV Encoding** | Switching | Comments |
|---|---|---|
| 0x1 | 1 | one-hot code for 0xff |
| 0x2 | 2 | one-hot code for 0xfff |
| 0x300 | 3 | |
| 0x2 | 3 | one-hot code for 0xfff |
| 0x2 | 0 | one-hot code for 0xfff |
| Total Switching | ***9*** | |

$\Downarrow$

| **Transmitted Binary Value With FV Encoding + XOR** | Switching | Comments |
|---|---|---|
| 0x1 | 1 | one-hot code for 0xff |
| 0x3 | 1 | xor of 0x2 and 0x1 |
| 0x303 | 2 | xor of 0x300 and 0x3 |
| 0x301 | 1 | xor of 0x2 and 0x303 |
| 0x303 | 1 | xor of 0x2 and 0x301 |
| Total Switching | ***6*** | |

$\Downarrow$

| **Transmitted Binary Value With FV Encoding + XOR + Equal** | Switching | Comments |
|---|---|---|
| 0x1 | 1 | one-hot code for 0xff |
| 0x3 | 1 | xor of 0x2 and 0x1 |
| 0x303 | 2 | xor of 0x300 and 0x3 |
| 0x301 | 1 | xor of 0x2 and 0x303 |
| 0x301 | 0 | same value, no transition |
| Total Switching | ***5*** | |

TABLE 6.1. An example illustrating reduction in switching transitions using FV encoding. The value stream in the first table contains two frequent values: 0xff and 0xfff whose one-hot codes are 0x1 and 0x2 respectively.

the last value from no bits to 1 bit transition. By performing the equal test this additional bit transition can be avoided leading to the final bit transition count of 5 bits.

### 6.1.4 Experiments

**Testing Different Components and Their Combinations.** The experiments were conducted by executing the `SPEC95INT`, `SPEC95FP`, and a subset of `mediabench` programs for embedded applications. I measure the reductions in *switching activity* on the external data bus due to FV encoding and its enhancements. I tested the impact of each component in the FV encoding scheme in reducing switching. The purpose is to answer the question: are all the components needed and if yes how much benefit does each one bring? To see this, I first considered the following three configurations of a frequent value based encoding algorithm:

1. *FV Encoding Only* – This is the base FV encoding algorithm.
2. *FV Encoding + XOR* – This is the base FV encoding algorithm enhanced with the correlator on sender side and decorrelator on receiver side.
3. *FV Encoding + XOR + Equal* – This is the complete encoding algorithm including the base FV encoding algorithm with the two enhancements of XORing values and performing equality test.

The results are shown in Figure 6.6. On average, the first configuration that uses only the base FV encoding scheme provides nearly 13% reduction in switching activity. The second configuration that uses the base FV encoding scheme and the XORing of values, on average, doubles the reduction in switching activity to nearly 26%. This is consistent with the previous observations. Recall that, on an average, half of the frequent value occurrences are preceded by frequent value occurrences while the other half are preceded by nonfrequent values. The base FV encoding scheme reduces the

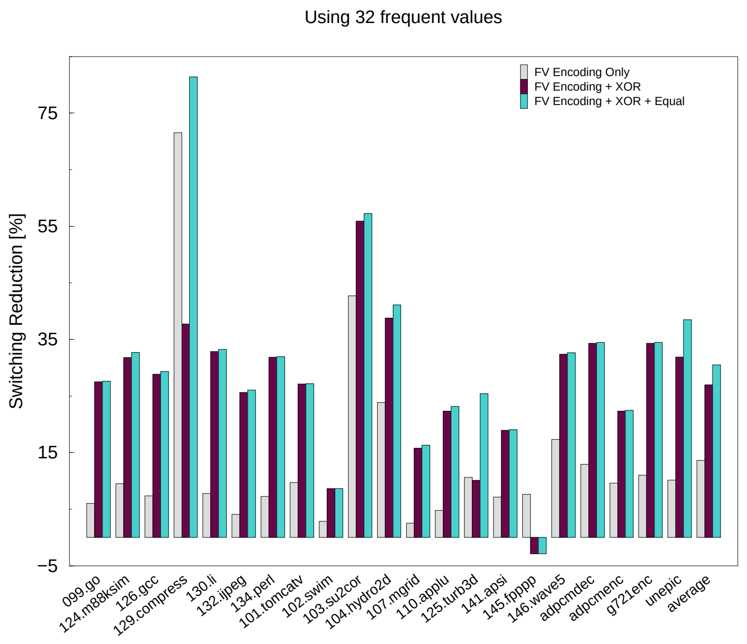


FIGURE 6.6. Effectiveness of FV encoding and its enhancements.

switching for the former category of frequent value occurrences while the XOR reduces switching for the latter category of frequent value occurrences.

The complete encoding algorithm does outperform the above configurations. On average, it achieves 30% reduction in switching activity. Therefore, overall, the equality test reduces switching by a small additional amount. However, for some benchmarks the equality test is crucial for obtaining good performance. For the `compress` and `turb3d` benchmarks the equality test provides a significant increase in performance because a sequence of equal values is encountered very frequently. In fact, as can be seen, the switching reduction obtained using the final configuration is more than twice that of the reduction achieved using the second configuration. In fact, in both these cases using the FV encoding scheme alone gives better performance than additional XORing of values.

Next let us consider the impact of excluding frequent values based upon hamming distance between the frequent values and their encoding. The following three pair of configurations of the encoding algorithm were considered:

1. *FV Encoding vs. FV Encoding + Exclusion* – This is the comparison of the base FV encoding scheme with and without exclusion of values 0 through 16 as candidates for frequent values (i.e., these values are never added to the frequent value table).
2. *FV Encoding + XOR vs. FV Encoding + XOR + Exclusion* – This is the base FV encoding algorithm enhanced with XORing of values. The two versions compared are the ones with and without exclusion of values 0 through 16 from the frequent value table.
3. *FV Encoding + XOR + Equal vs. FV Encoding + XOR + Equal + Exclusion* – This is the base FV encoding enhanced with both XORing of values and equal test. The two versions compared are the ones with and without exclusion of values 1 through 16 from the frequent value table. Recall that the equal test requires hardwiring the value 0 into the frequent value table. It is for this reason only values 1 through 16 are excluded from the frequent value table.

Figure 6.7 shows that the reduction in switching activity is slightly improved for the first two algorithms. This is because some values that now reside in the frequent value table more often replace small values with very few high bits. However, the performance of the third algorithm is unchanged. That is, the FV encoding algorithm enhanced with XORing and equal test performs equally well with or without exclusion of values. In the remainder of this section, for all experiments involving measurement of reductions in switching activity, I use the FV encoding algorithm with XORing and equal test as the basis for experimentation.

**Performance Without On-chip Cache.** In all of the experiments described so far it is assumed that there is an 8K byte on-chip instruction and an 8K byte on-chip data cache. The experiments are also repeated without on-chip caches. The architecture is sketched in Figure 6.8. This is because in many embedded and DSP

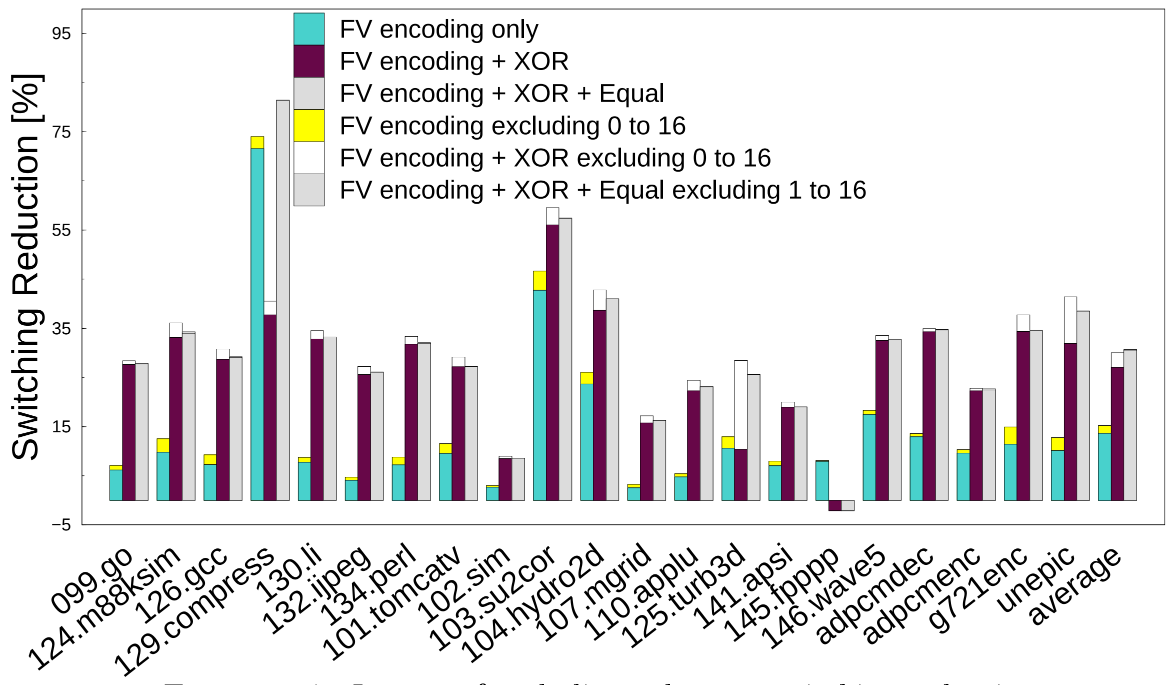


Figure 6.7. Impact of excluding values on switching reduction.

processors from AT&T Microelectronics, Motorola, Zilog and Texas Instruments there is no on-chip cache. The results in Figure 6.9 show that in the absence of an on-chip cache the reductions in switching activity are even greater. The average reduction for 32 values increases from 30% to 49%. The performance improvement is due to the data locality within cache lines, which was caught by instruction/data caches and is now being exploited by the encoder and decoder.

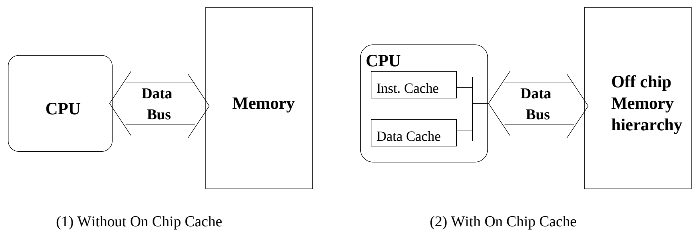


Figure 6.8. Architecture models with and without on-chip caches.

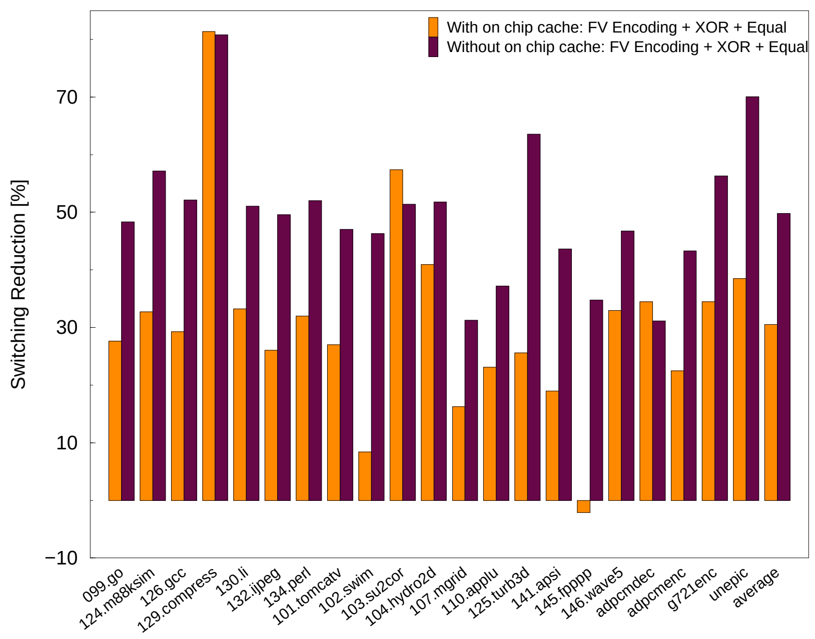


FIGURE 6.9. FV encoding performance with on-chip cache versus without on-chip cache.

## 6.2 Reducing Coder Overhead

The on-chip overhead of performing encoding and decoding is dominated by the accesses to the frequent value table, which involves associatively searching for the values and in case a value is not found, the frequent value table is updated using the LRU replacement policy. In this section the performance of various algorithms are compared from the perspective of this on-chip overhead.

### 6.2.1 Reducing Access to the Coder

Two of the six variations of encoding algorithms that were considered in the preceding section, namely the base FV encoding scheme and FV encoding with XORing, access the frequent value table for each value that is transmitted over the bus. However, the remaining four algorithms which use the *equal test* or *exclusion of values* or both avoid accesses to the frequent value table. This is because in both of these cases the frequent value encoding and decoding processes is bypassed. I measured the reduction in accesses to the frequent value table that these four algorithms achieve over the other two algorithms. The results of this study are presented in Figure 6.10. As can be seen, these reductions are substantial. The exclusion of values significantly reduces the accesses performed by base FV encoding and FV encoding with XORing. The reduction in accesses due to the equal test is generally less than that achieved by the exclusion of values 0 through 16 for these algorithms. However, for the `compress` and `turb3d` benchmarks the equal test reduces the accesses to the frequent value table dramatically.

From the perspective of reducing switching activity on the data bus the two algorithms that perform equally well are ones which uses all of the techniques (i.e., FV Encoding, XORing, equal test, and excluding values 1 through 16 from frequent value table) and the one that uses all techniques except that of excluding values. However, as expected, the former algorithm performs, on average, slightly fewer accesses to the frequent value table than the latter algorithm. Therefore it can be concluded that the algorithm which uses all of the techniques, that is, FV encoding, XORing, equal test, and exclusion of values is best overall.

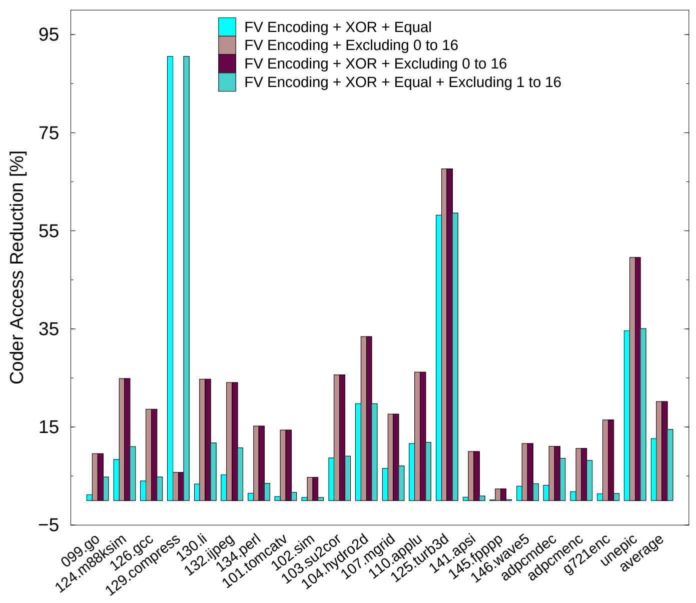


FIGURE 6.10. Impact of excluding values on encoding/decoding operations.

### 6.2.2 Reducing Coder Updating Activity

As mentioned earlier, an alternative to dynamically identifying frequent values is to identify them first during a profiling run and then use these fixed values during all future program runs. This fixed frequent value set approach avoids spending energy on updating the frequent value table. I compared the reduction in switching that can be obtained using fixed frequent values with that obtained using the dynamic algorithm described in this chapter. The results are presented in Figure 6.11. As can be seen, the reductions using dynamically detected frequent values is significantly greater. On average, using the enhanced dynamic FV encoding scheme 30% reduction in switching activity is obtained while with a fixed FV encoding the reduction is only 18%. Moreover for several of the benchmarks, including `su2cor`, `hydro2d`, `fpppp` and `wave5`, the difference is dramatic. This is because for those programs, there are a lot of values that are frequent for only a short time and those values can be captured by the dynamic algorithm but not the fixed algorithm. Therefore the dynamic approach to reducing on-chip overhead is not very attractive.

### 6.2.3 Reducing Coder Size

**Varying Number of Frequent Values.** I also investigated the effect of the encoder and decoder size on performance by varying the number of frequent values allowed. The FV encoding algorithm can be applied to the entire data bus width if the maximum reduction in switching is desired. It can also be applied to a subset of bus wires when minimum hardware expense is demanded. Minor changes are needed when only a subset of bus wires are encoded. For example, assume that only the first 8 bus wires are involved in encoding. Both the encoder and the decoder have only 8 entries. A full value is taken into the encoder and if it is encoded successfully, a code is sent out with respect to those 8 wires. The remaining wires always carry a zero for encoded values. If the value is not encoded, the original value is sent along the bus

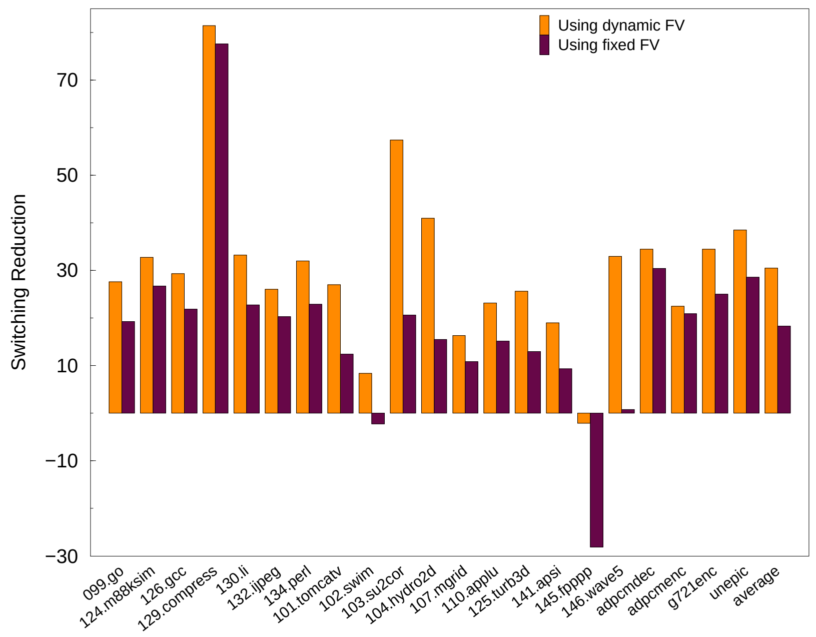


FIGURE 6.11. Dynamic frequent values versus fixed frequent values.

and the coders update their content accordingly. The receiver side can resolve both cases in the same way as before without confusion.

The number of entries in the coders were varied as 8, 16 and 32. The results are given in Figure 6.12. In some benchmarks, including `tomcatv`, `su2cor` and `wave5`, it can be clearly seen that reductions in switching activity increase significantly with an increase in allowable number of frequent values. In other benchmarks the vast majority of reductions can be achieved simply by using 8 frequent values. The average reduction increases from 23% for 8 values to 30% for 32 values.

**Byte Level Encoding.** So far the discussion on encoding and decoding schemes is based on word level frequent values. In many multimedia benchmarks where data is operated in unit of bytes, *frequent byte values* may be more abundant than *frequent word values*. FV encoding can easily adapt to handle frequent byte values so that it is friendly to multimedia benchmarks as well. To do this, I simply encode each byte in a 32-bit word independently. The 32 entries in the frequent value table can

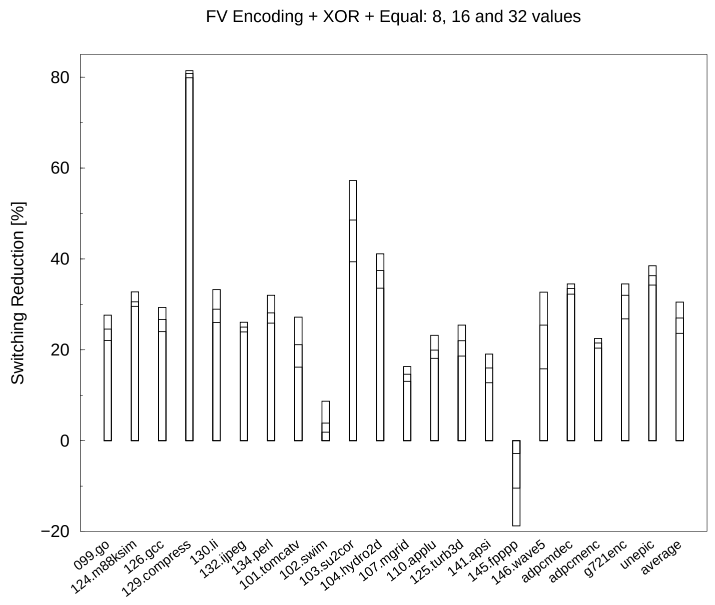


FIGURE 6.12. Varying number of frequent values.

be distributed among the four byte positions, that is, 8 frequent byte values are maintained corresponding to each byte position. In other words, the original word level encoder and decoder are broken into 4 byte level encoders and decoders, each with its own decorrelator and correlator.

The performance of frequent byte encoding as compared to frequent value encoding depends on program characteristics. For example, if more frequent bytes are found than frequent words, byte level encoding may out perform word level encoding. On the other hand if the benchmark contains mostly frequent words, byte level encoding will hurt performance. This is because the frequent word would now be split into four frequent bytes each of which will require one high bit during its transfer.

The results in Figure 6.13 shows the gain and the loss of using byte level encoding. For a program like `fpppp`, which has a high level of frequent bytes, the performance is dramatically improved using byte level encoding – instead of 5% increase in switching I now observe a 24% reduction in switching. On the other hand, for a benchmark like `compress` for which the word value encoding performs very well, byte level encoding does not perform as well as word level encoding. On average a little improvement of

3% was obtained using byte level encoding.

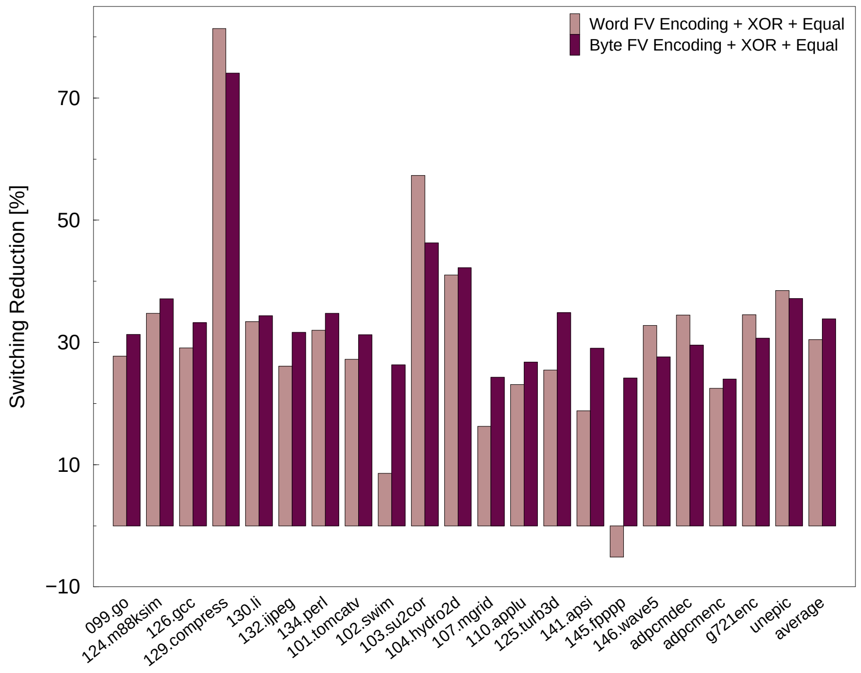


FIGURE 6.13. Byte level frequent value encoding.

## 6.3 Accuracy of Frequent Value Identification in Encoder

### 6.3.1 Approximate LRU versus perfect LRU replacement

The identification of frequent values is based on an approximate LRU policy which uses a timestamp. The size of the timestamp can be varied to achieve different levels of LRU replacement accuracy. Intuitively larger timestamps should provide a better estimation of the least frequently used information and thus perform well during replacement. I performed an experiment in which I compared the encoding rates when using a one bit timestamp (i.e., approximate LRU) with the encoding rates obtained when using an unlimited sized timestamp (i.e., perfect LRU). The results of this experiment shown in Figure 6.14 disproved the early intuition as it shows that a timestamp as small as one bit can perform as well as an unlimited timestamp.

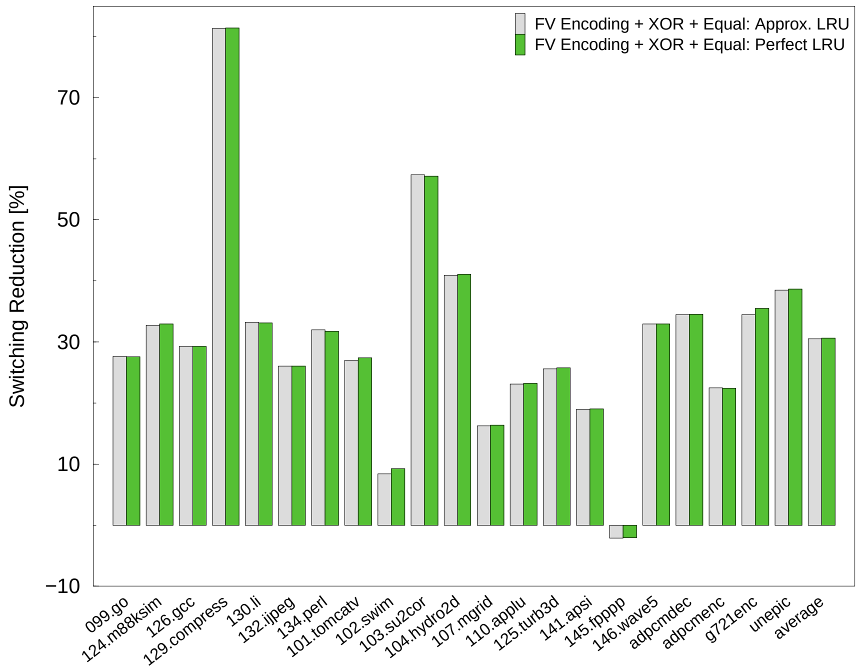


FIGURE 6.14. Comparison with perfect LRU.

The intuition behind the above result is as follows. The data transfered on the data bus between CPU and memory is due to transfer of *cache lines* that contain multiple words. Although these words flow through the encoder and decoder one by one, since they arrive at the bus in close succession, there is little or no difference in their timestamp values. Therefore, even the perfect LRU replacement policy cannot differentiate between these values. Picking any one of them may not actually yield a best result. The result shown in Figure 6.14 proves that a coarse timestamping method is sufficient in practice.

### 6.3.2 Approximate LRU versus optimal replacement

I also conducted another experiment to see how close does approximate LRU comes to optimal replacement. An *optimal replacement policy* is implemented in which I *replace the entry that will not be used for the longest period of time in the future.* The optimal policy will yield the best switching reduction because it will guarantee the highest hit rate in the encoder. To implement this scheme I ran each program twice. The value trace is collected during the first run and used in the second run to carry out optimal replacement. Every time I need to perform a replacement, I go into the value trace to find the frequent value in the table that appears furthest in the future in the value trace. This is a slow process since the value traces were extremely long. Therefore only a subset of the benchmarks (11 out of 22) were tested in this experiment. The results presented in Figure 6.15 show that on an average the switching reduction by the LRU implementation with a one bit timestamp is exceeded by the optimal policy by around 11% which is quite reasonable. This is because the optimal policy is an offline policy and therefore no online policy will be able to perform nearly as well.

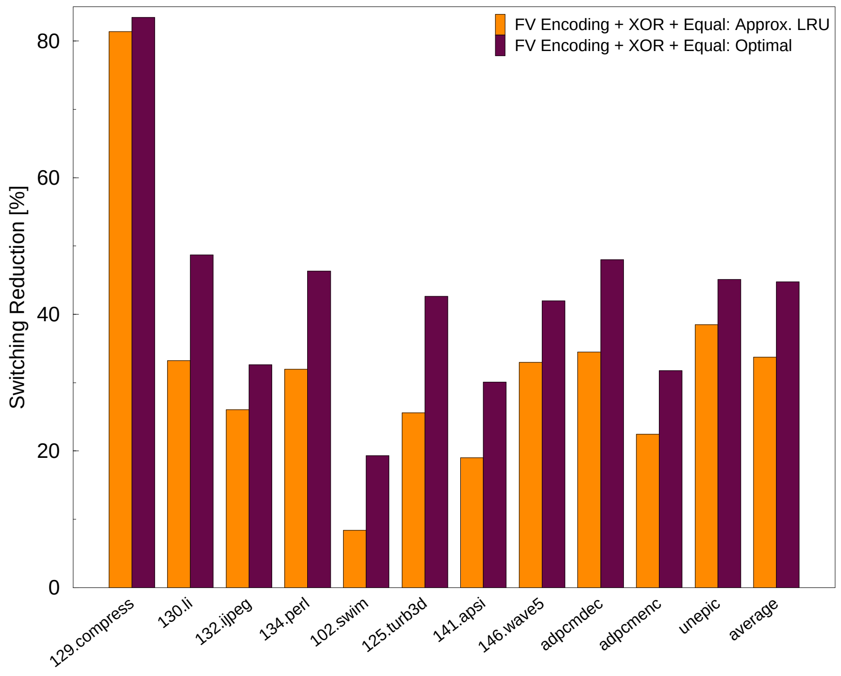


FIGURE 6.15. Comparison with optimal replacement policy.

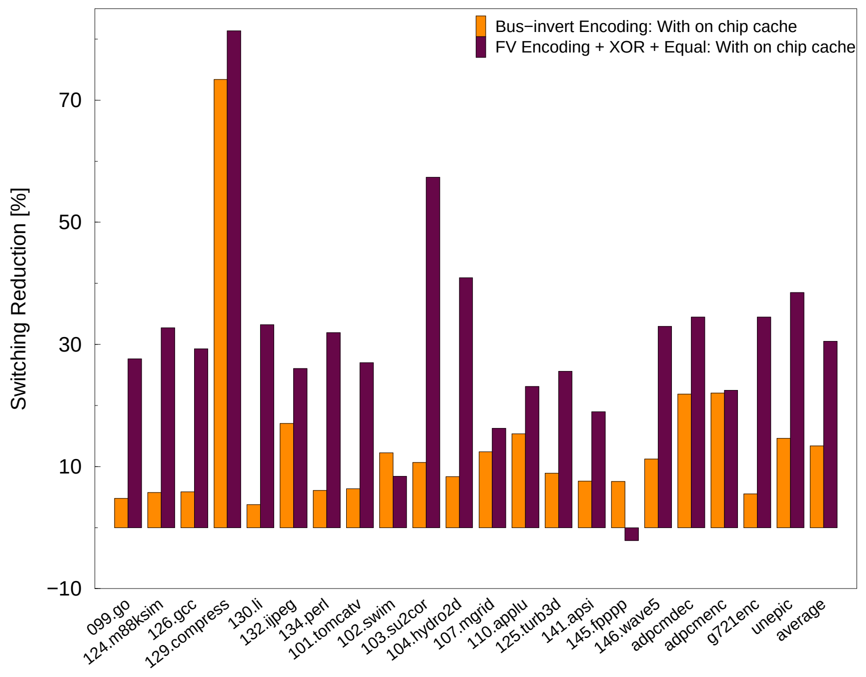
## 6.4 Comparisons with Other Techniques

There has been abundant research done on reducing address bus switching, based either on the sequentiality of program counters [8, 17, 61] or on regularity of memory accesses [43]. The work that applies to data buses falls in two categories: (a) general purpose techniques that apply to both data and address buses; and (b) techniques specifically developed for data buses. Now I compare the proposed technique with techniques in each of these categories.

A well known general technique for reducing switching is the bus-invert coding scheme. In this scheme the Hamming distance between the present bus value and the next value is computed. If this is greater than half the number of total bits, then the data value is transmitted in inverted form. An additional bit, the invert signal, is also sent to indicate how the data is to be interpreted at the other end. This technique was also implemented to compare its performance with enhanced FV encoding for data buses. The results in Figure 6.16 shows that on an average bus-invert scheme reduces switching by 13.4% and 9.6% in presence and absence of on-chip cache respectively. In contrast the enhanced FV encoding with 32 dynamic values reduces switching by 30.5% and 49.8% in presence and absence of on-chip cache respectively. Thus, the enhanced FV encoding scheme provides 2 to 4 times greater reduction in switching than bus-invert coding method.

## 6.5 Summary

In this chapter it has been demonstrated that by exploiting the characteristic of frequently transmitted values, one can design the FV encoding scheme to reduce the switching activity on an external data bus substantially. The reductions are even greater for processors without on-chip caches. Furthermore it has been demonstrated that the frequent values at any point during execution can be effectively identified using a simple hardware mechanism. The online frequent value identification algorithm



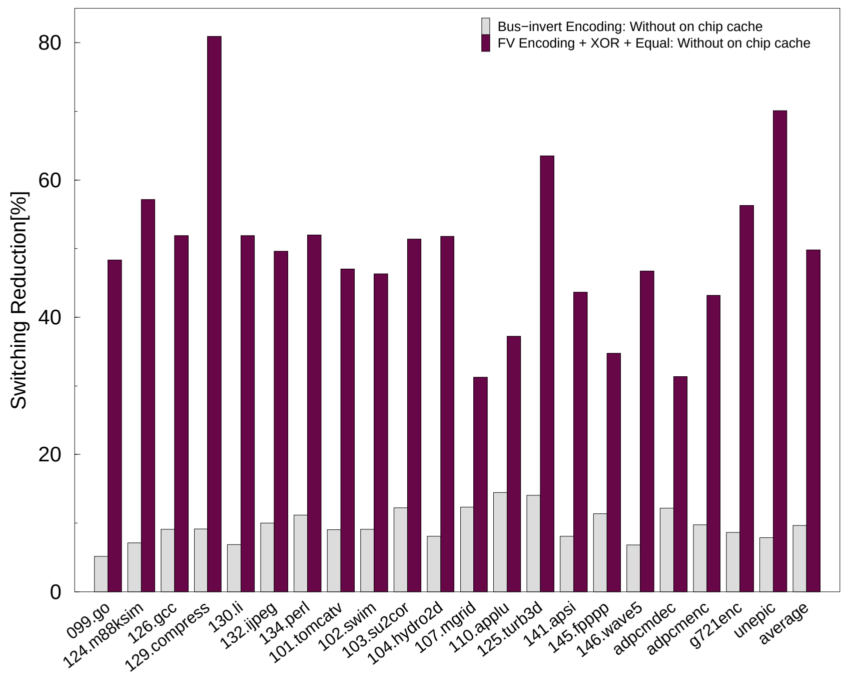
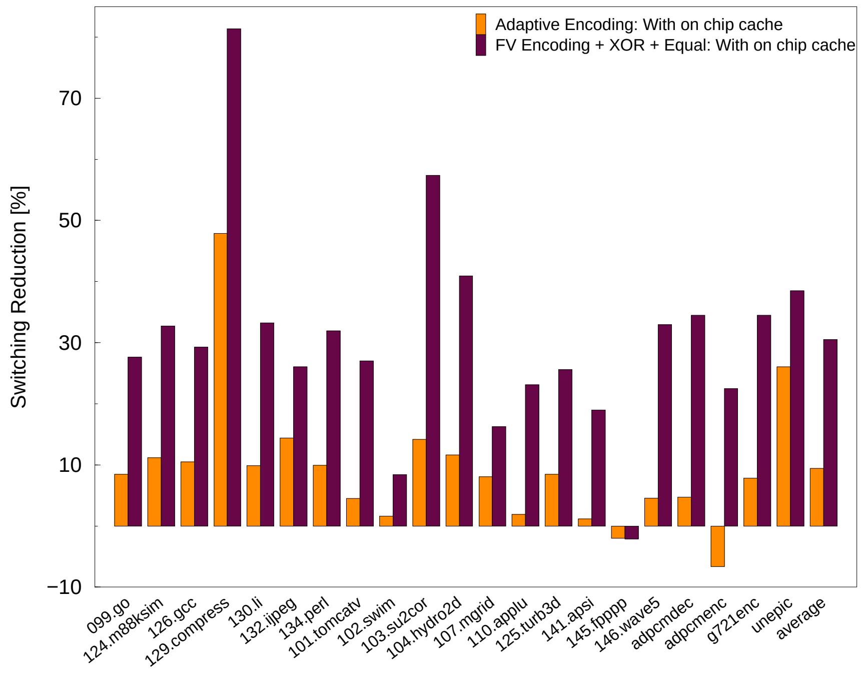


FIGURE 6.16. Bus-invert versus enhanced FV encoding.



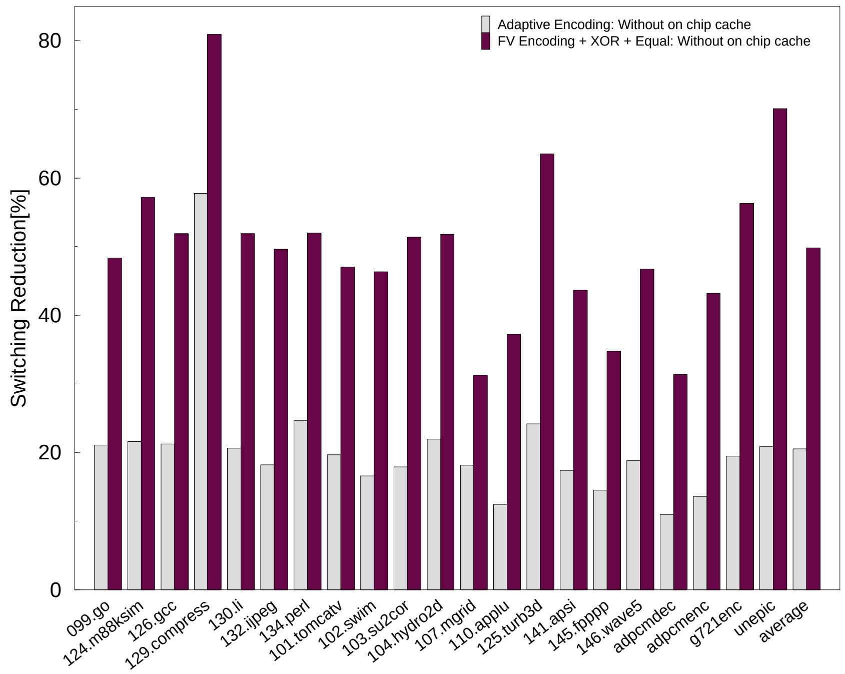


FIGURE 6.17. FV versus adaptive encoding.

compares quite favorably with an offline optimal algorithm. By allowing the set of frequent values to change during execution I obtain reductions in switching that are substantially greater than reductions achieved by a scheme that uses a fixed set of frequent values for the entire execution. Finally it has been demonstrated that FV encoding outperforms both bus-invert coding [60] and the adaptive scheme of [7].

# Chapter 7
# Computation Reuse

It is well known that, one source of chip energy consumption is the data cache which continues to grow in size and area with each new generation of high performance processors. Chapter 5 has proposed a design that reduces energy for average cache access. In this chapter, another method that aims to reduce the number of accesses directed to cache by providing a small and simple structure which intercepts what is called "reusable instructions" before they go into the cache.

An opportunity has been identified for reducing cache activity which can lead to reductions in energy consumed by the cache as well as improvements in executed instructions per cycle (IPCs) for a class of programs with significant levels of load and store reuse opportunities. In particular, it is observed that even programs compiled with optimization contain very high levels of load and store reuse opportunities. For example, in `SPEC95` benchmarks compiled using `gcc` with the `-O3` option average of 31% of the loads load the same value as their last instances and 42% of all stores do not change the memory content. This is a direct consequence of limitations of register allocation and compile-time analysis techniques - specially highly conservative memory disambiguation techniques used by compilers make it necessary to introduce load and store instructions to guarantee program correctness. If this reuse could be exploited to avoid memory references, cache activity as well as address and data bus switching will be reduced leading to lower energy consumption.

Extensive research has been carried out on load and store reuse techniques that are speculative in nature [33, 39]. However, speculative techniques often do not reduce memory activity. This is because such techniques may reference the memory to detect mis-speculation which implies they are not energy efficient. Non-speculative load and store reuse techniques are more appropriate for energy efficient design. For example, the instruction reuse technique developed by Sodani and Sohi [57] when considered in context of load and store instructions can simultaneously improve IPCs and reduce cache activity. This approach is adopted in this chapter. However, it is a challenging task even though this approach will reduce cache activity. This is because a load-store reuse filter for removing memory references will itself consume energy. To get an overall reduction in energy consumed one must save more energy in the cache than is consumed by the reuse mechanism. Experiments with an aggressive reuse mechanism resulted in a net increase in energy used because energy consumed by the reuse mechanism greatly exceeded the savings in cache energy. Also if the program contains little reuse, while the reuse filter will continue to consume energy, little savings in cache energy will result. Therefore the design goal is to minimize the increase in net energy consumed in such situations.

This chapter presents the design of a reuse mechanism which has been carefully tuned to achieve two objectives. The first objective is to develop a reuse filter that is able to capture and eliminate a large fraction of load and store instructions. The second objective is to adjust the reuse filter so that it performs in an energy-efficient manner. In programs with significant levels of reuse opportunities, it is desired to achieve net energy savings while for other programs it is desired to minimize the net increase in energy consumed.

The first objective is achieved by designing reuse hardware which is able to capture reuse opportunities arising from multiple sources. For example, a load can be reused through three distinct categories of memory instructions executed earlier. This will be discussed in Section 7.1.1. The second objective is achieved in two ways. First, the reuse hardware is designed such that the history of previously executed load and store operations is accessed through indexed tables - associative lookups are kept to a minimum. Second, the proposed algorithm is not too aggressive, and in fact if it fails to detect reusability, it backs off from certain types of detections so that the energy consumed by reuse detection hardware can be kept to a minimum. The experiments demonstrate that for programs with high levels of reuse, up to 55% of the IPC improvements and up to 47% of the net cache energy savings is achieved. Even more importantly, in programs where little reuse is found, the net energy loss is held to less than 3%.

A small subset of load and store reuse is also captured by the store buffer. For example, a load may not need to go to memory because it may find the value it requires in the store buffer. Similarly, if multiple pending stores to the same address location are pending in the store buffer, all but the last of these writes need not be sent to memory. However, it should be remembered that detection of reuse in these situations will involve associative searches of the store buffer. These operations are clearly quite energy consuming. In contrast, the reuse hardware that is designed in this chapter mainly involves indexed operations because direct mapped structures are used instead of associative structures. Therefore the proposed approach will not only capture far greater amounts of reuse opportunities than the store buffer, but it is also energy efficient.

The remainder of the chapter is organized as follows. Section 7.1 describes the different forms of load and store reuse and present the result of a study that measured the amounts of reuse in `Spec95` benchmarks. In Section 7.2 I present the hardware design of the reuse mechanism and develop an energy-efficient algorithm for using this mechanism. In Section 7.3 the results of the performance evaluation are presented.

## 7.1 Load and Store Reuse Study

This section presents the results of experimental studies designed to measure the degree of load and store reuse present in programs. Most of all, the study has identified different categories of load and store reuse opportunities and measured the contributions of each of these categories. This study is important not only because it tells how much reuse opportunities are present, but it also guides the design of hardware that focuses on capturing those forms of reuse opportunities that are present in substantial amounts. Therefore the resulting design obtained is *energy-efficient.*

### 7.1.1 Types of Load and Store Reuse Opportunities

The classification of reuse opportunities is shown in Figure 7.1. A load can be avoided using reuse if the value that it loads from an address was loaded or stored from the same address by a prior memory instruction and the contents of that location has not changed since. An instance of a load instruction can be reused because of a prior instance of the *same* or *different* load instructions, or a prior instance of a *store* instruction. In the examples of these situations in Figures 7.1(a-c) if the addresses, and values, involved in these pairs of instructions are the same, the second instruction which is a load can reuse the result from the earlier instruction.

$PC_1$: Load R1, addr
...
$PC_1$: Load R1, addr

(a) *Same* Load Reuse.

$PC_1$: Store R1, addr
...
$PC_1$: Store R1, addr

(d) *Same* Store Resue.

$PC_1$: Load R1, addr
...
$PC_2$: Load R2, addr

(b) *Different* Load Reuse.

$PC_1$: Store R1, addr
...
$PC_2$: Store R2, addr

(e) *Different* Store Reuse.

$PC_1$: Store R1, addr
...
$PC_2$: Load R2, addr

(c) *Store* Load Reuse.

$PC_1$: Load R1, addr
...
$PC_2$: Store R2, addr

(f) *Load* Store Reuse.

FIGURE 7.1. Classification of Load and Store Reuse.

A store is avoidable if the value it writes to a location is already present in that location. The store reuse can be classified as that arising at a previous instance of the *same* or *different* store, and previous instance of a *load*. Examples of these situations are shown in Figures 7.1(d–f). The second instruction, which is a store, need not be executed if the history of the prior load/store instruction is available and therefore it can be determined that the store is writing into the address the same value that the address already contains.

### 7.1.2 An Aggressive Reuse Mechanism

An aggressive algorithm was first implemented which tries to capture maximum possible reuse. A separate table structure was used to save the histories of past loads and stores. The algorithm first used associative searches on the address fields of these tables to determine if a load or store was reusable. This approach allowed it to capture all forms of reuse described above. The sizes of the history tables used were 256 entries each. The outcome of this experiment is shown in Figure 7.2. While significant IPC improvements were observed, more energy is consumed instead of energy savings. This is because the reuse mechanism uses much more energy than what is consumed due to the cache activity.

In order to develop an energy-efficient design the studies were carried out to help design such a mechanism. The first study measured the amount of different types of reuse opportunities so that I could focus on those types which will give most benefit. The next study further guides the selection of table sizes and the decisions relating to the complexity of algorithms for capturing *different* types of load and store reuse opportunities.

### 7.1.3 Amounts of Load and Store Reuse of Different Types

The results of the study of load and store reuse in `Spec95` benchmarks is shown in Figures 7.3 and 7.4. First let us consider the characteristics of load reuse opportu-

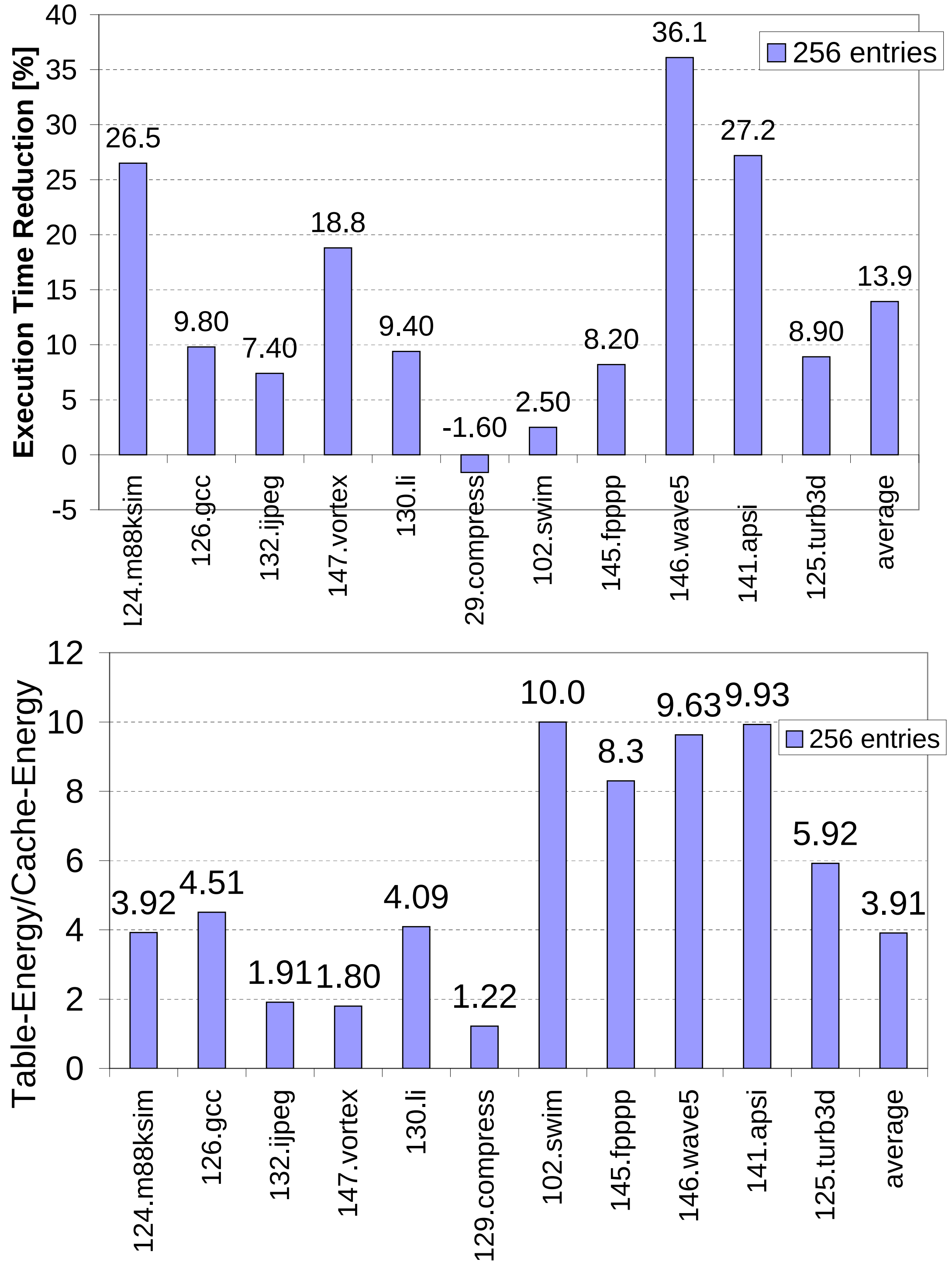


FIGURE 7.2. An Aggressive Reuse Scheme: IPC Improvement and Increased Energy Consumption.

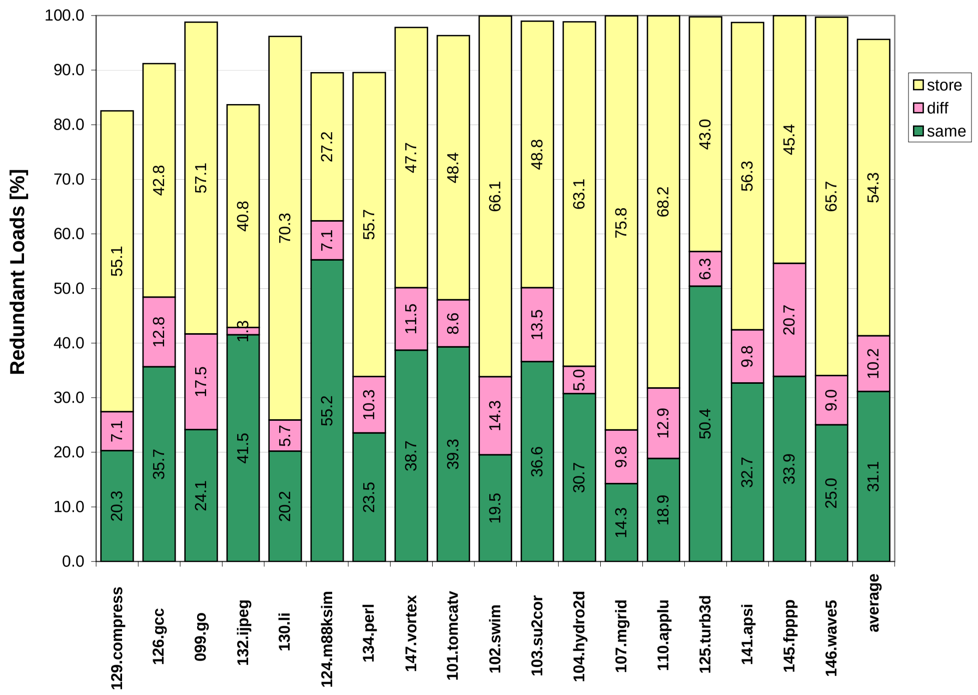


FIGURE 7.3. Load Reuse Characteristics of Spec95 Benchmarks.

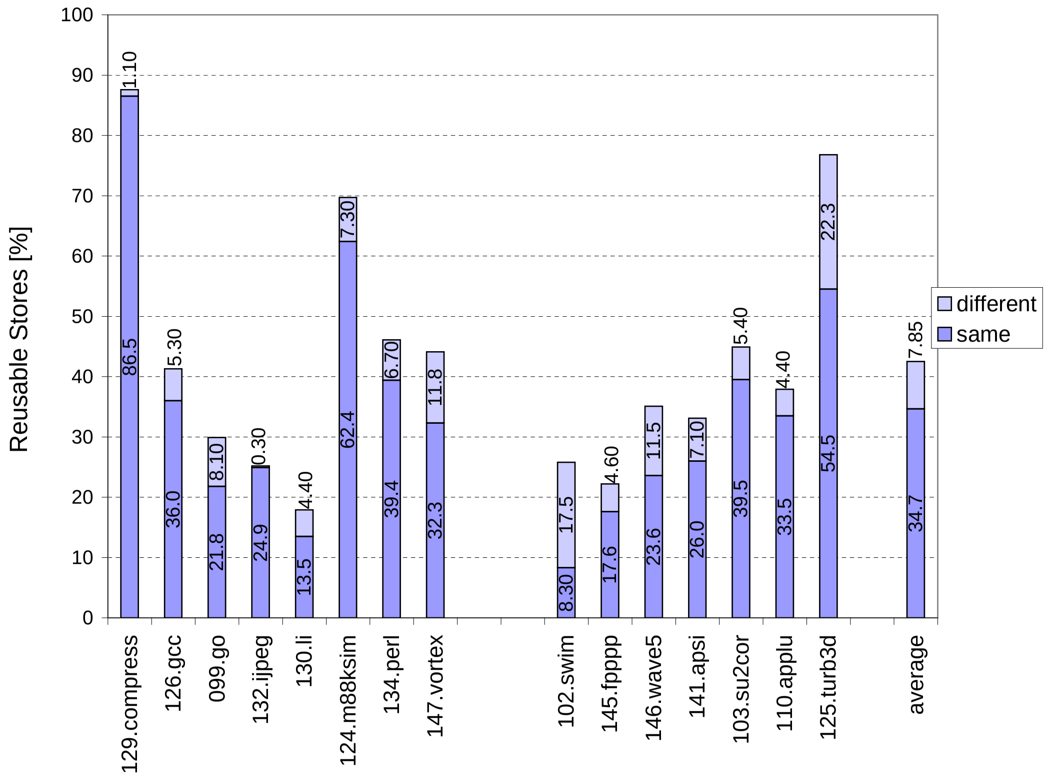


FIGURE 7.4. Store Reuse Characteristics of Spec95 Benchmarks.

nities. A load may be reusable due to multiple conditions in Figure 7.1 being true. For example, a load may satisfy conditions with both a store instruction or its own history instances, and reuse from the latter case is of lower cost. Therefore the reuse detection process is prioritized as the following: it first looks for same load reuse, then different load reuse, and finally store to load reuse. Each load is put only in the first category that it is found to satisfy. The results are shown in Figure 7.3, on average, nearly 95.6% of total executed loads present reusability. Moreover the breakdown in each three categories is substantial. Under this definition nearly all loads are reusable. This is because before a value is loaded it is almost certainly computed and stored at some earlier point in the program. Therefore if a load is not same or different load reusable, it most certainly is store to load reusable. The remaining few percent goes to other reasons such as memory initialization by system calls. Traditionally, reuse techniques have focused on exploiting *same* load reuse. However, as it can be seen that the other two categories of load reuse are also very important. The implementation designed in this chapter exploits all three types of load reusability.

Now lets look at the results of the store reuse study in Figure 7.4. On average, in these benchmarks 42% of all executed stores are reusable. As can be seen, most of the reuse is same store reuse. There is also some reuse due to different stores. However, the store reuse due to loads is almost nonexistent. Therefore, in the further studies in this section only those two types of reuse are considered.

### 7.1.4 History Table Sizes and Linking Parameters

The above experiments have already helped in restricting the attention to those categories of reuses that will provide the most benefit. Next, experiments are conducted to guide the selection of table sizes and the decisions relating to the complexity of algorithms for capturing *different* load/store reuse. The first consideration is different tables sizes to save load and store history since the energy consumed by these tables increases with their size. Experiments found that separate load and store history tables both of sizes of 32 or 64 entries are quite sufficient. If the table sizes are further increased the energy consumed grows much more rapidly than the increase in the amount of reuse. It is also important to design reuse hardware which minimizes high energy consuming associative searches and mainly uses indexed accesses whenever possible. For this purpose, other than the same load/store reuse cases, I explicitly link the two instructions involved in the reuse. An instruction may be able to reuse results from many different instructions. Thus it is unclear how many links need to be allowed. Again creating more links will cause increased energy consumption both in their creation and use.

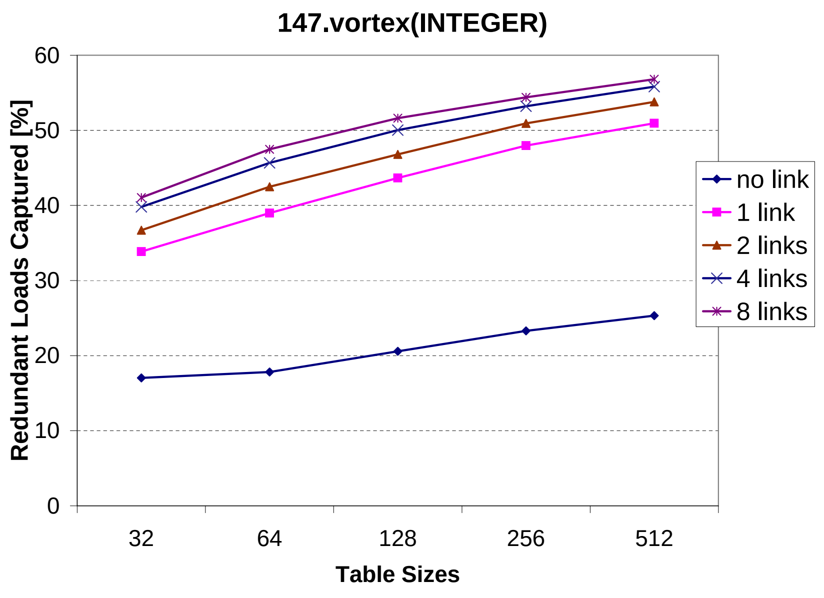
Figures 7.5 and 7.6 show the tests on two benchmarks, one in `SPEC95INT` and one in `SPEC95FLOAT`. It was found through these experiments that linking a load with one other load is sufficient—linking it with more loads finds only small amounts of additional reuse. The benefit of linking a store to different prior stores is also evaluated and it turned out that providing such links is not very useful.

## 7.2 Reuse Hardware and Update Algorithms

This section presents the design of the *Reuse Unit* (RU), the hardware structure for detecting and filtering load and store instructions. The various pieces of the reuse unit are not always needed by every memory instruction. The algorithm selectively prevents their use to reduce activity and save energy if it determines that their use will not result in additional load and store filtering.

### 7.2.1 The RU Structure

In Figure 7.7 the high level block diagram of the RU is first shown in the upper half. Next I describe the various structures in the RU and then later I describe how it is maintained through different operations plotted in the bottom half.



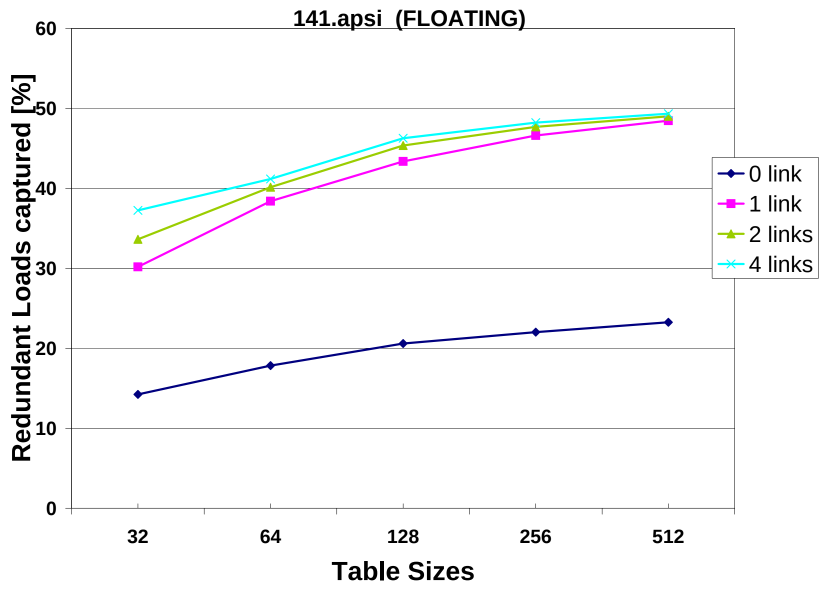
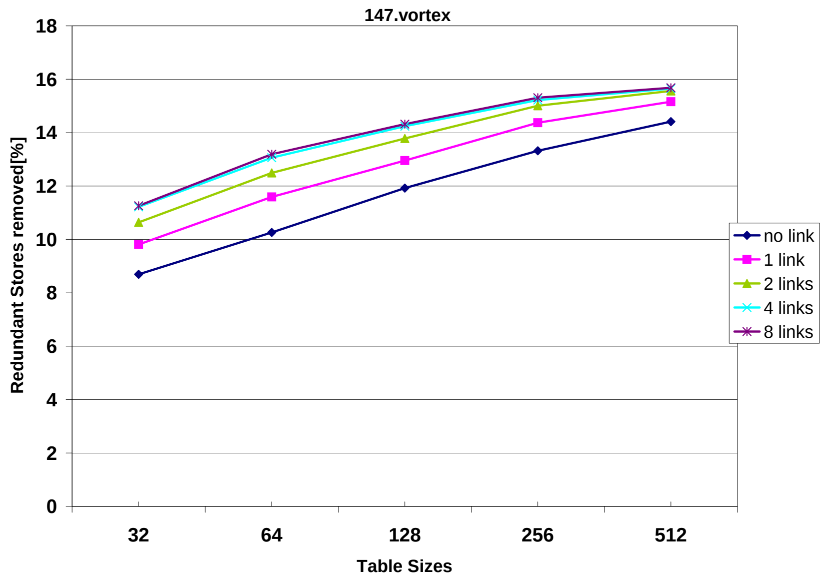


FIGURE 7.5. Impact of Table Sizes and Number of Links on Load Reuse.



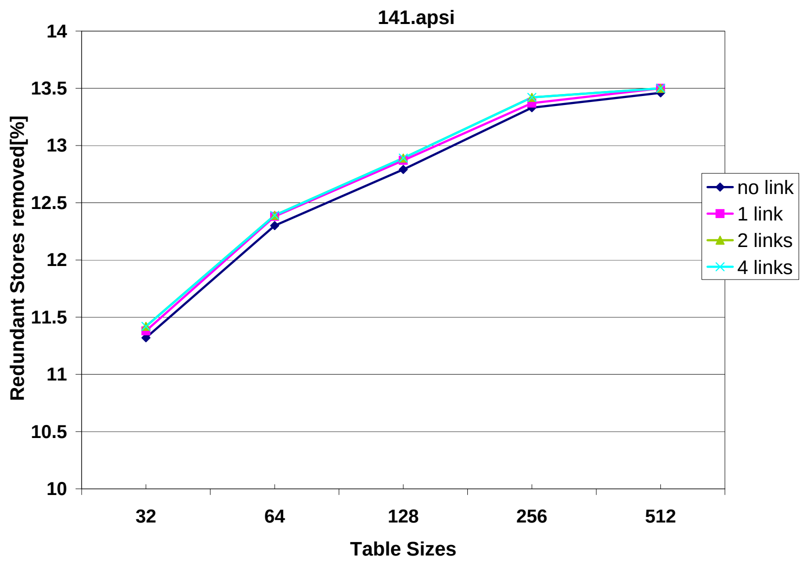
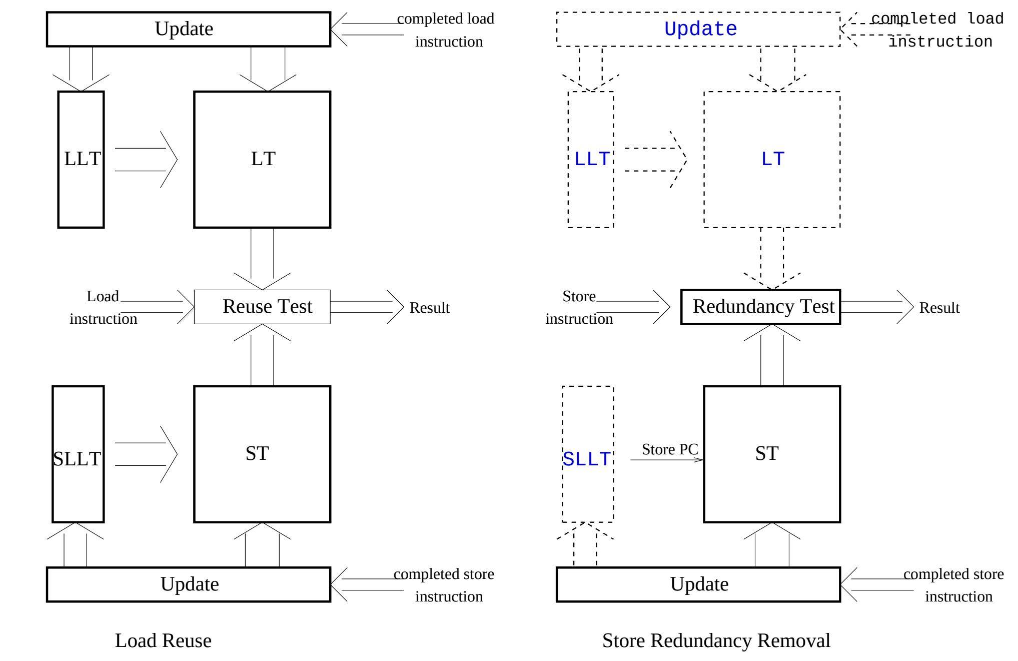
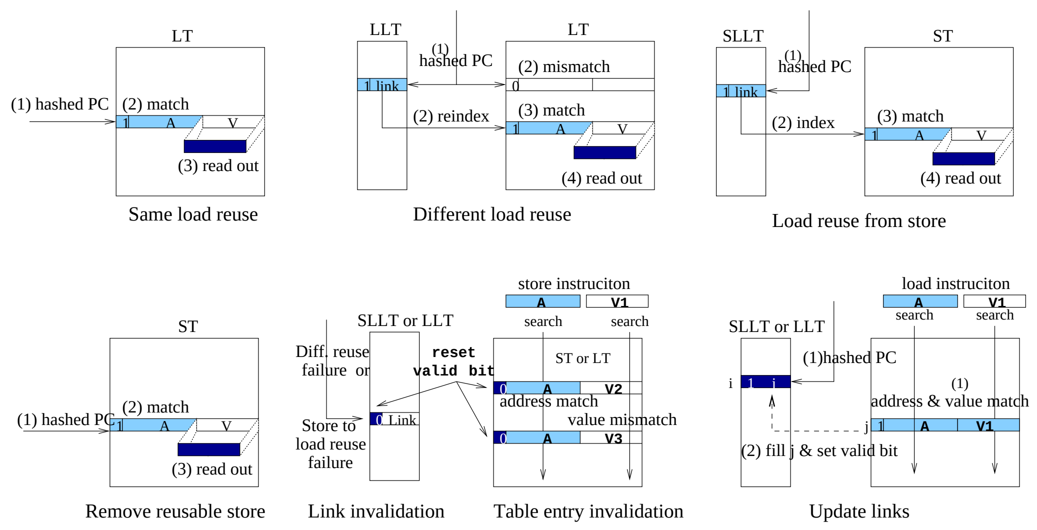


FIGURE 7.6. Impact of Table Sizes and Number of Links on Store Reuse.



(a) Block Diagram.



(b) RU Operations; A stands for address, V stands for value

FIGURE 7.7. The Reuse Unit (RU).

### LOAD TABLE (LT)

LT stores histories of loads that have already been executed. It is designed as a small indexed array so that it can achieve low access times and energy. Each entry contains the effective address, data value at that address, and a mem-valid bit. The mem-valid bit flags the validity of the current contents of the entry. The entry is invalid if nothing has been stored in it or if the value stored in the entry is stale.

### LOAD LINK TABLE (LLT)

This table is needed to capture *different* load reuse opportunities. It is also an indexed structure which provides a link between a load and another load whose results can be reused. The link is in the form of an LT index corresponding to the load at which the reuse originates. The LLT contains the same number of entries as the LT and there is a one-to-one correspondence between the LT and LLT entries. An additional bit in each LLT entry is used to indicate the *usability* of the link stored in that entry. A link marked unusable is not used to index the LT. This bit helps in reducing energy consumed by eliminating unnecessary look ups of the LT.

### STORE TABLE (ST)

ST is the counterpart of LT - it stores the history of past stores that have been executed. It serves two purposes. First, it provides a means for detecting store to load reuse. And second, it helps in the detection of same store reuse. It is also a small indexed array. The ST contains fields for the memory address, its value, and the mem-valid bit.

**STORE to LOAD LINK TABLE (SLLT)**

This table aids in detecting store to load value reuse. It basically links each load in LT with an entry in ST. Like the LLT, there is also a single bit corresponding to every link which indicates whether the link is usable or not.

### 7.2.2 Reuse Procedure

Now it is time to describe the complete reuse mechanism. A load's PC is hashed into an LT entry in order to attempt same load reuse. If this entry is valid, the address stored in the table and the load's effective address are compared for reusability. The load can use the value stored in the entry if the address comparison succeeds. The load's PC is hashed into an LLT entry for different load reuse. If the link in LLT is marked as usable, the corresponding entry pointed to by the link will be tested in the same manner as the same load reuse test. If the load passes this check it has succeeded in different load reuse. Otherwise, the load's PC is hashed into SLLT to attempt store to load reuse.

Filtering of stores is carried by simply using the ST history. Filtering of a store means it is not sent to or deleted from the store queue and therefore it never reaches the cache.

On completion of each load/store instruction, the LT/ST table is updated by setting the address and value fields and also setting the mem-valid bit. LLT links are set up only if they are needed, that is, only if same load reuse has failed I create these links to attempt to find different load reuse. Similarly the SLLT link for a load is set up only if the load fails to find different load reuse. The LLT and SLLT links are set up following an associative search of the address fields in LT and ST. A link is set to be usable when it is first created. It remains set if it is used successfully, that is, its use results in filtering. A link is marked unusable when the reuse test using that link fails.

*The above strategies of creating an LLT link only if same load reuse fails, creating an SLLT link only if different reuse fails, and marking links as unusable if their use fails to detect reuse all greatly reduce RU activity and were therefore extremely important in obtaining an energy efficient design.*

### 7.2.3 Integration into Superscalar Pipeline.

The reuse mechanism is incorporated into a superscalar pipeline as shown in Figure 7.8. An extra stage, the load store reuse (LSR) stage, is introduced immediately preceding the data cache access stage. This stage uses the RU to determine if the load or store instruction is reusable and therefore need not be sent to the cache. If the instruction is not found to be reusable because all reuse tests fail, it is sent to the cache. Therefore the load/store instructions that are not reusable pay an extra penalty of one cycle in this implementation. The ones that are found to be reusable benefit because they are completed in one cycle.

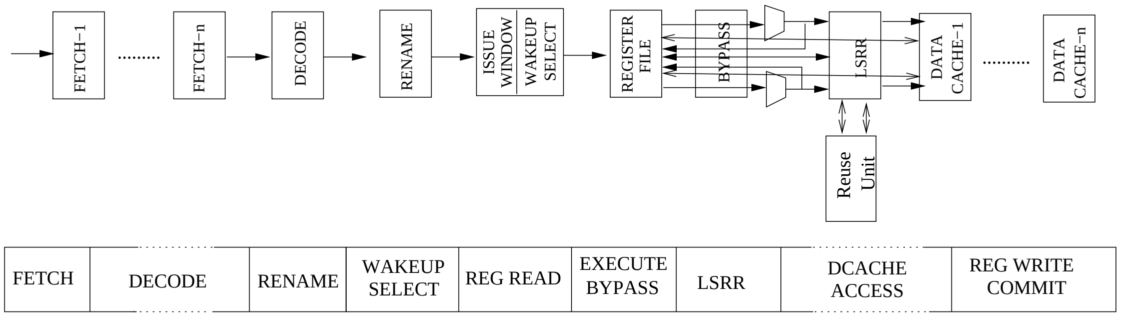


FIGURE 7.8. Superscalar Processor with Reuse Hardware.

## 7.3 Performance Evaluation

The techniques described in this chapter have been implemented and the experimental setup used is described. The experiments are based on an 8-issue superscalar processor with the pipeline structure shown in Figure 7.8. The baseline processor has one less stage since it does not carry out load and store filtering. Therefore if load/store reuse always fails, the processor will take one more cycle than the baseline processor for each data cache access. The data cache is a 32 Kb direct-mapped cache with 32 byte line size and 6 cycle miss penalty. There are 2 read and 2 write ports to the data cache.

The table sizes used in these studies are 32 and 64 for both load and store history tables. The cache energy models were obtained from the SimplePower [75] simulator and the energy models for the reuse hardware were implemented using the models for array and CAM structures used in Wattch [10], both using 0.8 micron technology.

Experiments were run where the number of cycles for data cache access was varied—2 and 6 cycle accesses were considered. This was motivated by the observation that future generation processors are devoting increasing number of stages to data cache access due to reduced cycle times which are too small to perform cache accesses in a single cycle. All data except the IPC improvements are presented assuming that data cache access takes 2 stages. This is because all other data shows very small changes when the number of stages is changed.

Now let us examine the results obtained. First, the degree of reuse captured is shown. Second, the reduction in cache activity in terms of references that go to the data cache and the amount of address and data bus switching is presented. Third, the energy savings in the cache and the energy used by the reuse hardware is also presented. Based on this the net cache energy savings are computed. Finally the IPC improvements are presented.

### 7.3.1 Loads and Stores Reused

On average over all the benchmarks nearly 25% and 31% of all loads from a table of 32 and 64 entries were eliminated by the proposed mechanism (see Figure 7.9). This number is quite substantial and most importantly it can be seen that all three types of reuse play an important role in the overall performance. On average nearly 12% of stores were also eliminated.

As expected, the reuse opportunities captured is lower than the figures observed in the ideal study. This is because the ideal study did not take into account the timing issues. It was assumed that an instruction could always benefit from a prior instruction. However, in a superscalar processor the prior instructions may not have completed execution and thus reuse may not be captured. In addition, limited table sizes, the delays in establishing links, the limitation on the number of links, etc. all contribute to the loss.

### 7.3.2 Cache Activity Reduction

The successful filtering of load/store instructions reduces cache related activity in two ways: first the read and write accesses to the cache are reduced and second the bus activity of the data and address busses is reduced. From the results in Figure 7.10 it can be seen that the number of cache accesses are reduced by nearly 15% and 18% when history tables of sizes 32 and 64 entries are used. The address bus switching is reduced by nearly 15% and 19% while the data bus switching activity is reduced by 22% and 28% for the 32 and 64 entry tables respectively. Therefore the load/store filtering yields a significant reduction in cache related activity.

### 7.3.3 Energy Savings

The energy consumed by the cache with and without reuse hardware were both measured. The energy consumed by the reuse hardware was measured as well. In Figure 7.11. the first graph gives the reduction in energy consumed by the cache. As can be seen the reductions are substantial and on the average 15% and 18% reductions were observed for table sizes of 32 and 64 respectively. The second graph gives the energy consumed by the reuse hardware as a percentage of energy used by the cache.

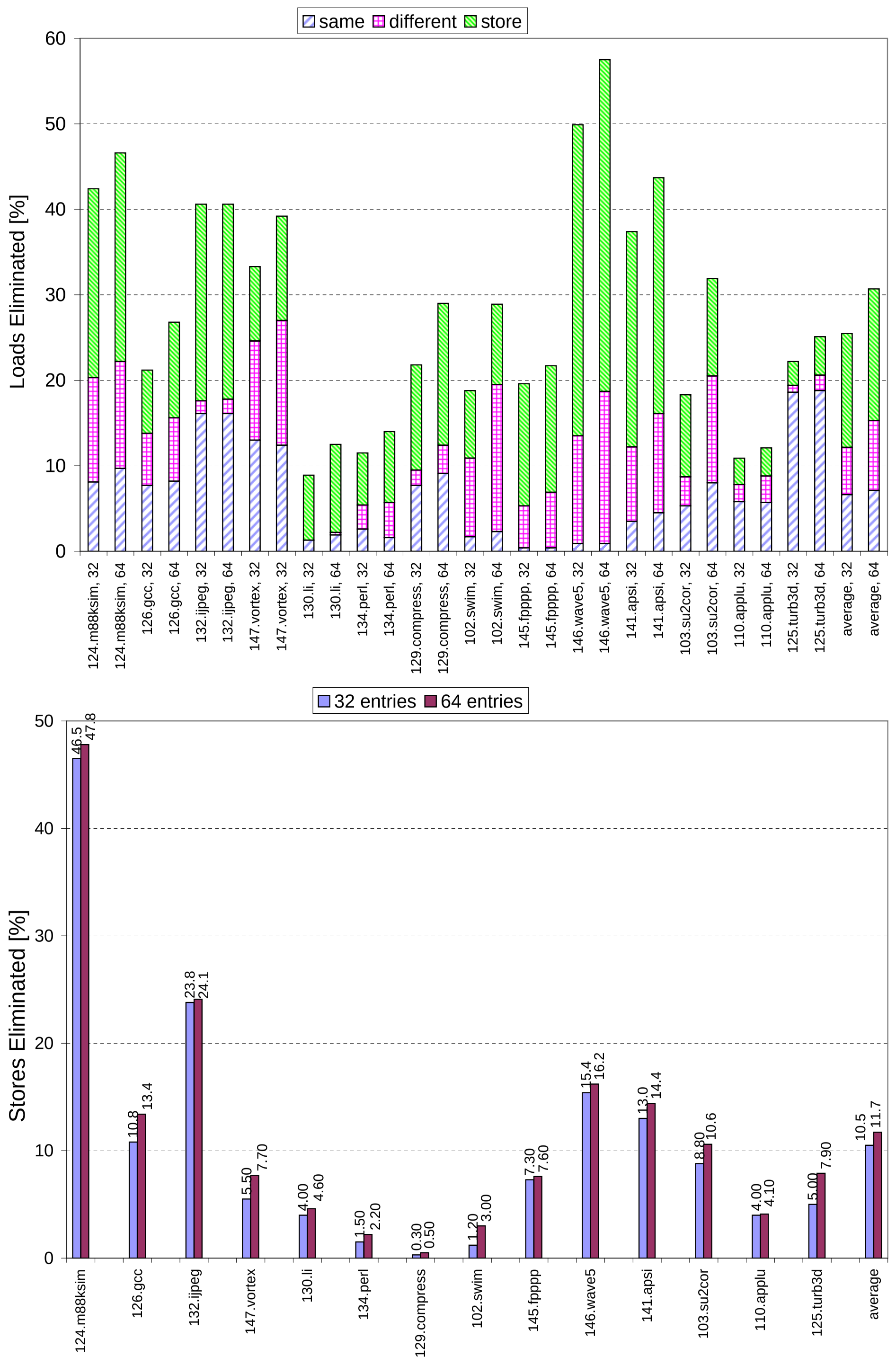
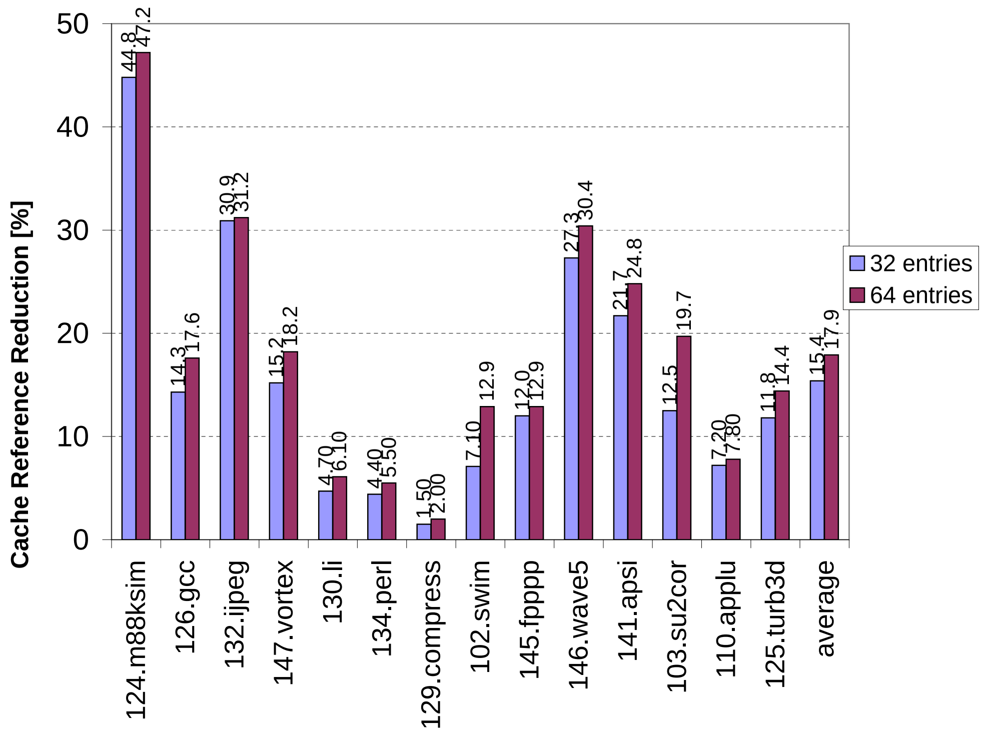


FIGURE 7.9. Reuse Opportunities Captured.



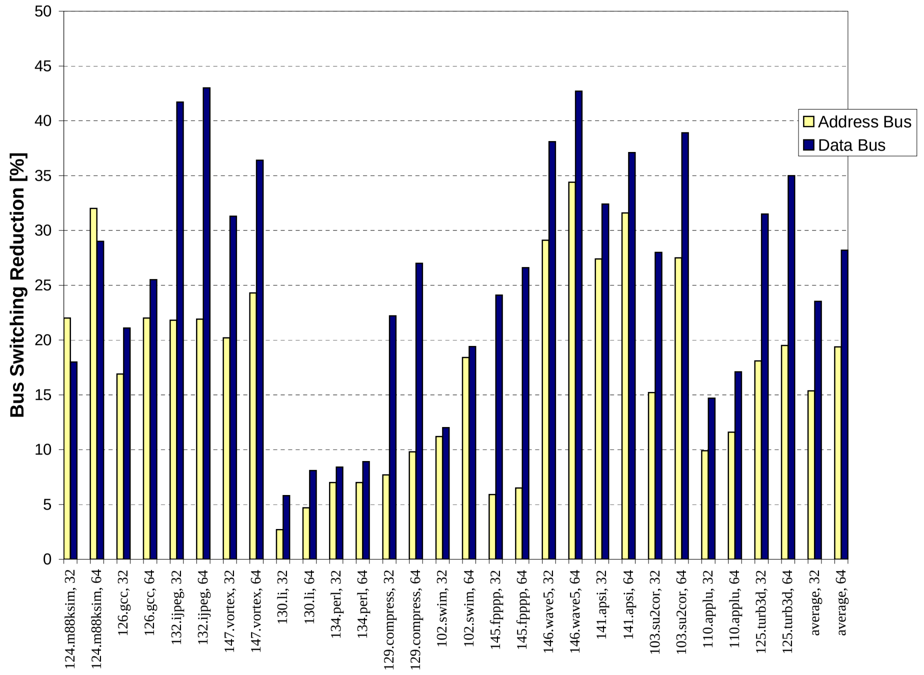


FIGURE 7.10. Reduction in Cache Activity.

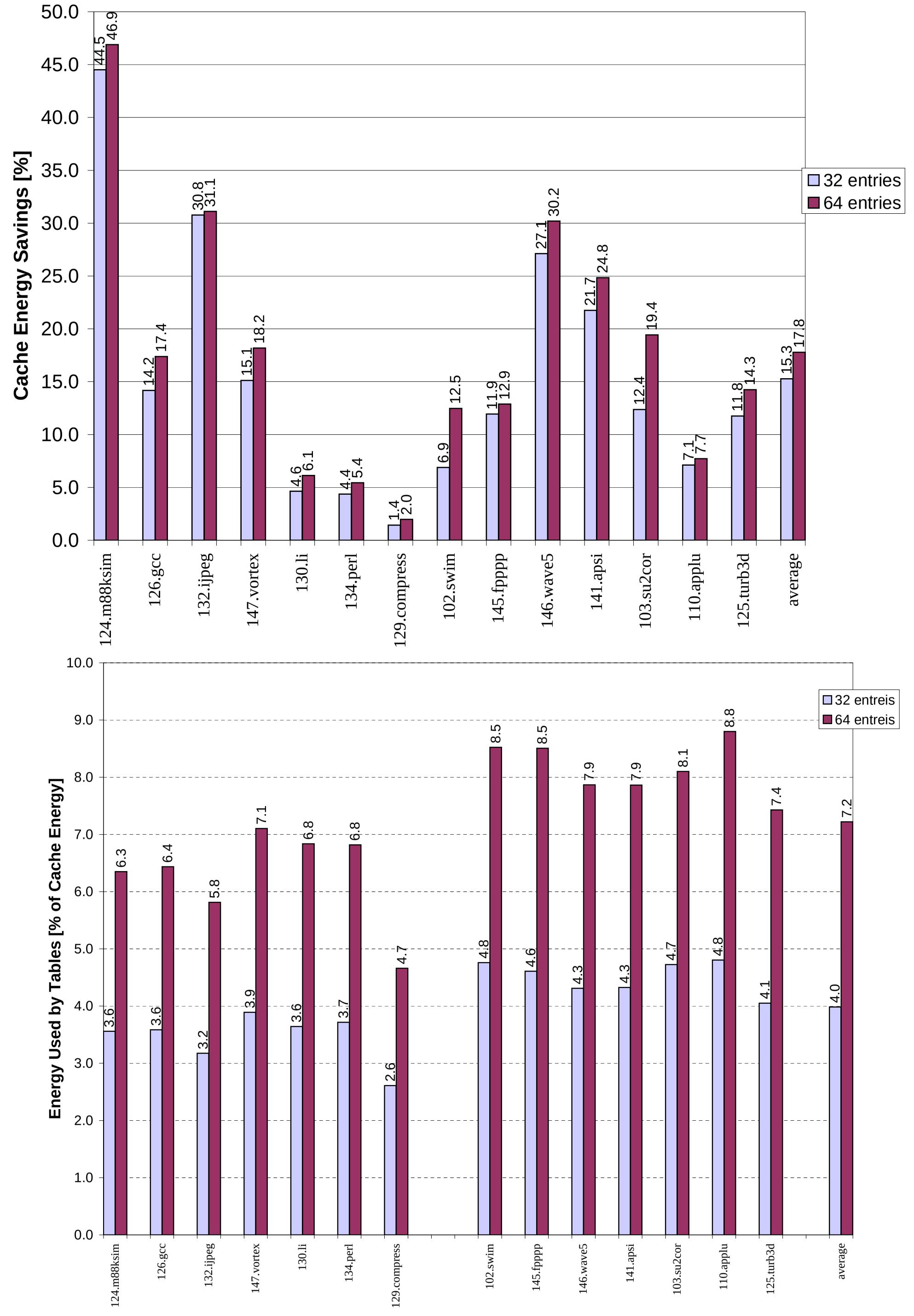


FIGURE 7.11. Energy Figures: (a) Cache Savings; (b) Reuse Hardware Consumption.

As one can see, the reuse hardware design is quite energy efficient as the 32 (64) entry table consumes energy that is only 4% (7%) of the energy consumed by the cache.

The net savings in cache energy is defined as the difference between the cache energy savings and the energy consumed by the reuse hardware. The net energy savings are plotted in Figure 7.12. As can be seen the savings are substantial ranging from a few percent to as much as 47%. For benchmarks with low levels of reuse with table size of 64 there is a net loss in energy. However, the loss is less than 3% in these cases. Thus the reuse hardware designed can provide substantial energy savings when high levels of reuse is captured and very little increase in energy is observed when little reuse is found.

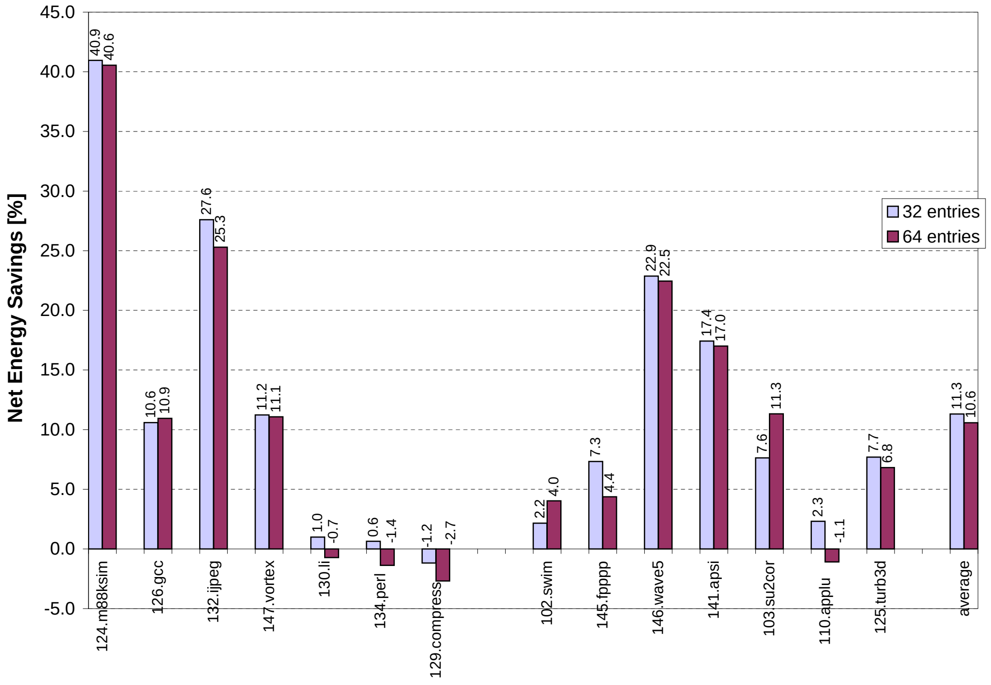


FIGURE 7.12. Net Energy Savings.

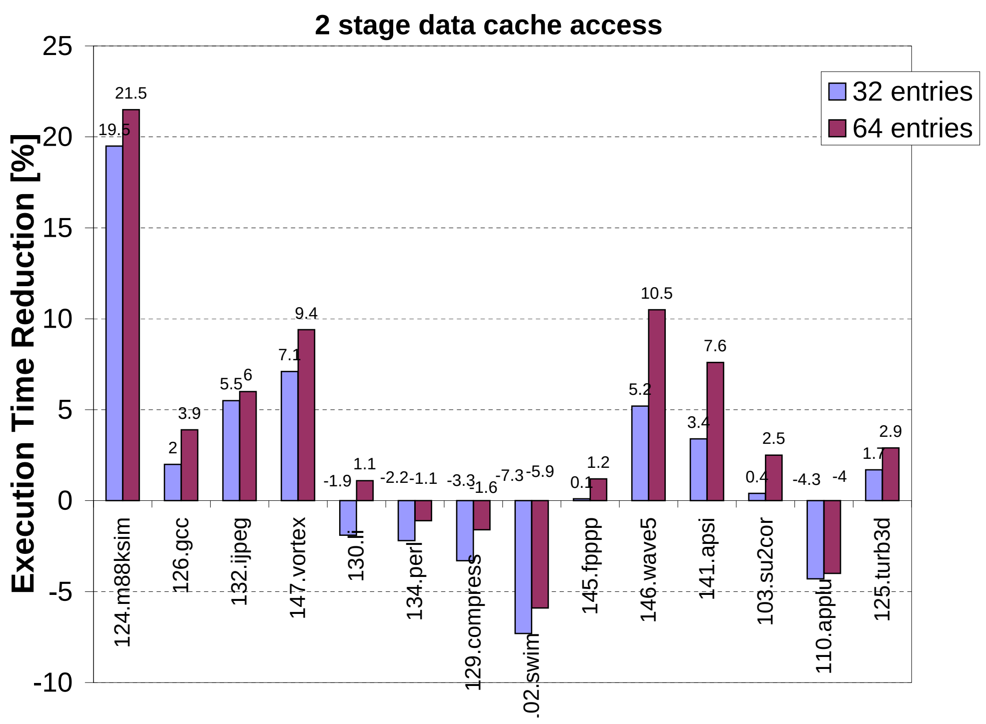
### 7.3.4 IPC Improvements

Next the IPC improvements that are observed due to load and store filtering is presented. Figure 7.13 shows these improvements assuming that data cache access takes

2 and 6 cycles respectively. As it can be seen the longer the cache latency, the greater the savings are due to load and store filtering. The average IPC improvements range between 3% and 15% for different values of data cache access cycles. For programs with high levels of reuse the IPC improvements are substantial - even as high as 55%. On the other hand for programs with very low levels of reuse, reductions in IPC were observed because the baseline processor has one less stage and therefore completes cache accesses one cycle earlier. The reductions in IPC are mostly less than 5%.

## 7.4 Summary

This chapter demonstrated that programs contain significant levels of load and store reuse opportunities. By filtering loads and stores significant energy savings can be achieved. While a number of techniques have been proposed to exploit reuse for program performance, these techniques are typically accompanied by an increase in energy consumption. In contrast the reuse unit design presented here simultaneously provides execution speedups and energy savings.



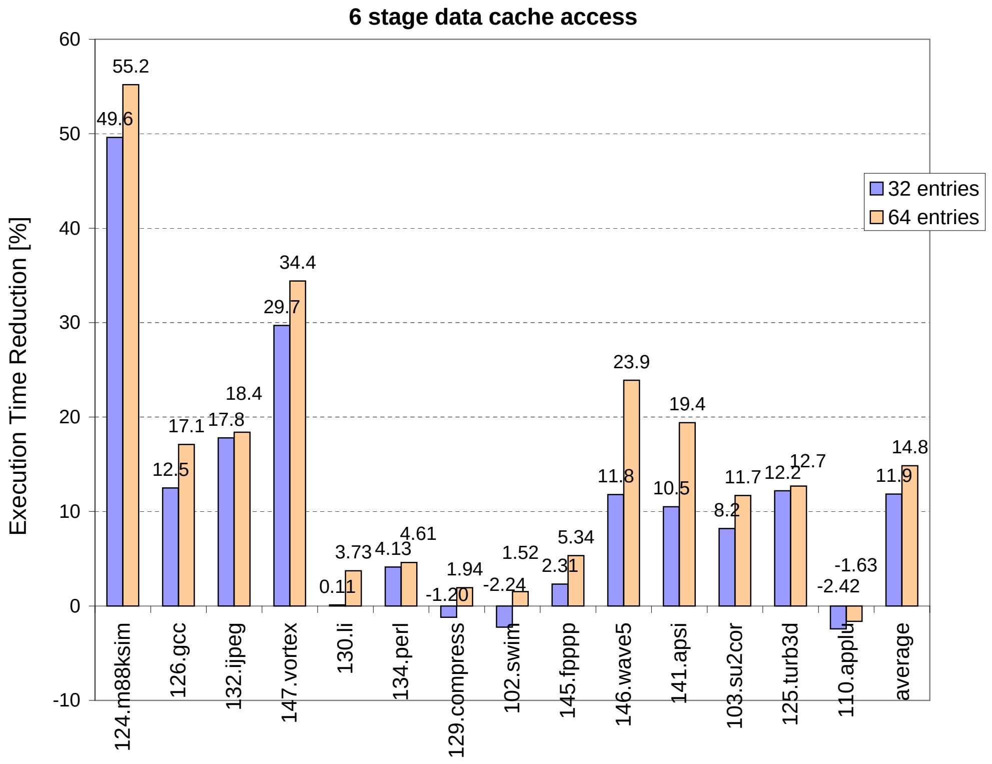


FIGURE 7.13. IPC Improvements for Varying Cache Latencies.

# Chapter 8
# Conclusion

As the demand in energy efficient processor designs increases, there is an expanding need for energy efficient memory systems since the memory hierarchy is one of the major sources of energy consumption. This dissertation has presented new energy efficient techniques for different components along the memory hierarchy. The proposed techniques include *frequent value data cache*, *frequent value data bus encoding* and *memory instruction reuse*. In addition to the energy efficient techniques, this dissertation also presents a thorough examination of values in a program's memory based on which of the techniques are deployed. In the following sections, the contributions are summarized and the directions for future research are proposed.

## 8.1 Dissertation Contributions

The main contribution of the work follows the line of *frequent value locality*. The characteristics found for frequent values led the way to the energy efficient cache design and the data bus encoding scheme. The new cache design explored the spatial locality of the frequent values and the new data bus encoding scheme explored its temporal locality. Techniques on how to identify the frequent values in the above scenario were also developed. Another contribution of this dissertation is the memory instruction reuse technique which explored the *value locality* for load and store instructions. Next, the frequent value locality and each contributive techniques are summarized.

- **Frequent Value Phenomenon** The values stored in the memory exhibit a high degree of redundancy in the sense that a small set of values keeps reoccurring both spatially and temporally. They are frequent values. Spatially, the frequent values occupy a large fraction of memory spaces almost anytime during program execution. They tend to distribute uniformly as well. Temporally, the frequent values exist from the beginning of the execution time till the end. To obtain the above conclusions, excessive experiments were conducted to fully explore the memory values across a set of very different benchmarks.

- **Identifying Frequent Values** Finding the frequent values in different applications can be achieved through either software or hardware. The software method runs the program using one input set and captures a set of frequent values for later use with other program inputs. This method is good for programs whose frequent values are input insensitive. The hardware method captures the frequent value set while the program is running. It can either run for only the beginning part or run throughout the program execution. The performance of both methods is satisfactory from the experimental study. The two hardware methods are used in the energy efficient cache and data bus designs.

- **Frequent Value Cache Design** The spatial locality of frequent values are used in new cache designs. The frequent values loaded into the cache are stored in a compressed form. Compressing the frequent values is not intended to keep more values in the cache, instead, it enables a restructured data array to operate at low power whenever a frequent value is accessed. Care is taken to ensure the restructuring of the cache does not incur longer time for every access. For nonfrequent value accesses, the cache takes more cycles to finish. Even though this introduces slowdowns, the load marking technique which marks dynamic loads of low frequent value accuracy as nonfrequent helps keep the speed loss at a minimum.

- **Frequent Value Data Bus Encoding** The temporal locality of frequent values is employed in designing low power data buses. The rationale is that the data bus commutes the value including frequent values between data cache and memory. The values flowing through the bus are encoded for low switching activity. To perform encoding and decoding, a simple frequent value based coder is crafted for good results and low overhead. Enhancements of the base encoding including XORing values, equality tests and excluding values are all developed for comparison and achieving maximum switching reduction. To keep the overhead of the encoder and decoder low, various methods are given such as using a fixed set of frequent values, and reducing coder accesses and size. The performance of the coders is also contrasted with one that runs an optimal replacement policy. Finally, the FV encoding scheme is compared against other data bus encoding techniques and shows significant improvements in bus switching reduction.

- **Memory Instructions Reuse** As an additional contribution to further reduce the cache energy, this dissertation has explored another type of opportunity. This opportunity enables a filtering mechanism that reduces cache accesses by satisfying the memory instructions through a smaller and simpler reuse unit rather than the cache. The reuse unit itself costs some energy but it is less than a cache access. The unreused instructions, however, waste the energy spent in attempting a reuse. The reuse mechanism is fine tuned so that the reuse unit does not consume unnecessary energy on reuse checking and information updating. When a substantial number of instructions can be reused an overall energy savings can be achieved.

## 8.2 A View to the Practicability of the Proposed Techniques

**Involving least amount of modification to the processor and the rest of the system.** Both the FVC and the FV encoding scheme are self-contained, in the sense that they do not require significant modifications to the existing architecture other than the cache and the bus. The FVC technique involves mainly the redesign of the conventional cache, which is almost transparent to the rest of the processor and memory hierarchy. The only change in the processor core is the frequent value finder whose location is flexible with respect to the pipeline stages. The FV bus encoding scheme logically needs the encoder and decoder at the two ends of data bus. Practically, the encoder and decode can reside in the memory controller in the CPU and on the bus. This modification is completely transparent to the rest of the system. As other data bus encoding schemes, the FV bus encoding also requires an additional control signal that goes between the memory controllers at the two ends of data bus.

**Area cost and complexity.** An important factor in considering the practicability of the proposed technique is the additional die area required and the complexity in implementation. For the FVC, the data array itself needs rewiring, the corresponding bitline, wordline drivers and sense amplifiers need to be repositioned. The additional hardware added are the FV decoder which is an array of registers, and the frequent value finder which is a CAM. To have a rough idea on how much area those hardwares could take, consider the floorplan of the Alpha 21264 processor given in Figure 8.1.

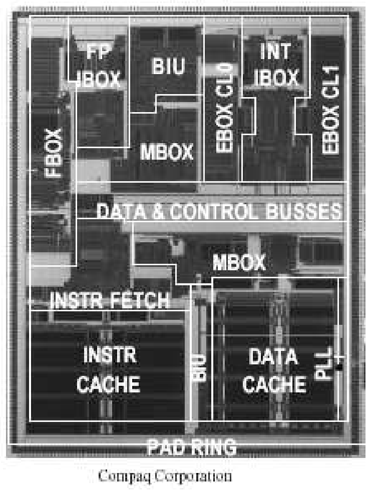


FIGURE 8.1. Alpha 21264 Die Photo and Floorplan.

Alpha 21264 contains a 64K byte on-chip L1 data cache. From the given figure, the area of this data cache is about 1/6 of the total chip area. Assuming the cache area is proportional to its capacity, a 32-entry register file with 4 byte in each entry is about only $1/2^9$ of the cache size (register file cell structure is similar to cache data array cell structure). Similarly, a 64-entry CAM array is about $1/2^8$ of the size of the data cache. Therefore, the additional hardwares added to the FVC design are nearly negligible in terms of area cost. Similarly, the encoder and the decoder in the FV encoding scheme also take very little space compared to the rest of the chip.

It is also very easy to design the extra hardware as the logic contained is very simple. Both register files and CAMs are standard architecture structures. Extending their functionality can be a good training process for college freshmen majoring in Electric Engineering.

**The significance of low-power memory techniques.** For both the high performance processors and the embedded processors, the memory system consumes significant amount of power if the processor core, the memory hierarchy and the disks are viewed as a "system". Within the processor, the main power-hungry units are still the L1 caches and the clock. Beyond the processor, a conventional disk contributes 34% to the average power and a low-power disk contributes 23% [26]. The memory subsystem, including the L1, L2 instruction cache, data cache and the off-chip memory, consumes 30–34% of the total system power. For single-issue processors such as embedded processors, the memory subsystem has a higher average power than the processor core [26]. Therefore, designing low-power memory subsystem has a significant impact on various future processors and systems.

## 8.3 Future Directions for Research

This research has focused on a uniprocessor configuration where there is only a single processor and the memory system is "private" to the CPU. In a multiprocessor environment, assuming a centralized shared-memory architecture, different processors may communicate through the shared memory data. In this scenario, the memory contents may have different characteristics than the uniprocessor memory. In the following, the possible extensions of the current frequent value study and the associated energy efficient designs in such a multiprocessor environment are discussed:

- **Re-examine Frequent Values Locality in the Multiprocessor Configuration.** While the frequent value property still holds for the private data of each single processor, the characteristics in the shared memory data may change. This is because the shared values can be read or written by more than one processor, indicating a stronger sense of being frequent. Consequently, the definition of a value being frequent should be associated with the number of processors sharing this data. This leads to a different methodology in exploring and identifying the frequent values.
- **Changing of the Data Cache.** The data caches attached to each processor in the multiprocessor architecture is usually larger than the uniprocessor cache. This is because larger caches can help reduce the memory bandwidth which is a critical resource in the multiprocessor architecture. Therefore it is desired that the caches consume less energy. For improved performance, the shared data are often duplicated in different individual processor caches or migrated between memory and caches for better accessing time as well as less contention. These movements of shared data may introduce new dynamic behavior of frequent values. It would therefore be interesting to see how the frequent values change by the cache coherence protocols, how they affect the cache energy consumption, if the old techniques still apply, whether different processor caches share some

frequent values, etc.

The address tags inside each cache which stores shared data may exhibit *frequent address property*. This is because the coherence protocol needs to locate a shared cache block via address tag comparisons for every processor. Therefore this property provides opportunities for designing energy efficient tag checks based on the frequent address property.

- **Exploiting the Bus Shared Among Multiprocessors.** It is natural to foresee that the data bus shared by all processors preserves frequent value temporal locality. This is due to the communication among processors sharing memory data values. The communication can be through the memory or directly between two processors. Either way the data need to go through the data bus. However, the FV encoding technique developed in this dissertation can not be directly applied since the bus has multiple senders and receivers and it is difficult to keep all the encoders and decoders consistent. Therefore it is challenging to derive an encoding scheme in a multiprocessor architecture whose data bus transmits abundant frequent data values.

  Besides the *data* value locality on the data bus, the address bus now may also exhibit the *address* value locality. As mentioned above, the coherence protocol sends the addresses for shared data on the address buses to locate it in a particular cache. This activity adds the frequent address locality on top of the address bus in the uniprocessor. Moreover, traditional address bus encoding schemes fail in a multiprocessor architecture because now the address streams may not be sequential. They are interleaved by the snooping addresses generated by the cache coherence protocol. As a result, the FV encoding scheme can be applied to the address buses since it does not require the sequentiality of the address streams. Yet, careful algorithms need to be developed to keep the encoder and the decoder simple and consistent.

# References

[1] Y. Aghaghiri, F. Fallah, and M. Pedram, "Irredundant Address Bus Encoding for Low Power," *ACM/IEEE International Symposium on Low Power Electronics and Design* pages 182–187, August 2001.

[2] D.H. Albonesi, "Selective Cache Ways: On Demand Cache Resource Allocation," *The 32nd Annual International Symposium on Microarchitecture*, pages 248–259, 1999.

[3] "ARM Processor Cores," *ARM Production Information Technical Report.*

[4] M. Azam, P. Franzon, and W. Liu, "Low Power Data Processing by Elimination of Redundant Computation," *ACM/IEEE International Symposium on Low Power Electronics and Design* pages 259–264, August 1997.

[5] R. I. Bahar and S. Manne, "Power and energy reduction via pipeline balancing," *The 28th Annual International Symposium on Computer Architecture*, pages 218–229, Gteborg, Sweden, 2001.

[6] L. Benini, "Automatic Synthesis of Sequential Circuits for Low Power Dissipation," Ph.D. Dissertation, Stanford, February 1997.

[7] L. Benini, A. Macii, E. Macii, M. Poncino, and R. Scarsi, "Synthesis of Low-Overhead Interfaces for Power-Efficient Communication Over Wide Buses," *ACM Design Automation Conference*, pages 128–133, 1999.

[8] L. Benini, G. DeMicheli, E. Macii, D. Sciuto, and C. Silvano, "Asymptotic Zero-Transition Activity Encoding for Address Busses in Low-Power Microprocessor-Based Systems," *Great Lakes Symposium on VLSI*, pages 77–82, March 1997.

[9] D. Books and M. Martonosi, "Value-Base Clock Gating and Operation Packing: Dynamic Strategies for Improving Processor Power and Performance," *ACM Transactions on Computer Systems*, pages 89–126, Vol. 18, No. 2, 2000.

[10] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *The 27th ACM International Symposium on Computer Architecture*, pages 83–94, May 2000.

[11] "SimpleScalar," *http://www.simplescalar.com/.*

[12] B. Calder, P. Feller, and A. Eustace, "Value Profiling," *The 30th Annual International Symposium on Microarchitecture* pages 259–269, 1997.

[13] R. Canal, A. Gonzalez, and J. E. Smith, "Very Low Power Pipelines Using Significance Compression," *The 33rd Annual International Symposium on Microarchitecture*, pages 181–190, December 2000.

[14] A. Chandrakasan and R. Brodersen, "Low Power Digital CMOS Design," Kluwer, 1995.

[15] N. Chang, K. Kim, and J. Cho, "Bus Encoding for Low-Power High-Performance Memory Systems," *ACM Design Automation Conference*, pages 800–805, 2000.

[16] "ChartWatch: Mobile Processors," *Microprocessor Report*, pages 43, vol. 14, archive 3, March 2000.

[17] W-C. Cheng and M. Pe, "Power-Optimal Encoding for DRAM Address Bus," *ACM/IEEE International Symposium on Low Power Electronics Design*, pages 250–252, 2000.

[18] D. Connors and W. W. Hwu, "Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results," *The 32nd Annual International Symposium on Microarchitecture*, pages 158–169, November 1999.

[19] A. Dhodapkar, C. Lim, G. Cai, and R. Daasch, "TEM$^2$P$^2$EST: A Thermal Enabled Multi-Model Power/Performance ESTimator," *PACS Workshop held in conjunction with ASPLOS*, 2000.

[20] J. Edler and M. Hill, "Dinero IV: Trace-Driven Uniprocessor Cache Simulator," *http://www.cs.wisc.edu/ markhill/DineroIV/*

[21] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy Caches: Simple Techniques for Reducing Leakage Power," *the 29th Annual International Symposium on Computer Architecture*, pages 131–140, May 2002.

[22] D. Folegnani and A. Gonzlez, "Energy-effective issue logic," *the 28th Annual International Symposium on Computer Architecture*, pages 230–239, Gteborg, Sweden, 2001.

[23] F. Gabbay and A. Mendelson, "Can Program Profiling Support Value Prediction," *The 30th Annual International Symposium on Microarchitecture*, pages 270–280, 1997.

[24] K. Ghose and M. B. Kamble, "Reducing Power in Superscalar Processor Caches using Subbanking, Multiple Line Buffers, and Bit Line Segmentation," *ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 70–75, 1999.

[25] R. Gonzalez and M. Horowitz, "Energy Dissipation in General Purpose Microprocessors," *IEEE Journal of Solid-State Circuits*, pages 1277–1284, September 1996.

[26] S., Gurumurthi, A. Sivasubramaniam, M. J. Irwin, N. Vijaykrishnan, M. Kandemir, T. Li, L. K. John, "Using Complete Machine Simulation for Software Power Estimation: The SoftWatt Approach," *the 8th International Symposium on High Performance Computer Architecture* , 2002.

[27] E.P. Harris, S.W. Depp, W.E. Pence, S. Kirkpatrick, M. Sri-Jayantha, and R.R. Troutman, "Technology Directions for Portable Computers," *Proceedings of the IEEE*, pages 636–658, Vol. 83, No. 4, April 1995.

[28] A. Hasegawa, I. Kawasaki, K. Yamada, S. Yoshioka, S. Kawasaki, and P. Biswas, "SH3: high code density, low power," *IEEE MICRO.*, pages 11–19, 1995.

[29] J. Hennessy, D. Patterson, "Computer Architecture A Quantitative Approach," *Morgan Kaufmann*, 2nd ed., 1996.

[30] J. Huang and D. Lilja, "Exploiting Basic Block Value Locality with Block Reuse," *IEEE International Symposium on High-Performance Computer Architecture*, pages 106–114, 1999.

[31] M. Huang, J. Renau, S. M. Yoo, and J. Torrellas, "L1 Data Cache Decomposition for Energy Efficiency," *ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 10–15, 2001.

[32] N. P. Jouppi and S. J. E. Wilton, "An Enhanced Access and Cycle Time Model for On-Chip Caches," *Technical Report, Western Research Lab*, July 1994

[33] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz, "A novel renaming scheme to exploit value temporal locality through physical register reuse and unification," *The 31st Annual International Symposium on Microarchitecture*, pages 216–225, December 1998.

[34] M. B. Kamble and K. Ghose, "Analytical Energy Dissipation Models for Low Power Caches," *ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 343–348, August 1997.

[35] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power," *the 28th Annual International Symposium on Computer Architecture*, pages 240–251, 2001.

[36] J. Kin, M. Gupta, and W.H. Mangione-Smith, "Filter Cache: An Energy Efficient Memory Structure," *30th Annual International Symposium on Microarchitecture*, pages 184–193, December 1997.

[37] J-S. Lee, W-K. Hong, and S-D. Kim, "Design and Evaluation of a Selective Compressed Memory System," *International Conference on Computer Design*, pages 184–191, October 1999.

[38] C. Lefurgy, P. Bird, I.-C. Chen, and T. Mudge, "Improving Code Density Using Compression Techniques," *The 30th International Symposium on Microarchitecture*, pages 194–203, 1997.

[39] K. Lepak and M. H. Lipasti, "On the Value Locality of Store Instructions," *The 27th Annual International Symposium on Computer Architecture*, pages 182–191, June 2000.

[40] M. H. Lipasti and J. P. Shen, "Value Locality and Load Value Prediction," *ACM 7th Architectural Support for Programming Languages and Operating Systems*, pages 138–147, 1996.

[41] T. L. Martin and D. P. Sewiorek, "A Power Metric for Mobile Systems," *ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 37–42, August 1996.

[42] T. Mudge, "Power: A First Class Design Constraint for Future Architectures," *High Performance Computer Conference*, Bangalore, India, December 2000.

[43] E. Musoll, T. Lang, and J. Cortadella, "Exploiting Locality of Memory References to Reduce The Address Bus Energy," *ACM/IEEE International Symposium on Low Power Electronics Design*, pages 202–207, August 1997.

[44] S. Onder and R. Gupta, "Automatic Generation of Microarchitecture Simulators," *International Conference on Computer Languages*, pages 80–89, Chicago, IL, May 1998.

[45] R. Panwar and D. Rennels, "Reducing the Frequency of Tag Compares for Low Power I-Cache Design," *IEEE/ACM International Symposium on Low Power Design*, pages 57–62 April 1995.

[46] D. Ponomarev, G. Kucuk, and K. Ghose, "Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources," *The 34th Annual International Symposium on Microarchitecture*, pages 90–101, 2001.

[47] D. Ponomarev, G. Kucuk, and K. Ghose, "AccuPower: An Accurate Power Estimation Tool for Superscalar Microprocessors," *5th Design, Automation and Test in Europe Conference*, pages 124–129, March 2002.

[48] M.D. Powell, A. Agrawal, T.N. Vijaykumar, B. Falsafi, and K. Roy, "Reducing Set-Associative Cache Energy via Selective Direct-Mapping and Way Prediction," *The 34th Annual International Symposium on Microarchitecture*, pages 54–65, 2001.

[49] M. D. 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," *ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 90–95, August 2000.

[50] S. Ramprasad, N. R. Shanbag, and I. N. Hajj, "A Coding Framework for Low Power Address and Data Busses," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 212–221, Vol. 7, 1999.

[51] P. Ranganathan, S. Adve, and N. Jouppi, "Reconfigurable Caches and their Application to Media Processing," *the 27th Annual International Symposium on Computer Architecture*, pages 214–224, June 2000.

[52] G. Reinman and N. Jouppi, "An Integrated Cache Timing and Power Model," *Technique Report, Western Research Lab*, Summer, 1999.

[53] J. A. Rivers, E. S. Tam, G. Tyson, E. Davidson, and M. Farrens, "Utilizing Reuse Information in Data Cache Management," *International Conference on Supercomputing*, pages 449–456, 1998.

[54] S. Segars, "Low Power Design Techniques for Microprocessors," *Tutorial of IEEE Solid-State Circuit Conference*, February 2001.

[55] J. Seward, "The CacheProf Home Page," *http://www.cacheprof.org/*.

[56] P. Shivakumar and N.P. Jouppi, "CACTI 3.0: An Integrated Cache Timing, Power, and Area Model," *Technical Report, Western Research Lab*, August 2001.

[57] A. Sodani and G. Sohi, "Dynamic Instruction Reuse," *The 24th International Symposium on Computer Architecture*, pages 194–205, June 1997.

[58] "SPEC CPU95 Press Release," *http://www.specbench.org/osg/cpu95/press.html*, August 21, 1995.

[59] M. R. Stan and W. P. Burleson, "Bus-invert coding for low-power I/O," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 49–58, Vol. 3, 1995.

[60] M. R. Stan and W. P. Burleson, "Coding a Terminated Bus for Low Power," *Great Lakes Symposium on VLSI*, pages 70–73, March 1995.

[61] C. L. Su, C. Y. Tsui, and A. M. Despain, "Saving Power in The Control Path of Embedded Processors," *IEEE Design and Test of Computers*, pages 24–30, Vol. 11, 1994.

[62] V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez, "Reducing Power in High-performance Microprocessors," *the 35th Design Automation Conference*, pages 732–737, San Francisco, CA, 1998.

[63] L. Villa, M. Zhang, and K. Asanovic, "Dynamic Zero Compression for Cache Energy Reduction," *The 33rd Annual International Symposium on Microarchitecture*, pages 214–220, December 2000.

[64] D. Vivek and S. Borkar, "Low Power and High Performance Design Challenges in Future Technologies," Invited Paper to GLSVLSI, 2001.

[65] K. Wilcox and S. Manne, "Alpha processors: A History of Power Issues and A Look to The Future," *Cool Chips Tutorial, held in conjunction with MICRO 32*, pages 16–37, November 1999.

[66] S. Wilton and N. Jouppi, "Cacti: An Enhanced Cache Access and Cycle Time Model," *IEEE Journal of Solid-State Circuits*, pages 677–688, Vol. 31, No. 5, May 1996

[67] E. Witchel, S. Larsen, C. Ananian and K. Asanovic, "Direct Addressed Caches for Reduced Power Consumption," *The 34th Annual International Symposium on Microarchitecture*, pages 124–133, December 2001.

[68] J. Yang, R. Gupta, "Load Redundancy Removal through Instruction Reuse," *International Conference on Parallel Processing*, pages 61–68, August 2000.

[69] J. Yang, Y. Zhang and R. Gupta, "Frequent Value Compression in Data Caches," *The 33rd Annual International Symposium on Microarchitecture*, pages 258–265, December 2000.

[70] J. Yang and R. Gupta, "FV Encoding for Low-Power Data I/O," *ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 84–87, August 2001.

[71] J. Yang and R. Gupta, "Energy-Efficient Load and Store Reuse," *ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 72–75, August 2001.

[72] J. Yang and R. Gupta, "Energy Efficient Frequent Value Cache Design," *ACM/IEEE 35th International Symposium on Microarchitecture*, November 2002.

[73] J. Yang and R. Gupta, "Frequent Value Locality and its Applications," *Special Issue on Memory Systems, ACM Transactions on Embedded Computing Systems*, Vol. 1, No. 1, 2002.

[74] S-H. Yang, M.D. Powell, B. Falsafi, K. Roy, and T.N. Vijaykumar, "An Integrated Circuit/Architecture Approach for Reducing Leakage in Deep-Submicron High-Performance I-Caches," *IEEE International Symposium on High-Performance Computer Architecture*, pages 147–157, January 2001.

[75] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "The Design and Use of SimplePower: A Cycle Accurate Energy Estimation Tool," *http://www.cse.psu.edu/m̃dl/SimplePower.html*, *ACM Design Automation Conference*, pages 340–345, 2000.

[76] Y. Zhang, J. Yang, and R. Gupta, "Frequent Value Locality and Value-Centric Data Cache Design," *ACM 9th International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 150–159, November 2000.