Abstract

Larin, Sergei Yurievich

Exploiting Program Redundancy to Improve Performance, Cost and Power Consumption in Embedded Systems

Under direction of Prof. Thomas Conte

During the last 15 years embedded systems have grown rapidly in complexity and performance to a point where they now rival the design challenges of desktop systems. Embedded systems are now targets for contradictory requirements: they are expected to occupy a small amount of physical space (e.g., low package count), be inexpensive, consume low power and be highly reliable. Regardless of the decades of intensive research and development, there are still areas that can promise significant benefits if further researched. One such area is the quality of the data which embedded system operates upon. This includes both code and data segments of an embedded system application. This work presents a unified, compiler-driven approach to solving the redundancy problem. It attempts to increase the quality of the data stream that embedded systems are operating upon while preserving the original functionality. The code size reduction is achieved by Huffman compressing or tailor encoding the ISA of the original program. The data segment size reduction is accomplished by modified Discrete Dynamic Huffman encoding. This work is the first such study that also details the design of instruction fetch mechanisms for the proposed compression schemes.

Exploiting Program Redundancy to Improve Performance, Cost and Power Consumption in Embedded Systems

by

Sergei Y. Larin

A dissertation submitted to the Graduate Faculty of North Carolina State University In partial fulfillment of the Requirements for the Degree of Doctor of Philosophy

Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, North Carolina

August 2000

Approved by:

Prof. Thomas Conte Chair of Advisory Committee	Prof. Eric Rotenberg
Prof. Edward W. Davis	Prof. Paul D. Franzon

Biography

I was born in a small town in the southern portion of Russia – Taganrog. In 1993, I enrolled in the Master of Science program at the Electrical and Computer Engineering department at the University of South Carolina under the supervision of Dr. Thomas M. Conte. After completion of my MS program, in 1995, I enrolled in the PhD program at North Carolina State University under the supervision of Dr. Thomas M. Conte and I have completed this program at August 2000.

Acknowledgments

Following the age-old tradition I would like to thank everybody who made this work possible.

The first and the most important are my parents Yuriy Alexeevich and Ludmila Vladimirovna Larin who made everything possible and supported me on each step of the way. Thank You.

As my parents granted me physical existence, my advisor Thomas Conte granted me the 'academic life'. Without his careful guiding and encouraging this thesis would never materialize. Thank You.

Next my thanks are going to my wife, Elena Vasiluevna Larina who helped and supported me in every possible way.

And final, but nonetheless important, I would like to thanks those who did not get in my way. Well, thank you.

Table of Contents

1 Introduction ... 1
1.1 Introduction and Motivation ... 1
1.2 Previous work ... 5
1.2.1 Previous Work in Code Compression for Embedded Systems ... 5
1.2.2 Previous Work in Bus Optimization ... 7
1.3 Target Architecture Description ... 9
2 Code Segment Redundancy Reduction ... 11
2.1 Motivation ... 11
2.2 Measuring the Available Redundancy ... 16
2.3 Code Compression Techniques ... 18
2.4 Tailored Encoding ... 22
2.5 Instruction Fetch Mechanism Issues ... 27
2.5.1 Instruction Fetch Organization and Modification of the Instruction Cache ... 27
2.5.2 Program Layout ... 29
2.5.3 Compiler Optimizations to Enhance Code Layout ... 31
2.6 Address Space Conversion ... 35
2.6.1 Branch Target Address Randomization ... 35
2.6.2 Base Line Instruction Cache Design ... 40
2.7 Compressed Instruction Cache Hardware Implementation ... 44
2.7.1 The Instruction Cache Design for Compressed Encoding ... 44
2.7.2 The Instruction Cache Design for the Tailored ISA ... 48
2.7.3 Decoding Complexity Evaluation ... 51
3 Data Segment Redundancy Reduction ... 57
3.1 Available Redundancy and Compression Strategy ... 57
3.2 Effects of the Data Cache on Data Compressibility ... 69

4 System Data Bus Redundancy Utilization 74
4.1 Motivation and Experimental Setup 74
4.2 Data Bus Coding Algorithms 80
5 Compressed Data Cache Hardware Implementation 85
5.1 Motivation 85
5.2 Compressed Data Cache Architecture 86
5.3 Dynamic Decoding Structure 96
5.4 Variations on the Compressed Data Cache Design 102
5.5 Final Configuration for the Compressed Data Cache 111
6 Conclusions and Future Work 114
References 118
7 Appendix 124

Table of Figures

Figure 2.1 Traditional ASIC Design 11
Figure 2.2 Proposed Approach to Embedded System Design. 14
Figure 2.3 Zero-NOP Encoding Example 17
Figure 2.4 Stream-based Huffman Encoding 20
Figure 2.5 Tailored Encoding Example 22
Figure 2.6 Comparison of Different Compression Techniques (code segment only). 23
Figure 2.7 Traditional Distribution of Miss Rate 27
Figure 2.8 Entropy Based Distribution 28
Figure 2.9 Atomic Fetch Block Structure 29
Figure 2.10 Treegion forming Example 31
Figure 2.11 Jump Optimization Example 32
Figure 2.12 Multi-way Branching Example 33
Figure 2.13 Branch Target Randomization 35
Figure 2.14 ATB Miss Ratio 36
Figure 2.15 Compression Including ATT Size 39
Figure 2.16 Banked Cache Architecture 41
Figure 2.17 Instruction Cache Structure for Compressed Encoding 45
Figure 2.18 Instruction Cache Structure for the Tailored Encoding 46
Figure 2.19 Cache Study Summary. Instruction Delivered per Cycle 48
Figure 2.20 Instruction Memory Bus Traffic Summary 49
Figure 2.21 Verilog Code for Decoder Example (Custom – left, Byte Based Huffman – right) 50
Figure 2.22 The Huffman Tree Decoder Structure 51
Figure 2.23 Estimated Huffman Decoder Complexity. 53
Figure 2.24 Estimated to Real Size Comparison for the Byte Based Compression Decoder (for the Compress Benchmark) 54

Figure 2.25 Estimated to Real Size Comparison for the Byte Based, Full Compression and Custom Coding Schemes (for the Compress Benchmark) 55
Figure 3.1 Dynamic Compression for M88ksim 62
Figure 3.2 Dynamic Compression for Go 62
Figure 3.3 Dynamic Compression for Vortex 63
Figure 3.4 Dynamic Compression for Gcc 63
Figure 3.5 Dynamic Compression for Perl 64
Figure 3.6 Dynamic Compression for Ijpeg 64
Figure 3.7 Dynamic Compression for Li 65
Figure 3.8 Dynamic Compression for Compress 65
Figure 3.9 Summary of Data Segment Compressibility 66
Figure 3.10 Entropy Change Due to Compression 67
Figure 3.11 Dynamic Compression for M88ksim in presence of a Data cache 68
Figure 3.12 Dynamic Compression for Li in presence of a Data cache 70
Figure 3.13 Dynamic Compression for Perl in presence of a Data cache 71
Figure 3.14 Effect of Data Cache on Data Stream compressibility 72
Figure 4.1 Traditional. Bus Encoding Experimental Setup 74
Figure 4.2 Bus Blocks and Tuples Structure 75
Figure 4.3 Busy Bus Cycles 76
Figure 4.4 Entropy Changes due to Caching 77
Figure 4.5 Oracle Block Distribution 78
Figure 4.6 Density of the Switching Activity on Compressed Data Bus 80
Figure 4.7 Transaction Intensity 81
Figure 4.8 Transaction Density 82
Figure 5.1 Compressed Data Cache Architecture 85
Figure 5.2 Block Placement Example - Expanded Block Placement 87
Figure 5.3 Block Placement Example - Reduced Block Placement 88
Figure 5.4 Read Pipeline. Multiple Set storage 89
Figure 5.5 Read Pipeline. Two cycle access 90
Figure 5.6 2x Restricted Compression Block Placement and Access 91
Figure 5.7 WUB Organization 92

Figure 5.8 Logical Structure of the Reprogrammable Huffman Decoder 93
Figure 5.9 Dual Bank RAM Implementation of the Huffman Decoder 94
Figure 5.10 Multiple Symbol Decoding Example 95
Figure 5.11 Biased Huffman Tree Example 96
Figure 5.12 Restricted Huffman Decoder Structure 97
Figure 5.13 Profile Point Selection 98
Figure 5.14 Compression Dependence on Profile Point Selection. Memory side vs. Processor Side 99
Figure 5.15 Miss Ratio Dependence on Profile Point Selection. Memory side vs. Processor Side 102
Figure 5.16 Compression Dependence on Profile Point Selection. Memory side vs. Storage Contents 103
Figure 5.17 Miss Ratio Dependence on Profile Point Selection. Memory side vs. Storage Contents 104
Figure 5.18 Two Level Compressed Data Cache 105
Figure 5.19 Miss Ratio for Two-Level Cache compared with the Original Implementation.... 106
Figure 5.20 Reference Hit Breakdown for the Original Compressed Data Cache 107
Figure 5.21 Reference Hit Breakdown for the Two-Level Data Cache 108
Figure 5.22 Two Level Cache Size Variation (Logarithmic Scale) 109
Figure 5.23 Miss Ratio Comparison between Compressed and Uncompressed Caches 110
Figure 5.24 Absolute Dynamic Compression for Storage Profile Scheme 111
Figure 5.25 Entropy Miss Ratio Summary 112

1 Introduction

1.1 Introduction and Motivation

The importance of embedded systems today is easy to underestimate. During the last 15 years embedded systems have grown rapidly in complexity and performance to a point where they now rival the design challenges of desktop systems. This evolutionary trend is known as the fifth era of computing: from main frames to minicomputers to microcomputers followed by PC or desktop era and finally into the embedded system age. In the last year the number of embedded processors sold actually exceeded the amount of general-purpose units sold. And the trend is growing. With the popularity comes the challenge. Embedded systems are now targets for contradictory requirements: they are expected to occupy a small amount of physical space (e.g., low package count), be inexpensive, consume low power and be highly reliable. However, they are asked to take on more complex functions [3],[17],[47]. The digital image processing, DVD and third generation cell phone base station require server-like performance from embedded processors and most of the time they stand up to the challenge.

Regardless of the decades of intensive research and development, there are still areas that can promise significant benefits if further researched. One such area is the quality of the data which embedded system operates upon. This includes both code and data segments of an application. Designers spend a long time on logical optimization and minimization of applications, but they can only achieve as much as the original instruction set architecture (ISA) will permit. Moreover, excessively cautious and there forth often redundant approach have to be

taken for data accommodation and processing. As a result, most of the time the code and data used on embedded systems contains a significant amount of *redundancy*.

The presence of this redundancy would degrade the potential performance of any processor. The Flynn's bottleneck [32] is one of the toughest frontiers in high performance system design. It receives increasing amount of attention as the core vs. memory speed gap multiplies. An elaborate design approaches and compiler techniques are taken to reduce this gap. Nevertheless few designers actually pay attention to the fact that information contents of data transferred through those elaborate units is relatively low.

As the complexity and size of applications increase, the traditional method of hand coding applications for embedded systems is quickly becoming obsolete. Although hand coding is still practiced for critical regions of a program, and is very popular for simple DSP applications, optimizing compilers starting to play a major role in the overall process[3],[28]. But, those compilers are normally built to compile a relatively wide spectrum of applications. So, compilers have to tolerate some redundancy in the code they produce in order to support the numerous applications encountered. For some real world applications (like the SpecInt95 benchmarks [45] compiled with the optimizing GCC compiler –O2 option for HP PA architecture) entropy ranges between 0.75 and 0.80, which means that the spilt between two possible symbols is 70 to 30%. On the other hand compilers have extensive information about the application being compiled which could be effectively utilized.

This work presents a unified, compiler-drive approach to solving the redundancy problem. It attempts to increase quality of the data stream that embedded systems are operating upon while preserving the original functionality. Moreover, some of the proposed methods offer increase in performance combined with smaller code and data sizes, which proves the negative

effect of high redundancy data and ultimate benefits that could be achieved by removing it.

There are three major components of embedded system front end considered in this thesis.

- Instruction fetch mechanism and instruction cache design;
- System bus;
- Data fetch mechanism and data cache design;

The first part considers reduction in static size of code segment and alternative design for instruction cache. It is primarily targeted at reduction of the instruction memory size requirements for System On Chip (SOC) architectures. It is achieved by compression or customization of the original instruction set to meet the needs of each particular application. Several prior approaches to this problem have either defined new instruction-set architectures (e.g., the ARM Thumb Instruction Set [26] or SGI MIPS16 [27]) or defined elaborate compression schemes without taking the impact on instruction fetch into account (e.g., IBM CodePack [9], Cooper and McIntosh [12]). The main contribution of this study is a unified, compiler-driven approach to the problem. It presents both code compression strategies and their corresponding instruction fetch mechanisms along with compiler techniques to facilitate them. The instruction cache is also reconsidered, and it is allowed to hold the high entropy representation of the original code segment. This fact significantly increases instruction cache effective capacity and ultimately results in a higher performance.

The second part of this work pays attention to communication issues within the memory hierarchy and front end of embedded processors. Following many previous researches in the area [33],[34],[49] this thesis research utilization, throughoutput and power consumption on memory and system buses in SOC. The improvement is achieved through dual coding of the original stream: a variation of Gray coding is applied on top of dynamic Huffman compression.

For some configurations the ultimate goal of shorter transaction time with lower switching activity is attained.

The last, but not least, considered component is the data cache. Even with a perfect instruction cache, a processor can only operate as fast as data could be delivered to it. Unfortunately dynamically accessed data contains a significant amount of redundancy, which reduces the effectiveness of the data fetch mechanism. The quality of this data could be significantly improved by dynamic compression, if a practical scheme can be invented to exploit this data quality. It is no surprise that dynamically accessed data is compressible. Research into value prediction hints at high rates of redundant data accesses [51]. However, exploiting this property to yield a performance benefit has been difficult so far. With the help from compiler and the run time profiling we can double performance (miss ratio) of the data fetch path. But it can only be truly effectively explored if the data cache is permitted to hold compressed blocks. This fact in turn introduces serious design challenges.

To summarize the proposed solutions to the above-mentioned problems it is necessary to state the following. In any embedded system, especially in those based upon the VLIW architecture, static (or compile time) part of design cycle gains greater importance very rapidly. This offers unique opportunity to improve performance of the available hardware base with minimal dedication of some logic complexity and maximum utilization of the compile technology. The target of optimization is the available redundancy in application. The beneficiary is the front end of an embedded system. The gain is lower physical space, higher performance and lower power dissipation.

1.2 Previous work

1.2.1 Previous Work in Code Compression for Embedded Systems

This study would not be complete without first mentioning previous research in this area. As was briefly mentioned in the introduction, in the past there were several studies regarding the reduction of the ROM size in an embedded system [1],[9],[10],[12],[17],[18],[47].

In several works by Wolfe, et al. [1],[17],[18], a technique to execute compressed programs on an embedded RISC architecture (MIPS was used as an example) was studied. The initial motivation involved reducing the code size difference between the RISC and CISC embedded processors. Besides the diversity in target architecture, one of the major differences from the current study was in the unified approach to the compression. Wolfe's studies used the Huffman algorithm for compression, but only one common histogram was built for a set of all experimental benchmarks and only *byte*-based alphabet was considered. In addition to that a fixed-size 32-byte blocks were considered an atomic unit of compression. The goal was to create a single encoding for a fixed architecture and satisfy some range of applications. This single encoding scheme is important when building a general-purpose system, but seems less effective for embedded applications with their unique code base. In the Wolfe's work [1] code is uncompressed at the instruction cache miss path, and the study does not discuss further details of instruction cache design. A special hardware structure (called Cache Line Address Lookaside Buffer) was provided to guarantee dynamic conversion between compressed and original address space. Regardless all the differences, the original work by Wolfe [1] was one of the major inspirations for the code compression part of this thesis.

Several industrial solutions to the code segment size problem include the IBM CodePack [10],[42], the ARM Thumb [26], and the SGI MIPS16 [27]. The first one uses the dictionary Huffman compression scheme, while the latter two of these provide special compact subsets of the original instruction set architecture (ISA). Truncating the original ISA reduces its flexibility, which ultimately results in increased instruction count and, in general, slower running applications. The CodePack also has the disadvantage of keeping the instruction cache uncompressed. The consequences of this decision are discussed in this study in great details. Recently some design issues regarding instruction fetch mechanism for IBM's PowerPC 405 (which uses Code Pack) were revisited by Lefurgy in [48]. The important achievement was the fact that after certain improvement to existing hardware structures (instruction fetch pipeline architecture) the compressed code exhibit better performance then the native program.

A work by Yoshida et al. [47] specifically concentrates on low power embedded system processor design. They are using a compression method to reduce power consumption of an embedded chip. It is specifically targeted at I/O interface by means of substituting original 32-bit instructions to a set of references in transform table (which resided on chip). This way by execising tradeoff between silicon area and encoding complexity they significantly decrease I/O load and executable size. For ARM610 used in experiments, for some benchmarks 12-bit pseudo instructions we reported to be sufficient to substitute original 32-bit operations.

Cooper and McIntosh [12] spend most of their effort reorganizing code at the assembly level via suffix-tree code compression. They reported a very moderate level of compression (5 to 15% reduction). In either way this study is orthogonal to the approach taken in this work.

A series of works by Fraser, et al. [9],[13] considers elaborate compression algorithms at the assembly level with the same lack of attention to the instruction fetch issues. The

experimental results in this thesis show that neglecting instruction fetch performance in the presence of compression may lead to incorrect conclusions about the appropriate scheme to implement.

A recent series of publications by Lefurgy [25],[43],[44] describes a way to manage the compressed code segment with software. In the [44], authors proposed software managed decompression tightly coupled with the instruction cache. The compression granularity in [44] is a single cache line, as opposed to a single basic block used in this thesis.

An interesting study by Liao, et al. [15] employs an effective compression algorithm (External Pointer Model by Storer and Szymanski [24]) at the assembly level. In essence this method introduces a micro procedure calls to a common regions of code. Liao reported an average of 30% code size reduction. Two implementations, software-only and 'call-dictionary' are considered. Both increase the number of branches in the code and (reportedly insignificantly) the operation count. Also due to high granularity, some opportunities for compression are missed, and as with the CodePack and many others instruction fetch schemes use decompression at miss time and uncompressed cache.

1.2.2 Previous Work in Bus Optimization

The next set of relevant research deals with bus power consumption. A number of researchers addressed power consumption on *address* buses. It is understandable since address stream exhibits a great deal of repetitiveness and very low entropy. The fundamental work by Hammerstrom and Davidson [40] in 1977 showed that there is less than or equal to one percent of information content in a typical address trace. This is rather understandable since majority of

the time address (Next PC) is merely an increment of the previous value. Combined with normally high spatial locality of references, this accounts for very low information contents.

There were several researches in the area, and the most relevant are as follows. The work by Su et al. [38] used the Gray Coding for address generation in RISC-like VLSI-BAM. The idea was to make program counter produce Gray codes as oppose to normal increment. Certain care had to be taken of branch target addresses and reported savings were up to 58% switching activity reduction. The next work by Musol et al [41] concentrated on improving the Address bus Gray coding by proposing a *Working-Zone Encoding*. This approach is based on observation that different address space areas exhibit different behavior. For some cases authors reported up to 65% switching activity reduction.

In the domain of data buses coding the following work deserves attention. The work by Stan and Burleson [33] concentrated on special encoding for terminated off-chip broad –level busses and tri-state on-chip buses for low-power communication. Authors considered not one but several different techniques for minimizing transaction activity. Those methods included not only logical minimization (via compression and special coding) but also by phase modulation of bus signals. Several techniques combining various encoding models were proposed. For their experimental setup savings of up to 68% were achieved. The same authors in [34] also proposed a Bus-Invert method which tradeoff performance and area for low power dissipation.

Finally in the area of data cache compression no previous research was found.

1.3 Target Architecture Description

All experiments in this work are based on the TEPIC (TINKER EPIC) VLIW embedded architecture [11]. It is a 40-bit version of the HP PlayDoh VLIW machine specification [21], adapted for embedded system use. It is important to note that the TEPIC ISA encoding is very similar to the Intel/HP IA-64 ISA, since the PlayDoh was one of the major influences for the IA-64. For a core processor configuration, we assume a six-wide issue machine, with four execution units that can execute any instruction except for memory access, and two universal units (that can perform any instruction including memory accesses). The register files are fixed to 32 general-purpose, 32 floating point and 32 predicate registers. The detailed encoding formats for operations are shown in the Appendix Table 1.

A special compact encoding is used to encode a single VLIW Operation (MultiOP or MOP for short – it combines all the instructions that must be issued in the same cycle). This compact encoding is known as *Zero-NOP* encoding and was originally designed to decrease the size of the VLIW code segment as compared to an equivalent RISC code [7]. Each operation in the original ISA is equipped with a dedicated bit, which is set only for the last operation in a MOP (See example in Figure 2.3). This encoding allows exclusion of the empty instructions (NOPs) in the final scheduling, which are normally the major contributors to the traditional VLIW code size explosion.

Generally speaking, TEPIC is a very powerful architecture, and a rather aggressive approach to an embedded system implementation. But with the current rate of progress, (Moores's law) it seems to be a very likely scenario for near future designs [52],[53]. However

the target architecture does not obscure the importance of the code and data compression in the current work, which might prove even more relevant for smaller systems.

As was mentioned before, the LEGO optimizing compiler [7] is used to schedule and optimize the code. The LEGO compiler employs standard optimizations and global instruction scheduling using Treegion [4],[5],[6] block forming. A modified version of the original TINKER assembler is used to generate custom encoding, compressed object file as well as the synthesizeable Verilog for the decoder. The compiler is able to output an intermediate code (Rebel) that is executed via the TINKER YULA emulation tool. Annotations are added by the compiler to emit an instruction address or load/store trace for simulations (these annotations are not included when determining the instruction addresses or performing compression). All the studied hardware structures are modeled via simulators using an execution-driven trace.
The SPECint95 [45] benchmark suite is used for all experiments.

2 Code Segment Redundancy Reduction

2.1 Motivation

The instruction fetch process is a well-known bottleneck for any architecture, and is a crucial process for embedded systems. Initial analysis shows that most of the time the information contents in typical applications is as low as 20 -30% [50]. This means that the expensive instruction fetch mechanism 80-70% of the time performing redundant operations. The code segment's size can be reduced as much as 50% [50], by modifying the original ISA design and adjusting the design of the instruction cache. As a result the new program can be

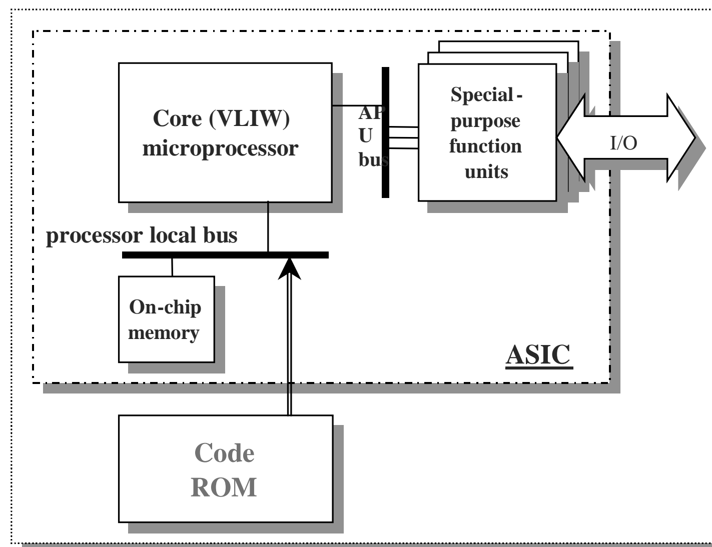


Figure 2.1 Traditional ASIC Design

executed at the same, or even faster rate, when compared to the original (native) code. By

tuning the instruction fetch pipeline, not only the performance but also the embedded system's overall size, cost and, indirectly, the power consumption are improved, *without reducing original functionality*. The key feature is that the original ISA is uniquely adjusted for the specific conditions of *each* particular application. The ISA could be either custom tailored or compressed. Once the code is compressed, it is kept this way throughout the whole instruction fetch pipeline, from the code ROM to the decoding stage of the processor. An important and unique contribution to the previous research in this area is the fact that the instruction cache itself is kept compressed, which significantly increases its effective capacity.

One commonly employed approach in building an embedded system is by using an Application-Specific Integrated Circuit (ASIC) [3],[27],[28],[42] design. It is commonly known as a System On Chip (SOC). Such a system is normally composed of several building blocks taken from a component library. All application code is stored in a ROM and an off-shelf on-chip core processor is used for execution [3],[52],[53] (see Figure 2.1). This method is a flexible and powerful way to quick implementation of an embedded system with minimal resources. At the same time, due to the inherited flexibility, a number of high level architectural enhancement are possible, which makes it easy to put into operation techniques and solutions described in this thesis.

First and probably the most important problem with modular ASIC approach and off-shelf core processor is that the instruction memory ROM size multiplies with the growth of the device's functionality. The ROM size will soon become the major cost defining factor and bottleneck for the instruction fetch (IFetch) mechanism [1],[3],[14],[27],[28],[50] as well as overall system implementability. As a result, the code ROM often has to be implemented on a separate chip, which involves the familiar difficulties associated with remote instruction

fetching and off-die power consumption. One of the many challenges of an embedded system design is to reduce the size of the ROM without sacrificing *the functionality* and *performance* of the system. As have been mentioned before, the approach taken in this work for the code segment is trying to reduce redundancy of the code stored in the ROM by customizing the original ISA or by utilizing compression to modify the existing code. This increases the utilization of the entire instruction fetch pipeline and in turn reduces the ROM size. This effect can only be achieved if a substantial amount of redundancy indeed exists in the code.

With the growth of embedded systems application complexity far beyond familiar embedded DSP applications, as have been mentioned before, the traditional methods of hand coding and optimizing are becoming less and less effective and increasingly time consuming. In addition market's requirements of a short design cycle and increased reliability have become a limiting factor. A practical way to use a high level programming language, while maintaining optimal target code quality is needed [3]. This calls for developing a sophisticated compile technology and majority of vendors invested tremendous amount of time and effort in doing that. For example, the TI 320Cxxx series of DSP processors has been successful in large part due to the vendor-supplied optimizing compiler [28]. This work follows this trend by focusing on compiler-driven code design and data optimization for embedded systems applications. The compiler used for this study is the optimizing LEGO compiler developed in the Tinker group at North Carolina State University (NCSU) [4],[5],[6],[11]. This is a highly optimizing speculating compiler targeted to a wide range of very long instruction word (VLIW) architectures including the Intel IA-64 prototype. It uses Treegion-based scheduling [4],[5],[6] and provides a comprehensive set of traditional optimizations. While being a research compiler it is available for easy modification to fit the current study.

In general an important feature of embedded systems is their specific code base. Since an embedded system is likely to execute the same code base throughout its life span, the compiler can customize the original instruction set architecture. After scheduling and optimization, once the final image is available, the compiler generates an efficient *custom-tailored* instruction set architecture and decoder structure, which can interpret it. In other words

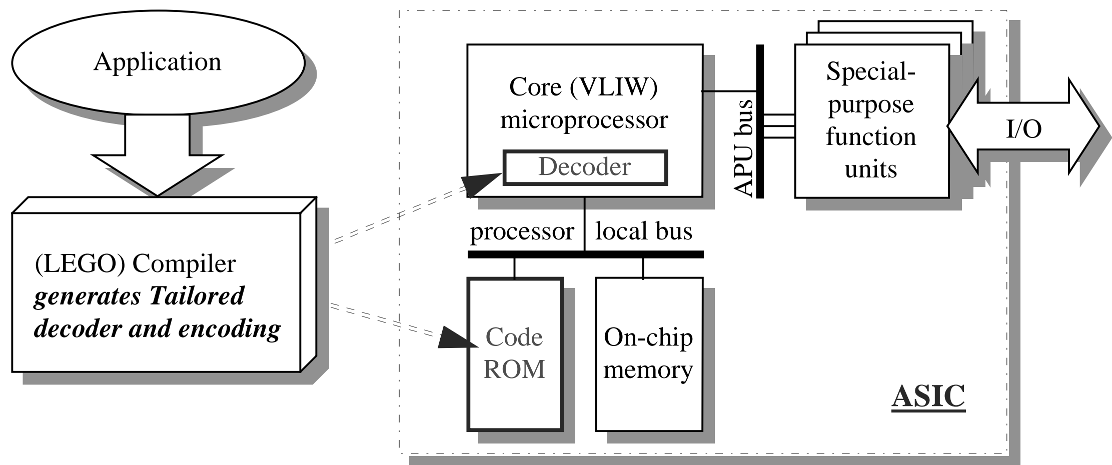


Figure 2.2 Proposed Approach to Embedded System Design.

this study is not bounded by the rigid traditional instruction set architecture. As have been mentioned before, traditional ISA's are normally designed to fit a wide spectrum of applications. In this study, the instruction set architecture is a parameter and is optimized for space and cache performance. In general the ISA could also be improved for power consumption, branch/data prediction accuracy, and various other design goals. Nevertheless only code size is considered at this point. Another closely coupled possibility is compression of the original ISA. The result is a compressed set of original instructions that need to be uncompressed prior to execution. The code could remain compressed all the way down to the instruction cache, with obvious advantages in the amount of required space. (See Section 2.3 for a more detailed discussion).

Another element of this work involves the justification of use of VLIW architectures for embedded systems in general. Obviously the VLIW architecture seems to be a natural fit for the embedded system environment since many traditional VLIW problems do not affect embedded systems. There is almost no code compatibility problem between generations. The use of relatively simpler hardware leads to a higher performance and low power consumption, when compared to an equivalent issue-width superscalar architecture. And one of the foremost advantages for this work is the primary role of compiler in producing the code. Since normally extensive information is available about the dynamic behavior of the program, high quality schedules can be produced by the optimizing compiler. All these statements were proven to be accurate at the design of the commercial TI DSP processors [28]. The familiar challenges present are the static code size and effective high bandwidth instruction fetch. The object code size difference between the VLIW and the superscalar architectures is first reduced by using the Zero-NOP encoding [7] and restricted code duplication in the LEGO compiler to the lower RISC-like levels. Then the code is further optimized for required space by reduction on an individual operation size through tailor customization or compression. The instruction fetch related issues for the Zero-NOP encoding have been discussed in [7],[8] and are further extended in this work. The decoder's structure for the optimized encoding is produced by the compiler, which has all of the information needed – logical functionality, in- and output connectors along with the required control signals. The overall structure of such a system is presented in Figure 2.2. Thus the compiler now plays a major role in dictating not only the ROM's contents (the executable code), but also the core processor decoder's logic and other components design.

2.2 Measuring the Available Redundancy

In order to estimate the amount of useful information in a data set we should choose a way to measure the present redundancy. The available redundancy could be measured by calculating the *entropy* of the code as it was defined by Shannon in [29]. In the Equation 1, $P_x(x)$ is the probability of alphabet character *x* in the set. The base of the logarithm only matters for convenience of presentation and is normally set to ten.

$$H(x) = \sum_{x, Px(x) \neq 0} - Px(x) \log_b Px(x) \text{ (Equation 1)}$$

For a binary system (see Equation 2), P_0 is the probability of zero and P_1 is the probability of one in the set (or embedded system application executable code in our case). It is also more convenient to use logarithm base two to represent results.

$$H = -(P_0 \log_2 P_0 + P_1 \log_2 P_1) \text{ (Equation 2)}$$

So for the logarithm base two, when the entropy is equal to zero this means that all probabilities but one are equal to zero (just one symbol in the set) and information contents is none. On the other hand, when the entropy is equal to one, all symbols in the set are equally probable and the amount of information contents is at the maximum. Once it was proven that there is sufficient slack in the data set, we should define a strategy on its removal, and first the redundancy of code segment is discussed.

Traditionally, there are two general approaches for the reduction of a program

redundancy (or increasing its entropy in different terms). One is the reduction of the number of redundant assembly operations in the code [9],[12],[13],[15],[27], and the other is the reduction of the operation size [1],[10],[28]. Generally, these two approaches are both trying to achieve the same goal – to increase the entropy of the code, but it is achieved in different ways. This means that applying *both* of them to the same portion of code obviously will not result in better compression. (Nevertheless different areas of code could be optimized with different methods.) It is important to notice that this statement is assuming that all traditional optimizations (like constant propagation, redundant computation and dead code removal) have been done and code does not contain any logically redundant operations or computations. This work chiefly concentrates on the latter approach - reducing the operation size.

There are two methodologies for reducing operation size: *tailoring the ISA* or

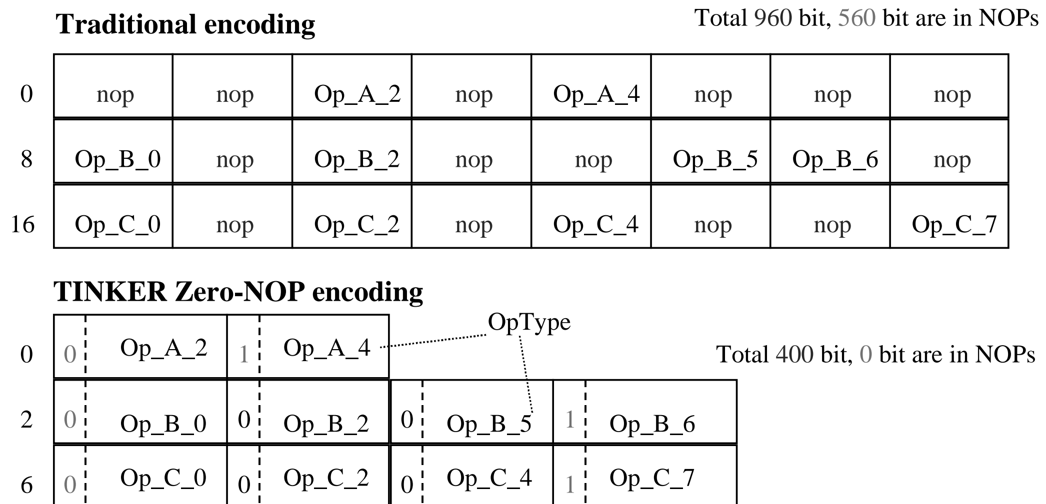


Figure 2.3 Zero-NOP Encoding Example

compressing the code. The tailored ISA is a new and unique encoding, which is generated for one particular program/application and best fits its characteristics. After decoding a tailored operation, the internal processor signals are obtained. Because of the tailoring process, different

operations are getting different sizes, which affects the branching mechanism (this topic will be discussed in greater details in Section 2.4).

On the other hand, compression of the code segment takes an existing encoding (ISA), and, according to the static frequencies of elements in the source code, determines the best way to pack it. Once a compressed operation is uncompressed at execution time, it still needs to be decoded to obtain internal processor signals. Theoretically tailoring the ISA should yield better performance (e.g., no intermediate decompression needed), while compression should yield a smaller code size. This work finds that this intuition holds, even when the improved instruction cache performance from caching compressed code is taken into account (see Section 2.7).

2.3 Code Compression Techniques

Let us first discuss compression of the original instruction set. In general, the compression circumstances for embedded systems code segment are very favorable, since the entire code image is available statically for the compression algorithm and compression objectives are to reduce the static size of the code. A fast hardware decoder could be used for decompression and interpretation. The main issue is the complexity, and therefore size and speed, of the decoder. There are virtually thousands of potential compression algorithms to choose from. First, we can safely eliminate dynamic compression algorithms since we need to store compressed code and also fully utilize its static availability. Here it is important to reemphasize that since we are concerned only with the static size of code (ROM size), we only need to know the static distribution of elements. This might not be the case for compressing

dynamic data sets, as we will soon see. Also if code segment would be optimizing for dynamic interpretability, it might involve dynamic profiling and analysis. Nevertheless for the static environment, the Huffman lossless compression algorithm [2],[30] produces near optimal results for an integer number of code bits. It is an entropy-bounded code and will use $N_i = -\log_2 P_i$ bits to encode character *i* in the range $1 \leq i \leq N_{TotalNumberOfCharacters}$. It also allows reasonably fast decompression (FSM or table lookup) at a realistic real estate price [17],[18]. For similar reasons, the Huffman compression algorithm was used in several previous studies [1],[9]. The closest rival to the Huffman compression scheme is the Arithmetic Coding compression [55],[30],[46] algorithm (or simply arithmetic coding), which could use a fractional number of code bits to encode the original symbol. Nevertheless, the arithmetic coding compression algorithm is much harder to decode in hardware and also might need some additional overhead for stop symbols on a low granularity compression [46].

In addition to the described methods, a new and unique compression algorithm could be created if a precise cost function would be defined, and this is reserved as a future work. Meanwhile for the rest of this work we will be using variations of the Huffman compression algorithm.

There were three major variations considered. They all differ in ways of composing the input alphabet for the Huffman compression algorithm. The first variation is the traditional *byte*-based method [1],[2]. The code segment is treated as a stream of bytes and compressed accordingly. As we will see shortly, this method produces the smallest decoding table and the simplest decoder.

Second variation is the *stream*-based approach. The idea behind it is that certain fields in an instruction encoding exhibit more repetitive patterns when taken as independent

compression streams, than when combined with other fields (see Figure 2.4 and Table 2 in Appendix). For example OpType and OpCode fields of the TEPIC operation encoding are set to 'INT_OpType' and 'ADD_OpCode' very often (up to 30% in some applications). The same is true for the predicate field, which is most of the time (except for the if-converted code) is set to predicate register number one which in the TEPIC architecture is hard-wired to 'true' and means unconditional execution of the operation. According to that observation, alphabet streams are fixed at certain operation field boundaries, as shown in Figure 2.4. The initial boundary selection was made manually after careful examination of the generated assembly code. Nevertheless, there were several experiments conducted in order to prove this selection and they will be discussed shortly. However, stream based encoding is not entirely new. A similar observation was made in the IBM CodePac [10],[42] instruction set architecture and used there as a base assumption for further Huffman table-compression scheme.

The last approach to alphabet selection for the Huffman input alphabet composition is to

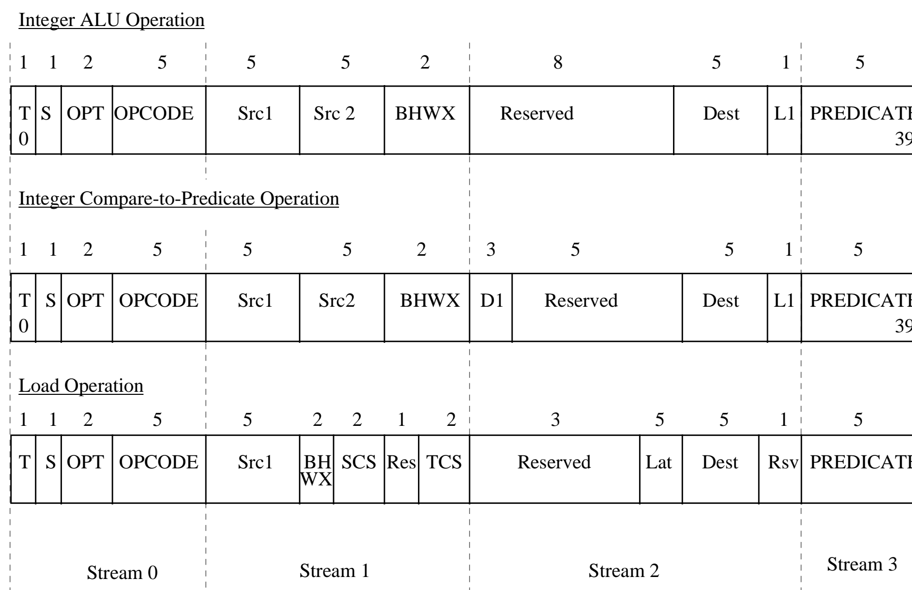


Figure 2.4 Stream-based Huffman Encoding

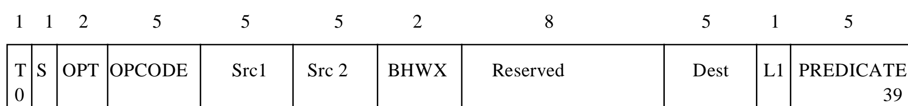
consider the *whole operation* as a compression unit. Theoretically this alphabet might have up to 2^{40} entries (since operation size is 40 bit long), but in practice this number is significantly smaller. This method produces the largest decoding table, but surprisingly enough, has the greatest potential for compression. This result becomes more understandable when one examines the generated Huffman codes. Even with a large number of dictionary entries, the size of the popular ADD instruction often went down from 40 to six bits. Furthermore, none of the Huffman codes for the full compression model has exceed the original operation size. In contrast, the maximum degree of compression of either *stream* approach is the sum of the maximum compression of all (four in our case) streams, which easily exceeds six bits in most cases. From another perspective the whole operation compression method (Full Huffman from now on) should take us as close to the entropy of the code as possible. If all possible bit combinations for all the possible instructions in the code would be determined (extract all unique instructions), and then encoded with the least number of bits according to their absolute frequency, we will get virtually perfect (or bottom line) compression while still having separate instructions in the code.

One additional detail of Huffman compression involves the maximum symbol length. For some inputs, the Huffman algorithm produces very long output codes that might exceed the original operation size and become incompatible with the instruction fetch hardware (like instruction bus weight for example). The compiler keeps track of such events, and either alternates the compression process (similar to the Bounded Huffman code described by Wolfe [1], where some additional encoding is required to guarantee the size) or substitutes the rare instruction with an equivalent group of more common ones. It is important to notice that virtually no such events were detected for the current experimental setup.

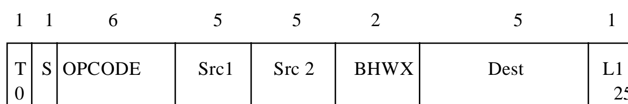
2.4 Tailored Encoding

The main idea behind the Tailored encoding is to provide to an operation as much space as it really needs in the contents of the current application, but not to compress it otherwise. At the same time we need to guarantee that within the same operation type/code instruction field boundaries remain at fixed locations. This requirement does not impose critical limitations on size reduction but significantly simplifies the decoder. As we will soon see, the tailored

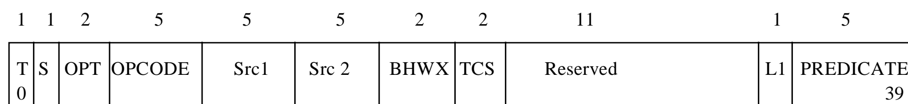
a) Original ADD Operation



b) Tailored ADD Operation



c) Original Store Operation



d) Tailored Store Operation

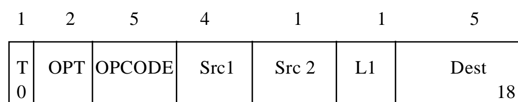


Figure 2.5 Tailored Encoding Example

encoding method still gets significant space savings compared to the original instruction set architecture, but avoids the intermediate decompression stage entirely. As a tailor-encoded instruction is decoded, the core processor's internal control signals are obtained directly. In this study, the Verilog code for the decoder is produced by the compiler and can be used to create

real hardware structure.

The algorithm for generating the Tailored encoding is rather straightforward. First, the entire code segment is 'logically' profiled/analyzed. The profiling information includes the number of operations used within each operation type, the maximum number of registers simultaneously live, the size of absolute field values (like addresses and immediate values included into operation body), and some others. If for example the application uses less than eight floating-point operations the floating point OpCode field only needs three bits. If a literal

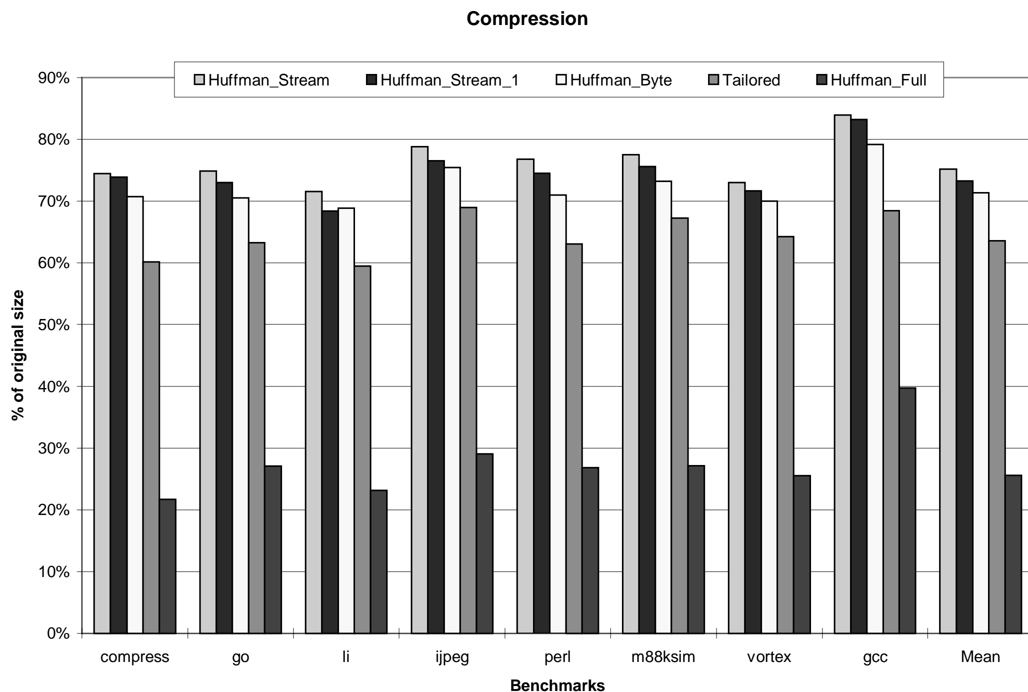


Figure 2.6 Comparison of Different Compression Techniques (code segment only).

in an immediate field of some operation exceeds some threshold value (16 bits in the current study), this operation is separated into a Load Immediate instruction followed by the original operation with register use instead of an immediate value. Similarly, after the register allocation, if no more than four different registers of some type are live at the same time in some source position, the source position needs only two bits to encode. The result is an

uncompressed, but *compact* version of the original program nearly optimal for this particular application (see Figure 2.5).

As have been mentioned at the beginning of this section, while forming the Tailored instruction set architecture, some enhancements are possible for the decoding stage. Indeed it is a big issue, because if we overly aggressive follow the minimum requirements for operation field we might end up with as many unique encodings (or masks) as there are different instructions in the application. That is why some size reduction needs to be sacrificed to guarantee interpretability of the customized code. For instance, if every instruction has its Tail bit, OpType and OpCode fields (see Figure 2.5) in a fixed position (and possibly of a fixed size) within the custom operation, decoding is significantly simplified. In addition to this, those fields could be grouped together for a MOP and placed at a certain location. Since the compiler is the one who generates the encoding and decoder, it looks for opportunities like this when it creates the Tailored ISA.

A comparison between all of these methods is presented in Figure 2.6 for the code segment only (see Section 3.3 below for more discussion). It should also be noted that for the stream based encoding, in addition to just choosing different fields for a stream, we could permute and combine them. This could potentially result in *N!* different combinations, where *N* is number of bits in uncompressed instruction (40 bit for the TEPIC). In order to select the optimal stream for *all* the benchmarks a variation of genetic algorithm have been used. From all the considered possibilities, only six different stream configurations were selected, and the two best performers are shown in Figure 2.6. The streams presented here were selected for the smallest average code size (*stream_1*) and for the smallest average decoder size (*stream*) among considered variations.

While analyzing the compression study results, several interesting points could be noted. The firs, and the most important one is that a significant amount of redundancy does exist in the code segment. It is quite understandable since an ISA is designed for a broad range of applications, and the application under consideration might use only a small fraction of the presented possibilities, it is intuitive that redundancy would be present. The second obvious conclusion is that not all compression algorithms are equally successful in removing this redundancy. The traditional byte-wise compression method could be used as a base line to evaluate others. This algorithm views the image as a byte stream with no other considerations, which yielding an approximate 25% size reduction. The antipode of the byte-based compression method is the Full Huffman encoding scheme, which compresses the image on an operation per operation level (40bit at a time). Note the remarkable code size reduction with the Full Huffman compression scheme (less than 30% of original size or 70% reduction on average). As have been mentioned before, this method approaches the entropy limit of the program's information contents. But, as we will see in Section 2.7.3, it produces a very large decoder, which in turn might prevent the use of this algorithm as the primary compression algorithm. This fact leads us to the final conclusion that there exists a tradeoff between the degree of compression and the complexity of the decoder. The tailored ISA approach produces code on the order of 64% of the original size, which can have favorable results for very little additional hardware overhead, so it is represents a middle point between high complexity of decoding and low compression effectiveness.

The results presented in the Figure 2.6 neglect the branch target table overhead, which is an integral part of the compression scheme. The goal thus far has been to address the compressibility of the code segment and reachability of the entropy threshold. All

implementation details are discussed in Section 2.5 below.

2.5 Instruction Fetch Mechanism Issues

2.5.1 Instruction Fetch Organization and Modification of the Instruction Cache

A significant component of this work is the joint consideration of the instruction encoding and instruction fetch organization. Once original ISA has been modified the whole instruction fetch pipeline must be adjusted. In order to interpret new instructions, the core decoder must be changed, cache design adjusted and system bus design reconsidered. All of it is

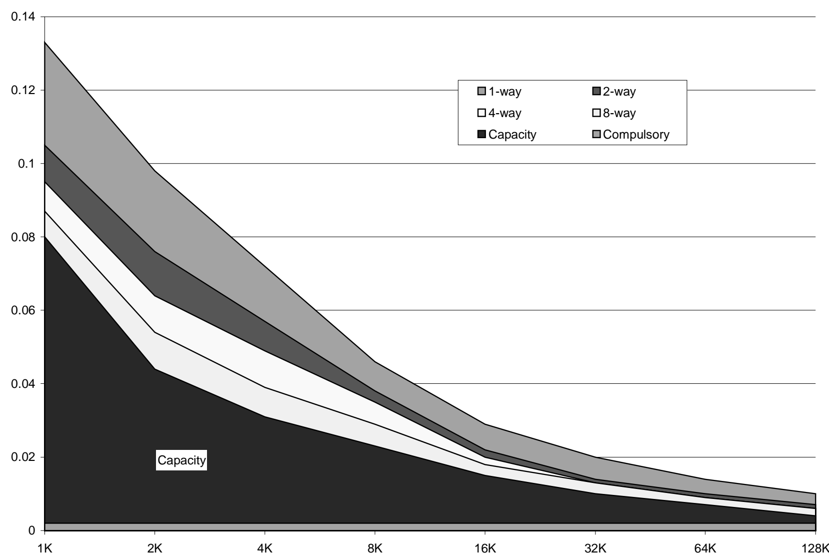


Figure 2.7 Traditional Distribution of Miss Rate

done with the help from the compiler. But most importantly of all those components, the organization of instruction cache must be reconsidered. In order to utilize the new low-entropy encoding throughout the whole instruction fetch pipeline (and not only as ROM size-reducing

technique) the instruction cache must be allowed to hold newly encoded blocks of instructions. For simplicity of discussion let us disregard the method of encoding and just call those instructions compressed.

The fact that the instruction cache holds *compressed* instructions increases its capacity and, as a result, the overall throughput. It is important to stress that we are breaking a fundamental limitation of current cache technology: the capacity miss ratio which is normally only attempted to be approached by a multi-way associativity and similar improvements. Figure 2.7 is adapted from Gee et al. [31]. The figure presents a breakdown of a miss component for caches of different associativity. The fundamental limit of a cache is compulsory (or first-seen)

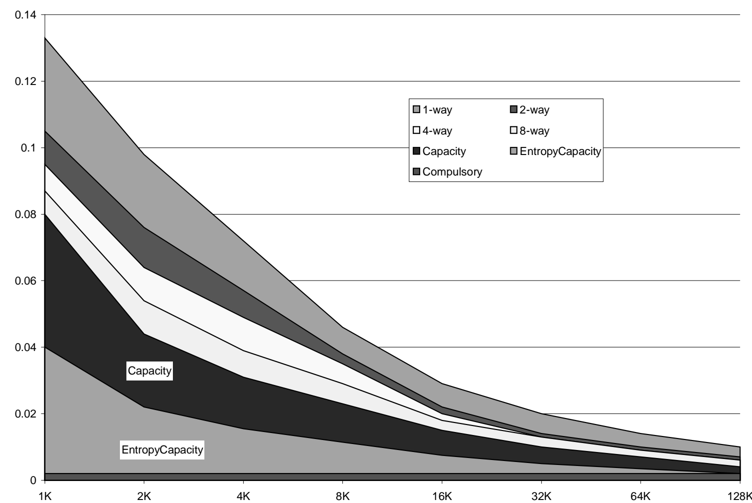


Figure 2.8 Entropy Based Distribution

misses. Compulsory misses cannot be eliminated, unless some sophisticated prefetch scheme is used. Traditionally, the next limit was always considered the capacity misses, which only depends on the physical size of the cache storage. Finally, conflict misses are normally defeated by a higher associativity and intelligent replacement policies.

Now with the compressed cache storage we are approaching the next fundamental limit:

the *entropy capacity* (see Figure 2.8). In this notation, no storage is wasted due to the low entropy of data being stored. This fact leads us to a paradoxical conclusion that we can improve the overall performance of a system by applying the compression technique, even though more work must be done to interpret the code. A disadvantage is that the cache controller needs to be designed differently to handle the compressed contents. However, on the positive side, the cache's data path design is not dependent on any particular encoding and could be generalized. This independence in turn makes modular core processor design possible.

2.5.2 Program Layout

Let us now define *an atomic fetch block* as a sequence of instructions guaranteed (or likely to be) executed sequentially once we start execution of the first instruction in the block.

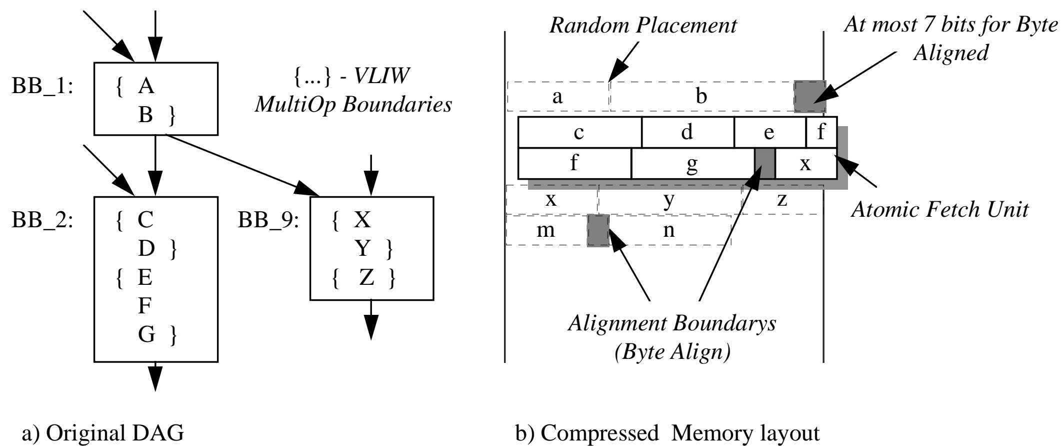


Figure 2.9 Atomic Fetch Block Structure

The simplest example of an atomic block is the *Basic Block* (a region of code with a single entry

and a single exit point). More sophisticated examples include a sequence of basic blocks with no side entrances, but multiple side exits (like *Superblocks* [22] or Fisher-style *Traces* [16]). Let us consider the simplest type for now, the basic block (BB).

As have been said before, the basic block can be treated as an *atomic unit of instruction fetch* (see Figure 2.9). This implies that cache can be accessed initially for only the first operation in the basic block. After this, the cache can supply operations in a streaming (pipelined if needed) fashion, until the end of the basic block is reached. This approach is completely transparent for the processor – it might keep on supplying each MOP address to the cache, but the cache controller does not need it to serve a miss. It starts to issue the MOP that is only going to be requested by the processor in the next cycle. This short term 'looking ahead' is a valid approach for the following reasons. First, control transfer can only occur to the first operation of a basic block (branch target). Second, a basic block should always be executed from the beginning to the end unless an interrupt has occurred, and even then its execution will be completed after the interrupt has been handled (here subroutine calls are considered to be branches that end a basic block). All the necessary NextPC computations local to the basic block are done within the cache, and are insignificant for the processor core, as long as correct VLIW group (MOP) is forwarded to the core decoder every cycle. Nevertheless this mechanism might be implementation specific and could depend on each particular embedded system architecture.

The use of more complicated blocks as atomic units is a matter of performance, not correctness. If the block is permitted to have side exits, we should guarantee that they are not taken frequently (or the instruction cache will get over-polluted). This requirement is true for

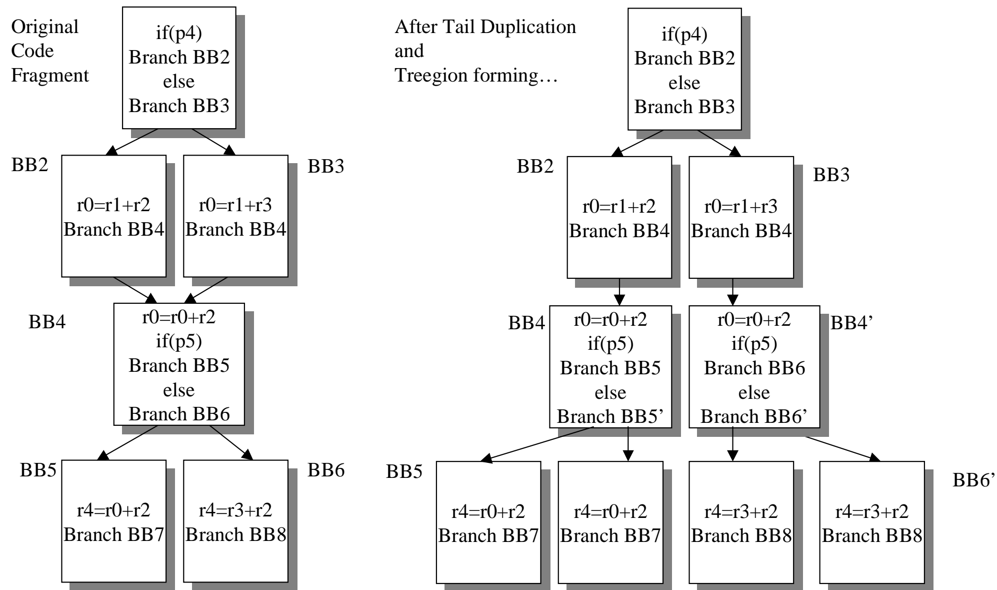


Figure 2.10 Treegion forming Example

superblocks [22] and Fisher-style traces [16], which are formed at compilation time with the use of profile information. But it is also true that for complex blocks some additional invalidation mechanism is needed. Nevertheless, in this study, only basic block atomic units are considered. (Note however that the code was scheduled by first building trees of basic blocks [i.e., treegions,] and then decomposed into basic blocks after the global scheduling pass.)

2.5.3 Compiler Optimizations to Enhance Code Layout

There are a number of possible code enhancements that could be performed in order to enhance the code’s compressibility and the instruction cache’s performance [36],[37]. They

include traditional optimizations as well as some specific actions. Since VLIW architecture chiefly depends on the compiler to achieve a high level of performance, it is essential to be aware of this matter during the scheduling.

The first enhancement is the Intelligent Code Layout to increase spatial reference locality. This optimization places the most commonly used sequences of basic blocks in close proximity of each other in the memory. In order to determine which basic blocks are more commonly used and in which order they should be laid out, the optimization requires some

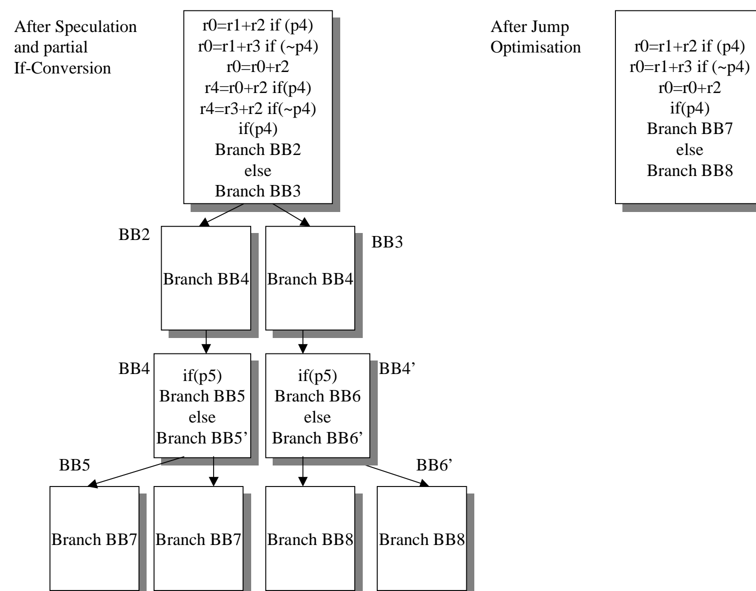


Figure 2.11 Jump Optimization Example

profiling information. This profile information is collected through execution of the application with some representative input data set and recording some run time statistics. The most important of those are number of times a basic block has been executed, and order in which most executed basic blocks were visited. In addition to that, if memory paging is used, the layout optimization also attempts to reduce the number of pages needed to execute commonly used parts of the program. This optimization increases spatial locality and is normally used to

increase instruction cache performance and has been proven to be effective.

The next set of compile time optimization is the *Jump optimization* and *Multi-way* branching. These optimizations are related to the code layout enhancement, but are more specific for the Treegion scheduling.

As have been mentioned before, the LEGO optimizing compiler conducts aggressive static scheduling of VLIW code. An integral part of the scheduling process is instruction *speculation* [37],[36]. Sometimes after instruction speculation by the scheduler, some basic

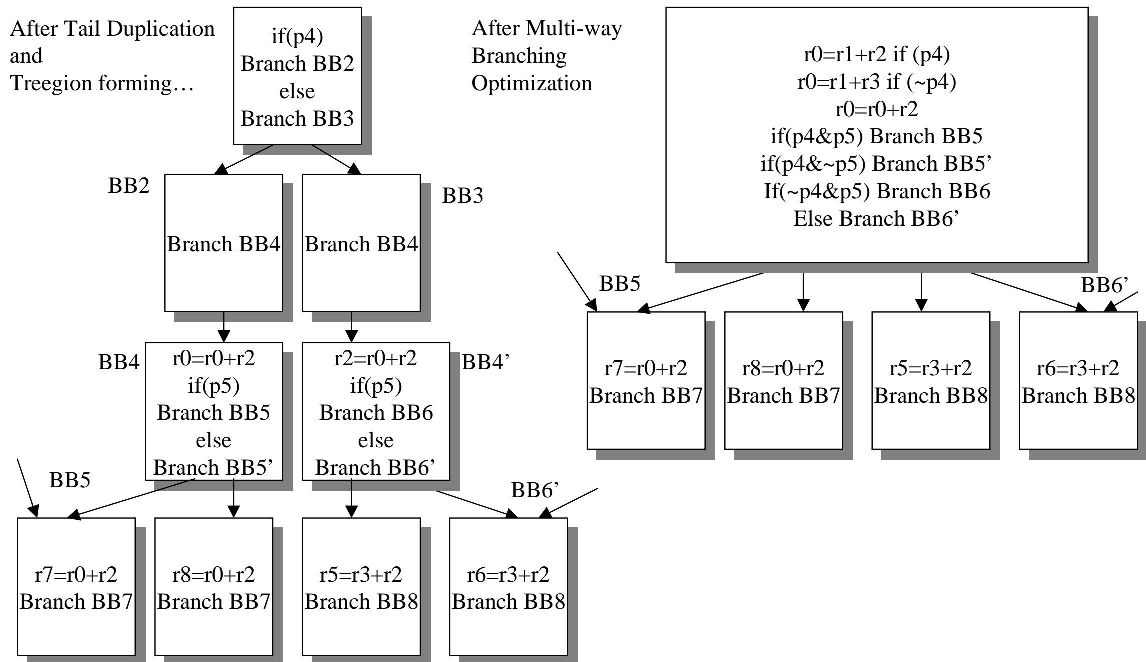


Figure 2.12 Multi-way Branching Example

blocks 'loose' all of their instructions except for the branch (see Figure 2.10 and Figure 2.11). This loss leads to multiple 'back to back' branches, which are hard to handle in the instruction fetch pipeline and often are logically redundant. Jump optimization tries to replace long chains of jumps (with no computations in between) to shorter ones by removing redundant branch instructions (see Figure 2.11).

The multi-way branching on the other hand allows more then one branch to be executed in a single cycle. For a VLIW architecture this branching scheme allows multiple branches a in a single VLIW instruction with priority given in left-to-right order. For control flow graph (CFG) multiple branches translate into multiple control edges from a single basic block. Since each branch instruction has an explicit conditional register (a predicate) associated with it, the sequence of branches is guaranteed to execute correctly. Besides obvious performance enhancement from these optimizations, they allow a reduction in the total number of basic blocks in the program which directly correlates to the size of the static address translation tables as will be described shortly.

Standard optimizations like common subexpression elimination (CSE), strength reduction and constant propagation [36],[37] generally contribute to logically compact code and undoubtedly are important for the current work. For example strength reduction might substitute a complicate uncommon instruction by a sequence of simpler, more common operations. Normally, all of these optimizations are performed prior to scheduling the code.

On the other hand, in the context of code size reduction, many of the traditional optimizations like loop peeling and unrolling [36] become less favorable. It is an important tradeoff between extracting or increasing the available instruction level parallelism (ILP) in a program and keeping the program size moderate. Since our primary goal in this study is the static code size reduction, no loop unrolling was performed.

2.6 Address Space Conversion

2.6.1 Branch Target Address Randomization

A critical issue for the execution of any compressed program is the change in branch

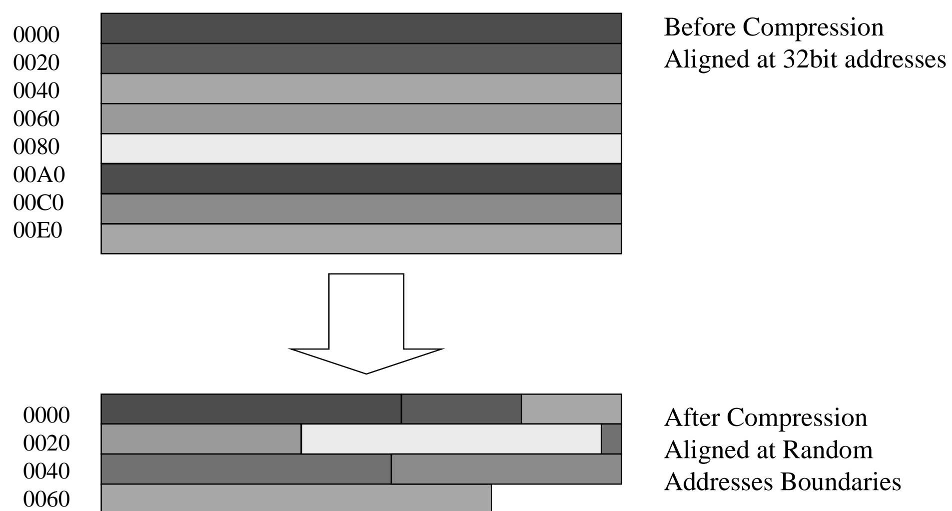


Figure 2.13 Branch Target Randomization

target addresses [1],[27],[42]. Every attempt to bound compressed instruction location to certain boundaries constrains compression. For instance, if the first instruction of a basic block would be aligned to the nearest *byte* boundary, compression degradation will range between one and three percent. If every instruction would be bounded, compression degradation would become unacceptable (more than ten or fifteen percent). If a high degree of compression is desired, each option must be considered and least bounded scheme selected. Once this is accepted, it should be realized that once different instructions obtain different length of codes, the original *branch targets* become meaningless. In fact the address space of a compressed code segment with

unbounded compression is absolutely chaotic (see Figure 2.13). Clearly, some kind of branch target address recalculation or translation must be performed.

First and the simplest solution is to convert the original branch targets to the compressed targets at compilation. This process could be performed in two passes. In the first pass a new code layout and new target addresses are generated (with enough space left for later 'plug in' of

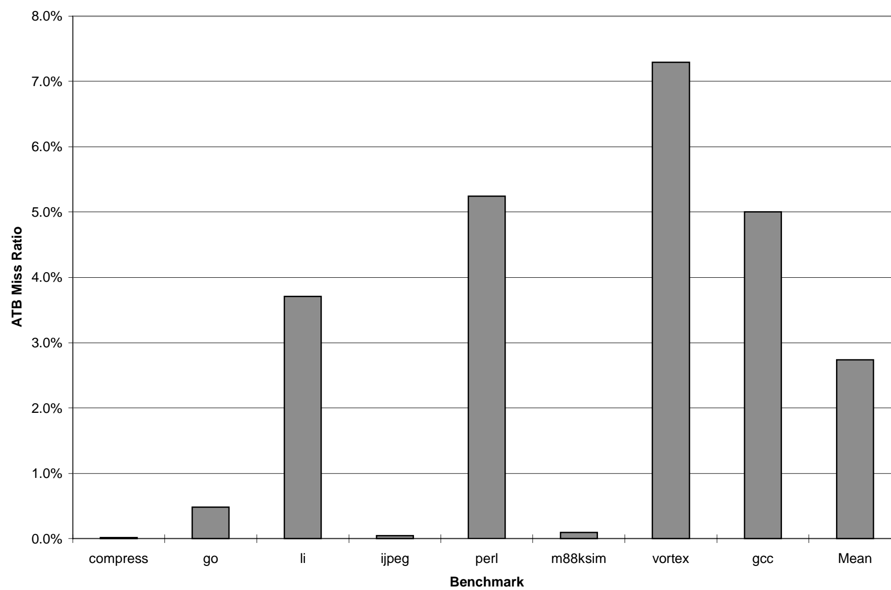


Figure 2.14 ATB Miss Ratio

new targets in relative branches). On the second pass, new addresses are 'plugged' or 'inserted' into the target slots and jump tables are updated. This method is a better fit for the Tailored ISAs than for code compression schemes, because compressed code with new targets will have to be recompressed with certain restrictions. Branch instructions could also remain uncompressed in which case a special 'escape' symbol should be added.

Another solution to the branch target problem is to leave the original target addresses the way they are, unchanged (just compress them along with the rest of the code) and provide a *dynamic* translation mechanism at run time. This approach is very well known in general purpose computing for the mapping of virtual address space to the physical one via the

Translation Lookaside Buffer (TLB). Similar hardware named the Cache Lookaside Buffer (CLB) is also employed in studies by Wolfe, et al. [1],[17] and has proven to be effective. We use a similar approach to map the original address space into the compressed space with aid from the compiler. The hardware structure is called the *Address Translation Buffer* (ATB) and the static table is the *Address Translation Table* (ATT). The ATB holds pairs of addresses, which maps the original address space to the compressed space along with information to aid decoding, decompression and Next PC computation. The ATT has *one entry* for each atomic compression block (currently a basic block). ATT is generated by the compiler and stored in memory in compressed form. The additional information stored in the ATB includes the number of memory lines that need to be fetched in order to get the whole block, and the number of operations in the block (or simply the number of VLIW multiops in the block. Portions of the ATT are uploaded to the ATB as needed. Due to the normally high spatial locality, the ATB has very low level of contention (see Figure 2.14) and the ATT has a tolerable static size (see Table 1). The Table 1 shows the results for Tailored instruction set encoding only. The overhead results for custom compression schemes are similar since they are not dependent on the compression algorithm employed. Nevertheless, the ATT does add some overhead to the final storage. In general, when number of basic blocks is not optimized, the ATT adds on average 15% to the compressed size of the ROM. This fact calls for a solution to minimize its size. As have been discussed before, the easiest way to reduce the size of ATT is to minimize the number of atomic units in the code through a compiler optimization known as the multi-way branching (this optimization was described in greater details in section 2.5.3) or use different atomic blocks. When the multi-way branching only is performed, the total overhead of the ATT table is reduced to 11%. If this optimization would be combined with a different atomic block

granularity, the overhead could be reduced even further, but this process needs a deeper investigation and is rserved as a future work.

Table 1 ATB Characteristics for Tailored ISA compression

	ATT Entries	*ATB Miss ratio*	*ATT Size (compressed, bytes)*	*Tailored ISA Code Segment size*	*Degree of Compression without ATT (%)*	*Degree of Compression including ATT (%)*
compress	352	0.0016	1,223	5,260	60.16%	74.14%
go	14,036	0.0029	51,853	199,420	63.32%	79.78%
li	4,027	0.0629	12,879	44,484	59.49%	76.72%
ijpeg	8,792	0.0004	32,570	176,252	68.95%	81.69%
perl	18,130	0.0764	72,012	259,736	63.04%	80.52%
m88ksim	8,413	0.0012	30,241	148,752	67.25%	80.92%
vortex	30,699	0.0830	117,421	629,636	64.25%	76.23%
gcc	98,564	0.0842	404,664	1,468,500	68.50%	87.37%
Mean	22,877	0.0018	91,758	366,505	66.47%	83.11%
				Average:	64.60%	80.05%

Briefly, at run time the ATB will provide the following information: the address of the requested block in compressed memory, the PC offset of the last operation in the block, and the predicted PC of the following fetch block. This information is enough to fetch atomic blocks in a pipelined fashion. There is also a clear tradeoff between the amount of additional information

in ATT and the compressed instruction cache performance. All the above-mentioned information could be deducted at run time and cached in ATB. If this would be done, the ATT size overhead would go down to approximately six percent. Nevertheless the rest of this work assumes that the support information is present in the ATT.

The next issue in this category is physical ROM access. The vast majority of modern

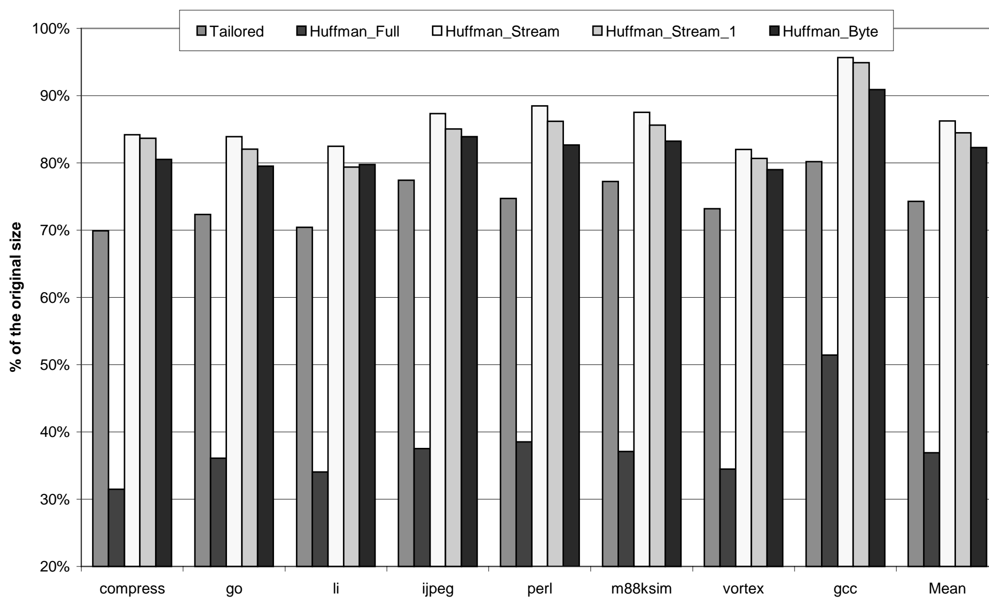


Figure 2.15 Compression Including ATT Size

memory systems support only *byte* or *word* aligned accesses. The access granularity could also be limited by the address bus width and should not be sacrificed. This puts some, but non-critical, limitations on code placement in the compressed storage. This issue is addressed by *aligning only the first operation* of a block to the physical ROM access boundaries. For the current study these boundaries are assumed to be byte aligned. This assumption means that if an atomic unit ends at a middle of a byte of storage, it is padded with up to seven bits at the end

(see Figure 2.9 and discussion at the beginning of this section), so the first operation in the next atomic block is byte aligned. All consecutive compressed operations in the block are sequentially placed in memory. Padding add some overhead to the overall compressed segment size (from one to three percent on average) and should be factored in for overall size calculation.

After all of these 'extras' were factored into the general compression picture, and the Multi-way Branching and Jump optimization were applied to reduce the number of atomic compression blocks (see Section 2.5.3), the final compressions looks like follows: (see Figure 2.15). The relative performance of all algorithms remained unchanged and only a slight overall increase in size could be seen. Nevertheless, this fact might encourage us to re-evaluate some of the conclusions. The tradeoff between decoding complexity and the degree of encoding is still factual. But another component needs to be factored into the overall equation – the extra information needed for compressed instruction cache operation if it has to be stored along with the compressed code segment (as in the case of the ATT table).

2.6.2 Base Line Instruction Cache Design

The instruction cache is a critical element of any high performance system, and it is especially important for this study. The main instruction cache tradeoff in a compressed system is the address space to which it belongs. In other words, whether the cache holds compressed or uncompressed operations (see the discussion at the beginning of 2.5, Section 2.5.1). Most of the researchers [1],[9],[10] uncompress their instructions prior to putting them into the instruction cache. This decision normally allows them to hide the performance penalty associated with the decompression of encoded instructions. But as have been mentioned previously in Section

2.5.1, the compressed cache is able to hold several times more instructions than an uncompressed one. (See Figure 2.7 and Figure 2.8) The only problem is that some work should be performed at the hit path, which potentially increases the branch missprediction penalty (if instruction fetch hierarchy is pipelined) or stretches the cache access cycle time. Since the vertical size of the cache main storage can be reduced now with no loss in performance, cycle time stretch is less likely. Nevertheless, this study assumes pipelined hierarchy and further concentrates on keeping the pipeline full at all times.

The next important issue is the NextPC calculation. A cache that supports a zero-NOP encoding employs a NextPC calculation mechanism [8],[7] that is applicable to this study as well. Let us differentiate the NextPC *within a block* and the NextPC of the *next block*. The

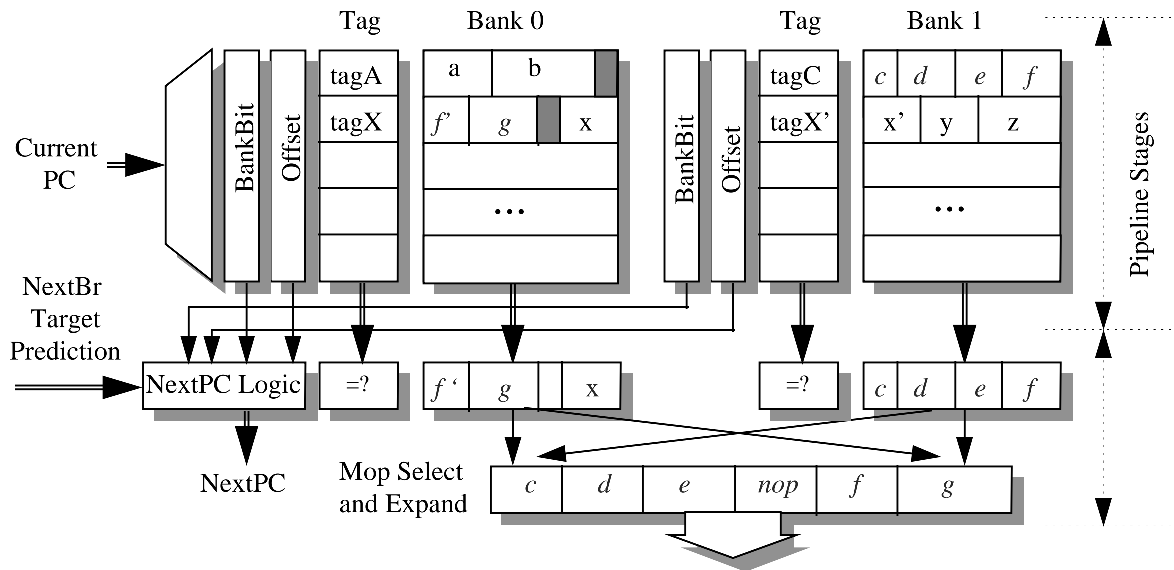


Figure 2.16 Banked Cache Architecture

NextPC within a block does not need to be predicted (since by definition we are going to fetch the block till the end) but rather can be locally calculated with dedication of some additional hardware [8]. This hardware, along with the access pattern, varies with *placement* and

invalidation policies. If a block is *atomically* (in unbroken form) placed in the cache, an intermediate instruction access does not have to be checked for validity. If a block id displaced from the cache, it is invalidated entirely, so no partial hit is possible. This will be called the *restricted* placement model.

If pieces of the block (like individual encoded instructions) are allowed to be scattered around the storage, cache controller needs to generate an intermediate PC within each multiop, and locally 're-access' the cache to check if we have the valid data present. The invalidation is possible on individual instruction level, so a partial miss might occur. In the latter case, all additional information (individual MOP length in terms of encoded instructions for instance) could be extracted from a fetched block at miss-repair time and stored in the cache with association to the first instruction of the MOP. In the tailored ISA approach, it is especially easy to do since the size of all operations of the same type and opcode is the same, and the location of this information is fixed within an operation (see Section 2.3). For the Huffman compressed encoding approach, this information might be generated by the compiler and stored along with ATT in compressed form (which will further increase the space penalty). For all the outlined reasons, and for sake of relative simplicity, in the current study we only consider the restricted placement model.

The Next PC of the *next block to be fetched* is the more traditional branch target problem. This address needs to be dynamically predicted if we want to achieve full capacity for the instruction fetch pipeline. This prediction is even more important if a sophisticated (like in our case) instruction fetch is used (which means longer pipeline and higher missprediction penalty). In the current study, we have coupled the branch prediction table with the ATB. This combination means that for every block entry, there is one branch predictor with *taken/not-taken*

and *target address* prediction information. It predicts the outcome of the last instruction in the block (which by definition is always a branch, see Section 2.5.2). To predict the outcome of the branch, a simple two-bit saturating counter is used [13]. To predict the target address, the 'last-target address' (if branch predicted taken), or next sequential address (otherwise) heuristic is employed. We have to keep the last taken address of the branch locally because we cannot wait for the branch instruction to get decoded (which happens much later in the processor pipeline).

Theoretically, a more complex branch predictor could be used (e.g., gshare or PAs Yeh/Patt predictor) since there will most likely be several cycles to access the prediction, unless the code has multiple sequential branches with no other computation in between. But this option is not considered at this point of time and is reserved as future work.

The baseline cache that was selected for this study is the *Banked Cache* described in [7],[8]. Originally designed to fetch variable length MOPs for Zero-NOP encoding, it fits to all the requirements outlined earlier: fetches unaligned blocks, could be pipelined and wastes no storage. The structure of it is depicted in Figure 2.16.

The storage of the Banked Cache is separated in two banks, similar to that of the Intel Pentium processor [20]. The cache line size is equal to the maximum size MOP, which in turn is proportional to the issue width of the processor core. In our case, it is a six wide issue TEPIC VLIW core. A MOP can begin at an arbitrary location in the bank and span two cache blocks, but it still can be extracted in a single reference to the bank storage (see the MOP 'cdefg' in Figure 2.16, which spans form the Bank0 to Bank1; the shaded region correspond to alignment padding for the first operation in MOP). Two sub-blocks are brought down to the *alignment* stage on a cache hit -- the block that was referenced by PC, and the next sequential blocks. If for example the beginning of a MOP resides in the Bank1 at index *N*, the next sequential block is

brought from the Bank0, index *N*+1. The MOP is guaranteed to be within these two cache sub-blocks because their combined size equals to the maximum sized MOP. Then the alignment hardware scans individual operations in parallel, but with left-to-right priority for the Tail bits (see Section 2.3). Then it extracts the original MOP (see Figure 2.16). The whole process from receiving an original address till the NextPC calculation is accommodated in two pipeline stages – storage search and tag match followed by alignment network operation. More details on the design and tradeoffs of the base line Banked Cache can be found in [7],[8].

2.7 Compressed Instruction Cache Hardware Implementation

2.7.1 The Instruction Cache Design for Compressed Encoding

The implementation of the instruction cache for the compressed encoding is designed to reduce the impact of decompression time on the instruction fetch rate. One atomic block is decompressed at a time and is held in a buffer, which is accessed in parallel with (but has a priority over) the main cache. This buffer is organized as a small fully associative cache. In general, the whole structure could be seen as a two-level instruction cache, where decoding is done at miss time of the L1 cache and the buffer is, in essence, an L0 cache. This organization makes sense if we analyze the overall situation from a standpoint of data usage. Since code is compressed based on the static distribution and does not take into account the frequency of certain block usage we might have suboptimal performance. It might occur when the most commonly used block is present in the cache, but is kept compressed throughout access time. Even with the high rate of following block address prediction, we will have to perform

decompression again and again, and even though it might not slow down execution (due to pipelining of the whole process), it might affect power consumption. For these reasons we are trying to keep the most frequently used block in uncompressed form in the buffer. It also should be remembered that by the block here we mean the atomic unit of compression, which in this

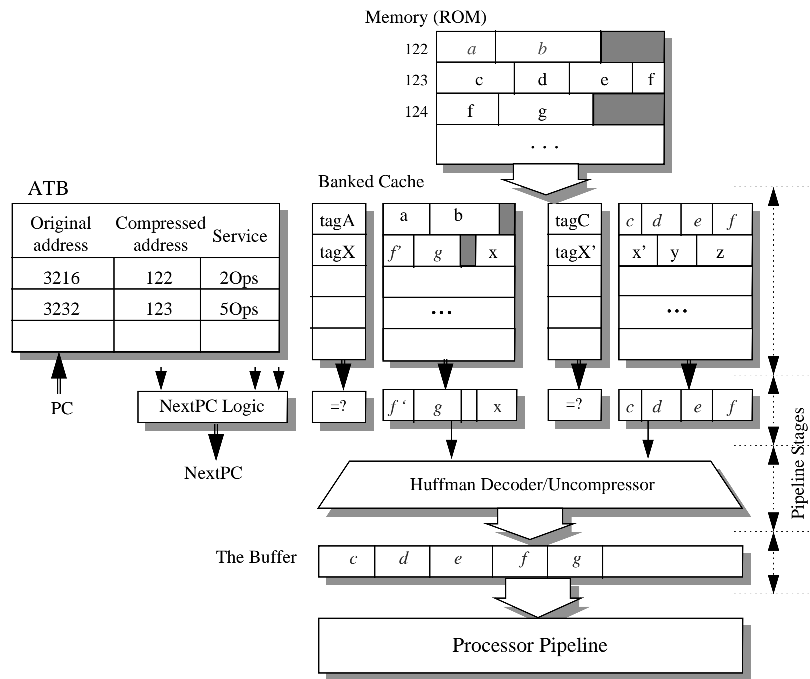


Figure 2.17 Instruction Cache Structure for Compressed Encoding

case is a basic block. The buffer is not explicitly purged when new block is placed in it, so it potentially can hold several commonly accessed basic blocks.

The main cache storage is organized the same way as the Banked Cache described in Section 2.6.2. The size of the buffer (L0) has been set at 32 operation entries (160 bytes for the operation size of 40 bits). From the performed experiments there are indications that tight, frequently executed loops (like DSP kernels) can fit into this buffer completely, which will result in equivalent (or possibly better) performance (access time) to an uncompressed cache. Nevertheless it is very important to keep the size of this buffer at minimum in order to not

compromise the performance of the main compressed storage. In addition to that, some researchers [24] indicate that similar two-level architecture organization might contribute significantly to low-power design, since the buffer cache filters out power-consuming accesses to the larger L1 cache. The structure of the entire system is depicted in Figure 2.17. The pipeline stages are outlined in the diagram. The worst hit time for a compressed block that is located in main storage and needs to be uncompressed is three cycles. The detailed cycle count assumptions could be found in the Appendix Table 2.

The cache has the above-described mechanism for dynamic address translation (ATB) and coupled branch predictor. It is important to notice that the location of this branch predictor differs from the conventional. In a majority of today's systems, branch prediction is performed

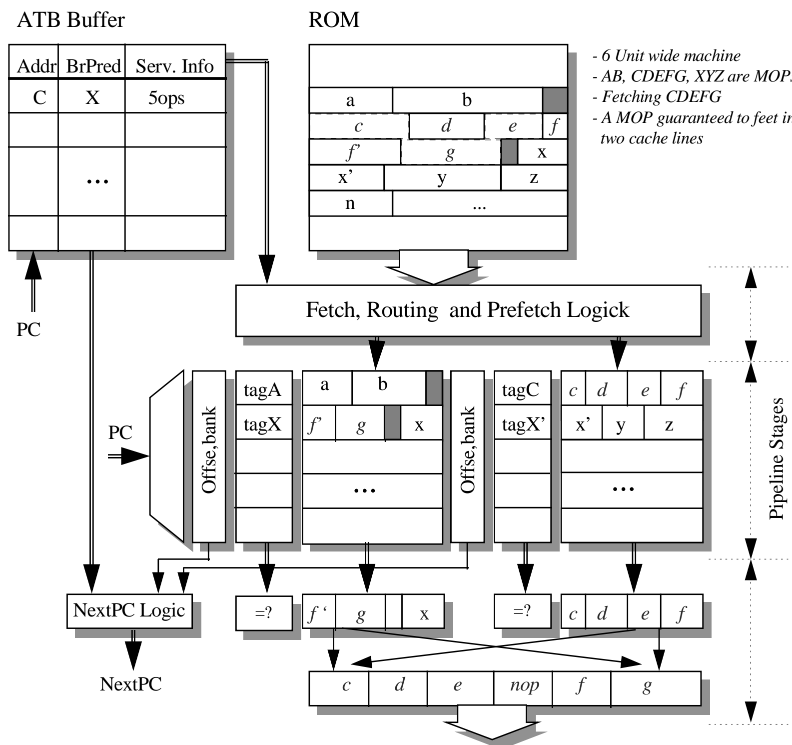


Figure 2.18 Instruction Cache Structure for the Tailored Encoding

later in the pipeline. Often it is no earlier then the decode stage – the predictor is accessed once

it is known that the instruction being decoded is a branch. If the branch predictor is accessed with address of every instruction, it is destined to be polluted and has to be of a large size to guarantee low level of conflicts. In this case the branch prediction is moved all the way to the instruction cache. Since we know that the last instruction of an atomic fetch block (basic block) is a branch, the branch predictor could be indexed with the address of first instruction in the block, without increasing aliasing or causing conflicts. The NextPC calculation is identical to that of the base line banked cache (described in Section 2.6.2). The only difference from architectural standpoint is the 'black box' – the decompressor being added between the main storage and the buffer.

2.7.2 The Instruction Cache Design for the Tailored ISA

The objectives for the tailored ISA cache are quite different from the compressed encoding cache. The operations are stored in a form ready for consumption by the core decoder. However, this cache also uses the Banked Cache as its core design element to guarantee single cycle access for unaligned MOPs. The key difference is the logic in the miss path, which is

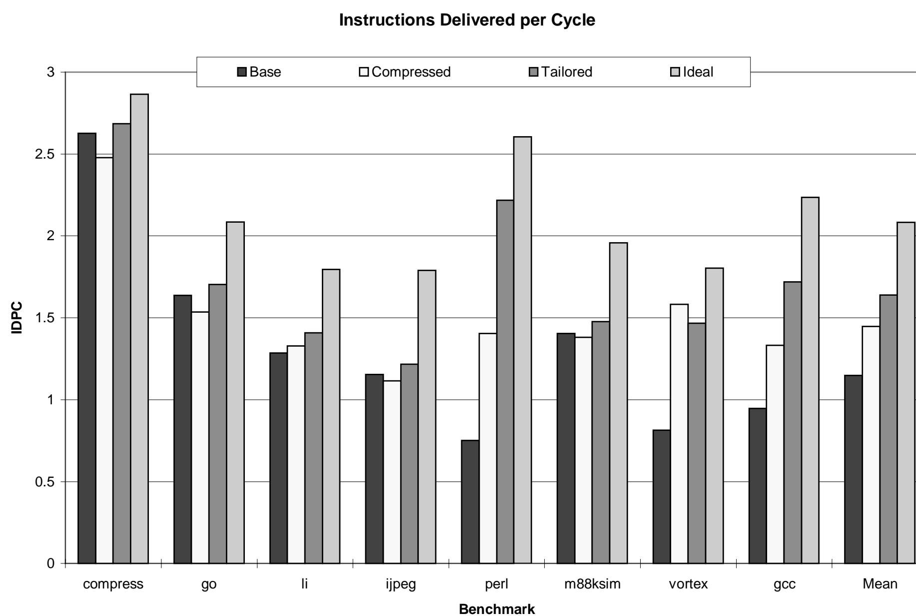


Figure 2.19 Cache Study Summary. Instruction Delivered per Cycle.

responsible for the extraction and placement of MOPs in the main banked storage. The overall organization is shown in Figure 2.18.

The hit path now has only one stage for the alignment of operations. Branch prediction is still used to ensure high pipeline utilization. From an architectural point of view, an extra stage is added on the miss path. (For the detailed summary of all the performance penalty assumptions, please refer to the Appendix, Table 1.)

For the experiments we choose moderately sized caches on a scale suitable for an embedded system: 16KB, 2-way set associative for both compressed and tailored models. The baseline cache has to have a block size that is a multiple of the TEPIC 40bit op size, so its

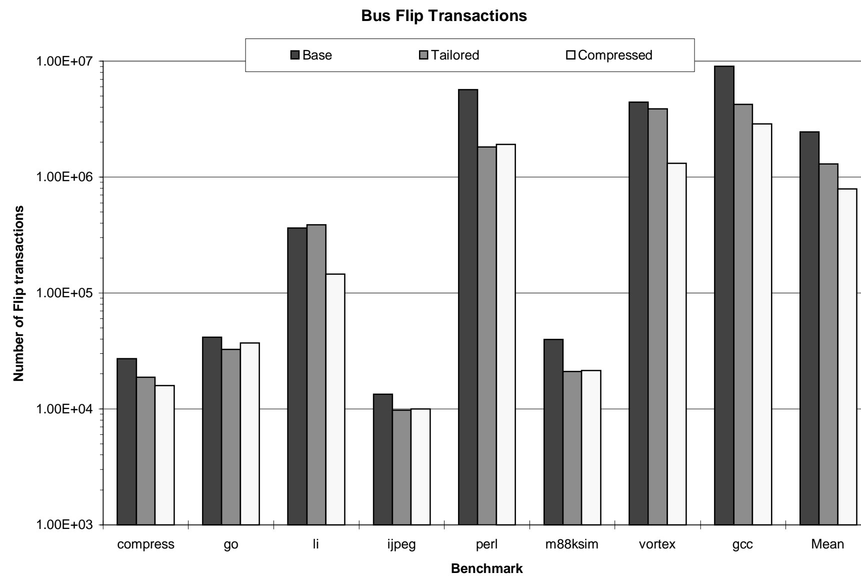


Figure 2.20 Instruction Memory Bus Traffic Summary

effective size is slightly larger: 20KB, 2-way set associative. All results are summarized in Figure 2.19. The metric is a measure of instructions (operations) delivered per cycle. The issue width for the core is six operations. The average for "Ideal" is limited by the quality of the schedule as well as ideal cache and branch predictor performance. The "Base" represents uncompressed code, whereas "Compressed" uses the Full operation compression scheme and "Tailored" is for Tailored ISAs. It is particularly interesting to note that both the Compressed and Tailored exceed Base on average, although the Compressed does poorer than Base for several benchmarks (compress, go, ijpeg and m88ksim). This decrease in performance is due to the higher missprediction/miss repair penalties for Compressed compared with Tailored. Introducing a larger buffer size and more accurate branch predictor could significantly increase the performance of the compressed cache model, but that was not the main goal of the

experiment.

Another interesting result is the change in the amount of bus traffic due to instruction cache misses. It is one of the defining factors of power consumption, especially if the ROM is

```
module custom_decoder (clock,code_bus,h_bus,opc_bus,src1_bus,src2_bus,
            sp_src_1_bus,sp_src_2_bus,sp_src_3_bus,dst1_bus);
input                   clock;
input       [31:0]      code_bus;
output                  h_bus;
output      [5:0]       opc_bus;
output      [7:0]       src1_bus;
output      [8:0]       src2_bus;
output      [2:0]       sp_src_1_bus;
output      [2:0]       sp_src_2_bus;
output      [2:0]       sp_src_3_bus;
output      [6:0]       dst1_bus;
reg         [7:0]       src1_bus;
reg         [8:0]       src2_bus;
reg         [2:0]       sp_src_1_bus;
reg         [2:0]       sp_src_2_bus;
reg         [2:0]       sp_src_3_bus;
reg         [6:0]       dst1_bus;
            assign h_bus = code_bus[31];
            assign opc_bus = code_bus[30:26];
always@(posedge clock)
begin
            case(opc_bus)
            0:          begin           /* ADD */
                        src1_bus[7:1] = code_bus[25:19];
                        src2_bus[8:0] = code_bus[18:10];
                        sp_src_1_bus[2:0] = code_bus[9:7];
                        dst1_bus[6:0] = code_bus[6:0];
                        end
            21:         begin           /* MOVE */
                        src1_bus[7:0] = code_bus[25:18];
                        sp_src_1_bus[2:2] = code_bus[17:17];
                        dst1_bus[6:0] = code_bus[16:10];
                        end
……..
```

```
module huffman_decoder (code_bus,instruction_bus);
input         [14:0]        code_bus;
output        [8:0]         instruction_bus;

reg [8:0]  instruction_bus;

always@(code_bus)
casez(code_bus)
   14’b0zzzzzzzzzzzzz: instruction_bus  = 8’b00000000;
   14’b10000zzzzzzzzz: instruction_bus  = 8’b00000010;
   14’b1000100zzzzzzz: instruction_bus  = 8’b00001011;
   14’b100010100zzzzz: instruction_bus  = 8’b00110001;
   14’b100010101zzzzz: instruction_bus  = 8’b00010101;
   14’b10001011zzzzzz: instruction_bus  = 8’b00011001;
   14’b1000110zzzzzzz: instruction_bus  = 8’b00000111;
   14’b10001110zzzzzz: instruction_bus  = 8’b00011100;
   14’b10001111zzzzzz: instruction_bus  = 8’b00001001;
   14’b10010000zzzzzz: instruction_bus  = 8’b00000101;
   14’b100100010zzzzz: instruction_bus  = 8’b00011101;
   14’b10010001100zzz: instruction_bus  = 8’b00010100;
   14’b10010001101zzz: instruction_bus  = 8’b00011110;
   14’b10010001110zzz: instruction_bus  = 8’b10000100;
   14’b10010001111zzz: instruction_bus  = 8’b11110000;
   14’b1001001zzzzzzz: instruction_bus  = 8’b00000100;
   14’b10010100zzzzzz: instruction_bus  = 8’b00010010;
   14’b100101010zzzzz: instruction_bus  = 8’b00000110;
……..
   14’b11011111zzzzzz: instruction_bus  = 8’b11000000;
   14’b111zzzzzzzzzzz: instruction_bus  = 8’b00000001;
   default : instruction_bus = 8’b0;          endcase
endmodule
```

Figure 2.21 Verilog Code for Decoder Example (Custom – left, Byte Based Huffman – right)

placed on a separate die. In the experiments, power is modeled by counting the number of transactions on the memory bus when bits are flipped. With the increase of hit ratio, the number of blocks needed to be brought from the memory for miss repair drops. The summary of these changes is presented in Figure 2.20. The results track the degree of compression and show savings for the Tailored and Compressed encodings over the Base. From this it can be concluded that each of the compression schemes brings in more instructions for a given number of bit flips.

One interpretation of the combined results of Figures 9, 12, and 13 is that the Tailored

ISA encoding has more advantages than otherwise clear from the degree of compression data of Figure 5. Because an additional decoder is not required (as opposed to the Huffman-based schemes), there is a net savings in the processor core that can be significant. What is more interesting is that, although the Tailored encoding achieves a lower overall cache utilization, the missing extra cycle of branch missprediction penalty more than makes up for this absence in overall performance.

2.7.3 Decoding Complexity Evaluation

Since, in the case of code compression, we choose to place decoding on the critical path of the instruction fetch mechanism, it should be made as efficient and fast as possible. In essence, it is now the critical factor for the compression algorithm's selection, which directly

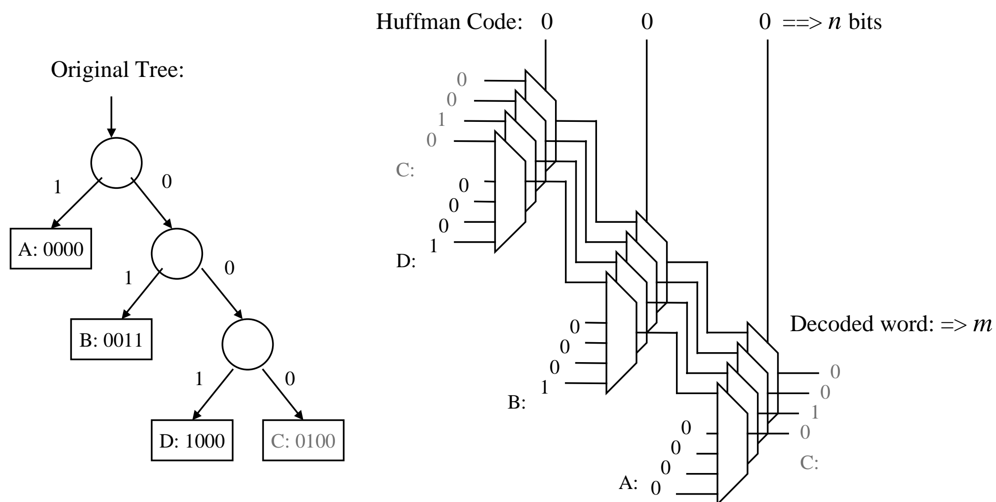


Figure 2.22 The Huffman Tree Decoder Structure

corresponds to the degree of compression (see discussion in Section 2.4). Fortunately, the compression algorithm is static in time. Once selected, based on static frequencies of elements in code segment, it remains the same. Therefore a fast fixed hardware decoder can be generated.

As have been mentioned before, the compiler has all the information needed and generates Verilog description (see Figure 2.21) of the decoder that could be used to program the PLA decoder or as an input for custom decoder design (after possible steps for optimization). In the case of the Tailored encoding, this problem is somewhat less critical. Decoding of tailored instructions is a part of the processor pipeline. Nevertheless, it might be more complex when compared to the traditional fixed-size-op decoder. An interesting consideration would be to combine the Huffman decompressor with the processor decoder on the logical design level, and optimize it as a flat logic. This approach might present more opportunities for CAD tools for optimization and result in a smaller overall decoder. There is only one consideration which prevents us from attempting it in the current study, – we want to affect the design of the core processor as little as possible (if any at all) so the instruction fetch process remains completely transparent for it. Nevertheless, this topic could be considered in future research, which would include fully customized processor core design.

Since, in this study, we generate a multitude of various decoders we need to establish a mean for their fast evaluation and comparison. As a method of comparison of the Huffman decoder size, we can evaluate the complexity of the correspondent *Huffman tree*. If we imagine the structure presented in Figure 2.22 (where n is the longest Huffman code size, k is number of entries in the Huffman dictionary and m is longest dictionary entry size), it is possible to derive an equation to estimate the worst-case decoder complexity. It is not intended to suggest a real hardware implementation, but only as a criterion for evaluation. The worst-case number of elements in a Huffman decoder can be expressed as follows:

$$T = 2m(2^n - 1) + 4m(2^n - 2^{n-1} - 1) + 2n \qquad \text{(Equation 3)}$$

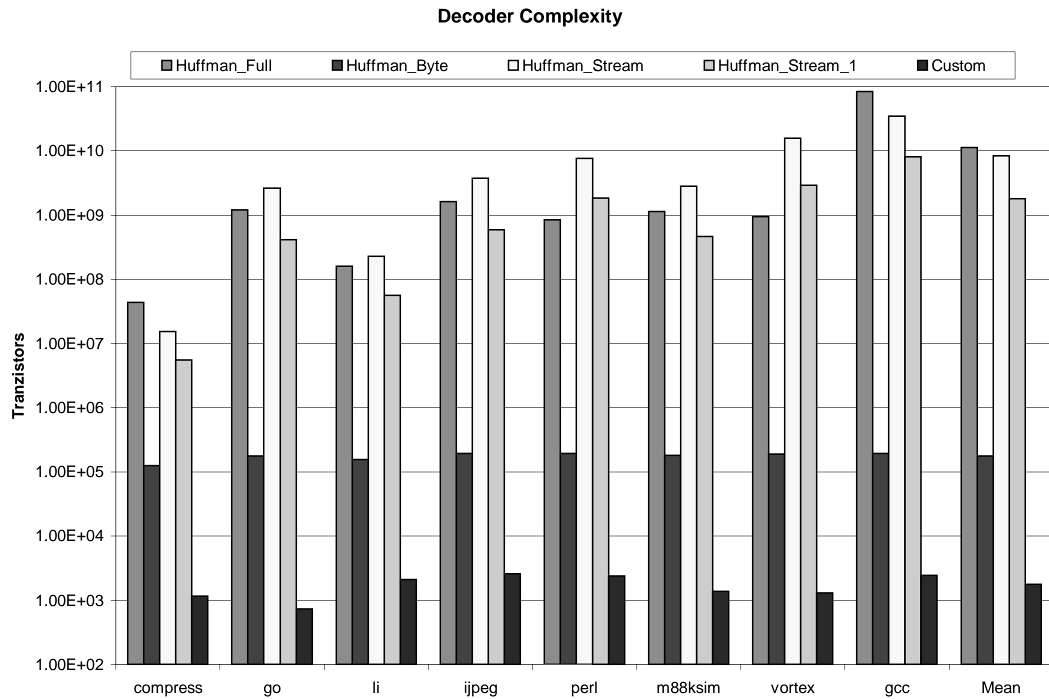


Figure 2.23 Estimated Huffman Decoder Complexity.

This equation assumes a multiplexer implementation using CMOS transmission gates (TG), which account for the fact that the first row passes constants and needs only one transistor to operate. Elements to form inverters (dual rails) are included as well.

Assuming this model we can evaluate complexity of the various Huffman decoders (see **Error! Reference source not found.**) without actually synthesizing every one of them. **Error! Reference source not found.** in conjunction with Figure 2.15 and the Table 1 allows us clearly see the tradeoff between the decoder complexity and degree of compression. The best compression algorithm (the Full Huffman) yields the largest decoder size. This relationship is

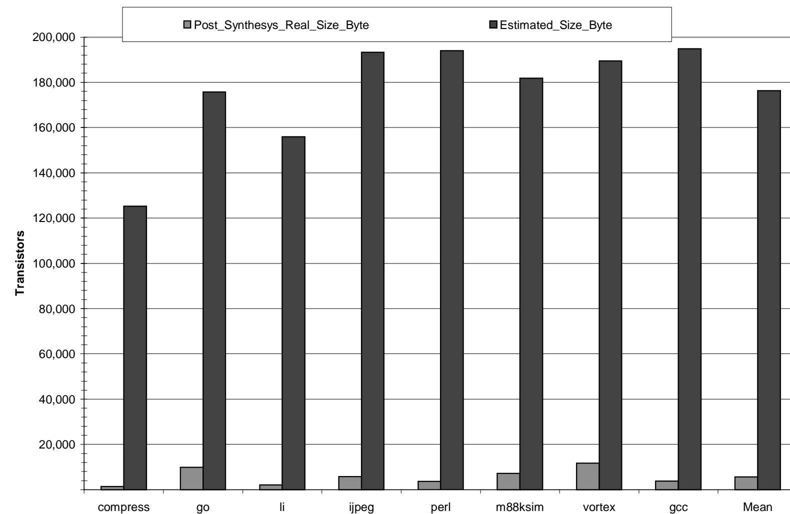


Figure 2.24 Estimated to Real Size Comparison for the Byte Based Compression Decoder (for the Compress Benchmark)

not necessarily linear. Byte-wise compression yields an intermediate degree of code compression. It is approximately 72% of the original image size yet has the smallest decoder (See **Error! Reference source not found.**). The worst Huffman compression scheme, the Stream, achieves approximately 75% of the original image size. Yet, it has a significant decoder complexity. The reason is the limited input width and dictionary size of the Byte method.

After the initial estimation, several decoders were actually synthesized with the Synopsys CAD tools in order to compare actual parameters with the estimated ones. As we can see in Figure 2.25 the real synthesized numbers are less then one percent of the estimated worst case,

which proves the possibility of such a design. Regardless that we miss the actual decoder size by 99% while estimating it we still can use the estimated value for *relative* evaluation of the design.

If we ever wonder into generating a *custom compression algorithm* for each particular application (as oppose to just using the same Huffman algorithm for all of them) we will need a *cost function* to describe all the parameters of the new algorithm. Definite components of the

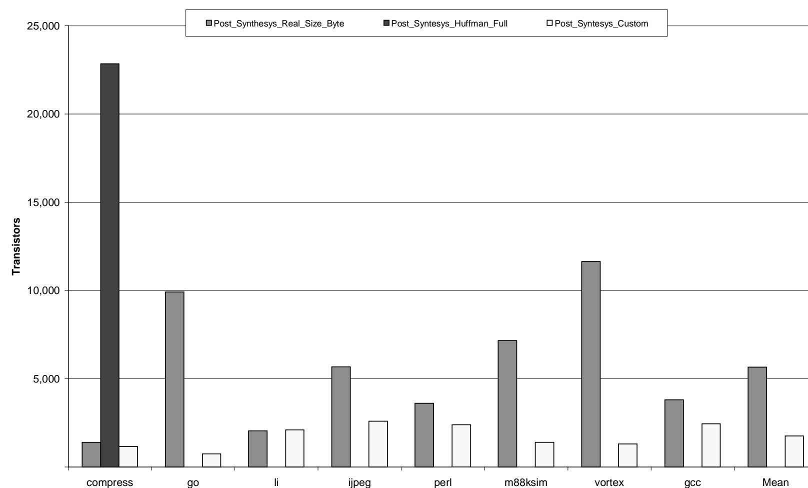


Figure 2.25 Estimated to Real Size Comparison for the Byte Based, Full Compression and Custom Coding Schemes (for the Compress Benchmark)

cost function are resulting code size, decompression speed/complexity, and the decoder size/complexity-weighted according to their importance. For one of the major parameters to this cost functions -the decoder complexity estimation – a similar function could be used. An interesting issue is estimating the speed of decoding, which is virtually impossible until real hardware is generated. Let us reserve this issue as a future work.

The fact that our estimate was so far from actual numbers is rather understandable for two reasons. First, the worst case is always pessimistic comparing to the actual decoder

structure and second, the design tools provide comprehended Boolean optimizations which further reduces the complexity of the decoder. The longest synthesized delay chain included, on average, five levels of logic, which promises a high speed of operation.

Several implementations of the Huffman decoder in hardware have been proposed in previous studies [17,18]. Both models are strongly dependent on specific implementation (MPEG-2 decompression for example), but generally can achieve 300-600 Mbit/sec for a table with 114 dictionary entries and codes in range from 1 to 16 bits. The real-estate budget ranges from 10,000 to 28,000 transistors. This data allows an assumption for the time needed to decompress the code. For the 20-50ns cycle times typical in embedded processors, we can assume decoding of 40 bits (op size in the baseline TEPIC architecture) is practical. Therefore, it is assumed that one op could be decoded in a cycle. Furthermore the decoding process is pipelined. Since instructions are supplied to the processor on each cycle we can keep this pipelined filled (warm) if we correctly predict the next sequential block after the current (being decoded). As we will see soon we couple the compressed cache design with a branch predictor, which aids in continuous (streaming) decompression.

3 Data Segment Redundancy Reduction

So far the author have only considered the code segment as the subject for redundancy reduction. Nevertheless, the memory is also occupied by the data segment as well. Besides, the data fetch performance might be an important bottleneck of overall system performance. There are three general sections that could be recognized in a Data Segment (DS): Initialized data, Storage Reservation (or uninitialized data) and the dynamically allocated Heap. In the framework of this study the next logical step would be to reduce redundancy of these elements as well. If that could be done so that the performance of the data path from memory to the processor gets improved, overall performance of the system will be further increased.

3.1 Available Redundancy and Compression Strategy

The first question that needs to be answered is whether or not the compression of the data segment will result in any significant savings. An important consideration is the fact that the compression circumstances for the data segment are rather different than that for the code segment. Furthermore, different parts of the data segment require different approaches. Ultimately, a dynamic data *stream* as opposed to a static data *set* is compressed. The main difference here is in the static availability of data and the purpose of compression. If in the case of the code segment we needed to improve static characteristics of the code (the size), now we

need to optimize the dynamic qualities, so a different kind of information should be obtained.

Although several compression algorithms were considered (Arithmetic Coding [55],[46], Huffman [2], Lempel-Ziv [39]), a variation of the Huffman code compression algorithm was chosen (see discussion in Section 2.3). The Huffman method produces near optimal results for an integer number of code bits. It also allows reasonably fast decompression (either as FSM or via a lookup table) at a realistic real estate price [17],[18]. Nevertheless, the application of this algorithm should be different. If in the case of the code segment we had all the data to be compressed statically available to us, and the histogram was static, now, in the case of the data segment the story is different. When dealing with *Initialized data*, we still have somewhat fixed histogram and static availability of data (though the nature of data is different). But in case of the *Heap* (or dynamically allocated memory) there is no prior knowledge of what we are going to be dealing with. The *Uninitialized* storage in its original form could not be compressed at all because the programmer has just reserves an area of memory without providing any additional information on what kind of data will occupy it. One possible optimization for the uninitialized storage is static conversion to the dynamically allocated storage. This conversion might be a compiler optimization hidden from the user. In either way (whether we perform this optimization or not) the impact of the uninitialized storage on the overall data stream compression performance is minimal, as we will soon see.

Instead of applying the traditional adaptive Huffman algorithm [54], a *discrete regeneration* approach was chosen. This approach is based on the following assumptions:

- The frequency distribution is nearly constant during a short period of time (t) where t is bounded to the number of references to the cache;
- The next time slot ($t+1$) is likely to exhibit behavior similar to the current time

slot (t); and,

- By analyzing the data stream during time slot (t), it is possible to come out with near-optimal encoding for the time period ($t+1$).

If t is carefully chosen we will have an adaptive algorithm which has a *fixed* encoding for time period t. This fact is important for a caching structure because the compressed codes are statically stored within the time period t and is flexible enough to adjust to the changing environment while maintaining a near optimal quality of compression. In order to select an optimal t value, a set of experiments was conducted. Results are summarized in Table 2. A positive compression difference means a *decrease* in compression effectiveness.

Table 2 Regeneration Period selection

Number of references (t)	1,000	5,000	10,000	50,000	100,000
Average Compression Difference	+0.028%	+0.024%	0	+0.041%	+0.054%

Taking into account all the facts outlined above, we used three following adaptations of the Huffman encoding for use with the data segment. The first is called *Rigid Huffman* (RH). The histogram for this method is calculated once, at the compilation time, based on the initialized data segment and is not changed thereafter. This is the simplest and cheapest method, and should work fine for programs that do not use dynamically allocated memory much or who's dynamic histogram, is similar to the static one.

The second method is the *Flexible Huffman* (FH) (not to be mistaken with the Dynamic Hufmann or other dynamic compression algorithms [39]) method. This method's histogram is

being recalculated with certain period at execution time and a new encoding is generated. The profile data is being collected all the time and is discarded at the regeneration points. In this sense, the method remains static between regeneration points, which allows the storing of compressed data at these periods of time. This method is extensively opportunistic, and assumes that for the period of time $t+1$, the histogram will be similar to the period t but different from the histogram in the period t-1.

The third method is the *Flexible Huffman with Long Memory* (FHLM). It is similar to the previous method except the fact that its statistics keep on accumulating. This accumulation means that history is collected across multiple regeneration points, but is ignored only if some hardware limit is reached. A good example of one such event would be a counter overflow, in which case the counter just saturates. As suggested from the name, this method has a 'long memory' and should work best with highly unpredictable patterns: it will be more conservative in taking new opportunities, which might turn out to be either good or bad decision.

The last question to answer here is what granularity of compression to use. Based on the discussion in Section 2.3, the choices are unlimited, and as is commonly known, the bigger the atomic unit of compression, the better size reduction, and more complex is the decoder. In the case of the data segment compression, as we will soon see, decoder size is critical and some hardware structure is needed to collect the frequencies of all atomic elements. Sizes of both those hardware structures directly correlate to the size (and therefore the total number) of the atomic compression units. All these facts leads us to conclude that byte-level compression would be an optimal choice. This way the profiling hardware unit would consist of 256 counters and the decoder will be of a manageable size (more discussion is in the following section).

Now we should select the point where profile observation takes place. The natural

choice is the data bus between the processor and the main memory. What we going to see there is a stream of data going in both directions in response to the load and store instructions issued by the core processor. An important consideration for this introductory experiment is whether a data cache is present or not. It is not a trivial task to calculate the available redundancy, so first we monitor the system without the data cache. Below in Figures 14 through 21 the behavior of all the three compression methods is shown for each benchmark separately. The summary is in the Table 3 and Figure 22.

It is clearly can be seen that amount and dynamic performance of compression strongly depends on the benchmark and differs from region to region.

Table 3 Data Segment Compressibility Summary

	Rigid Huffman	Flexible Huffman	Flex/w Long Memory Huffman
Compress	0.90769	0.1569917	0.1570149
Go	0.55434	0.4612735	0.4478486
m88ksim	0.426866	0.32564377	0.33425884
Li	0.822175	0.6050848	0.581241
ijpeg	1.04028	0.8594622	0.809376
Vortex	0.741659	0.43141306	0.431698
Perl	0.720565	0.51699235	0.517772
Gcc	0.760478	0.55783421	0.577297
Average	0.735123	0.487659	0.482756

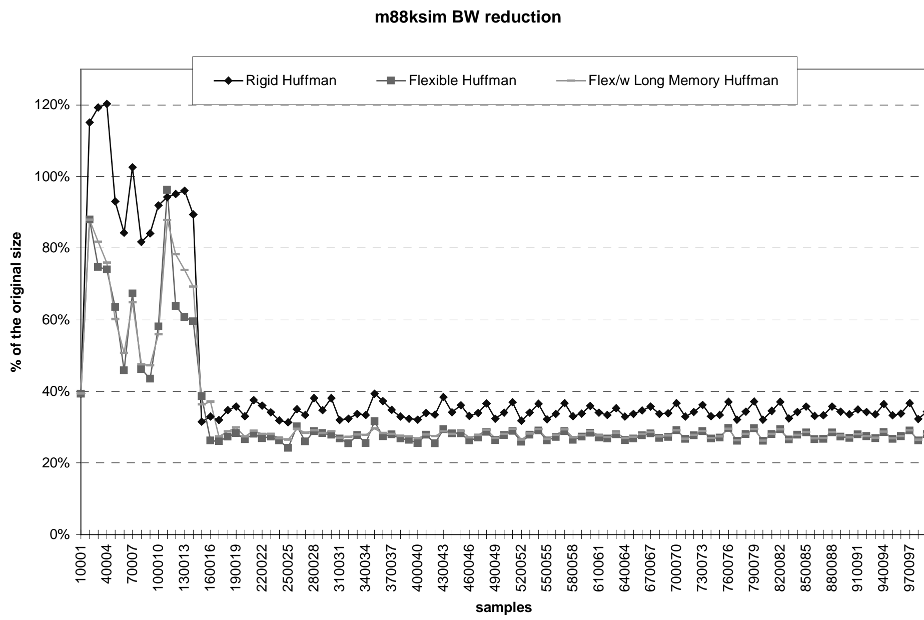


Figure 3.1 Dynamic Compression for M88ksim

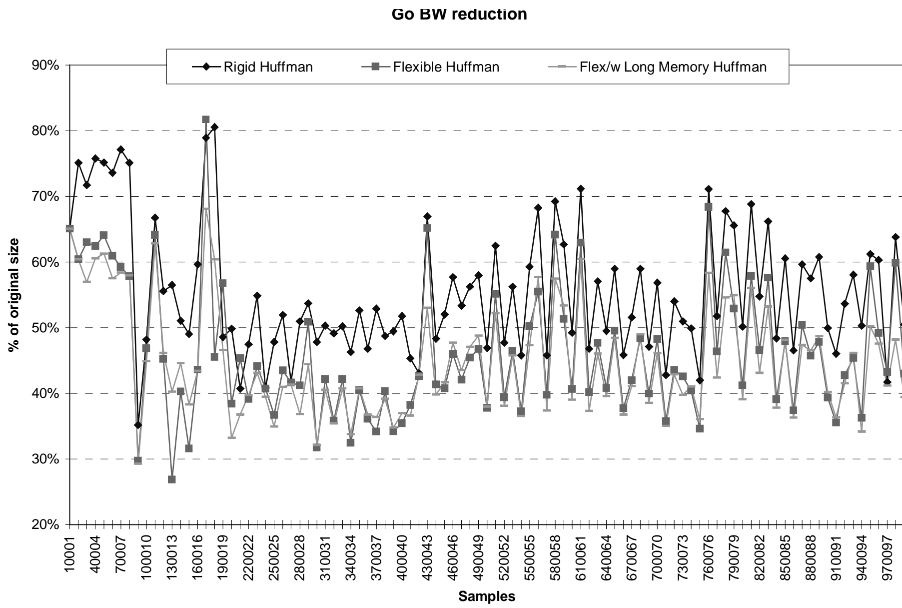


Figure 3.2 Dynamic Compression for Go

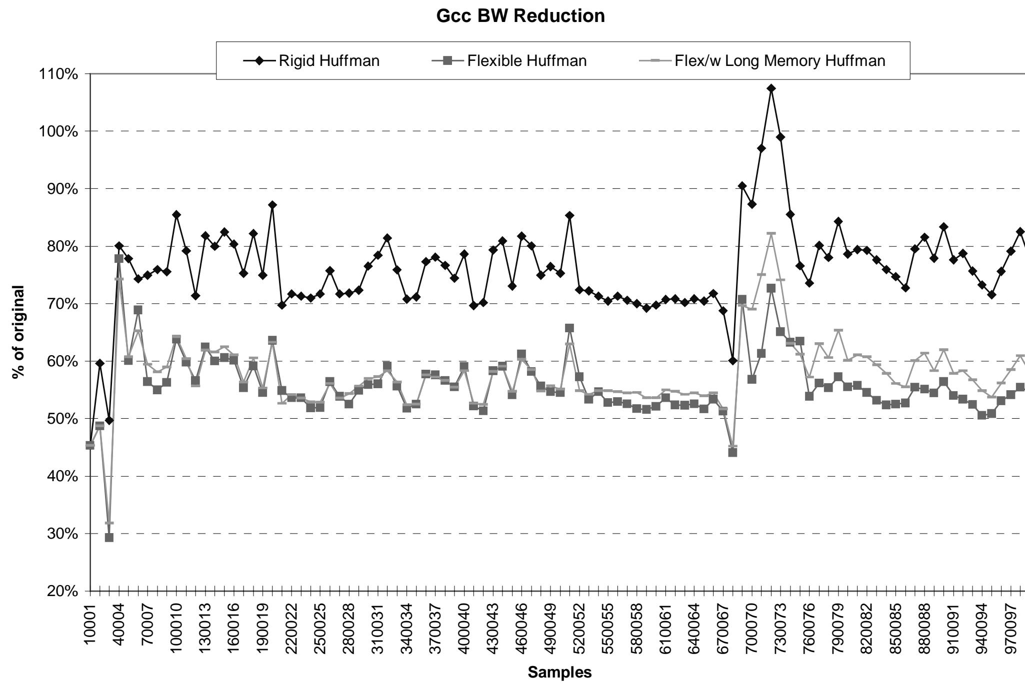


Figure 3.4 Dynamic Compression for Gcc

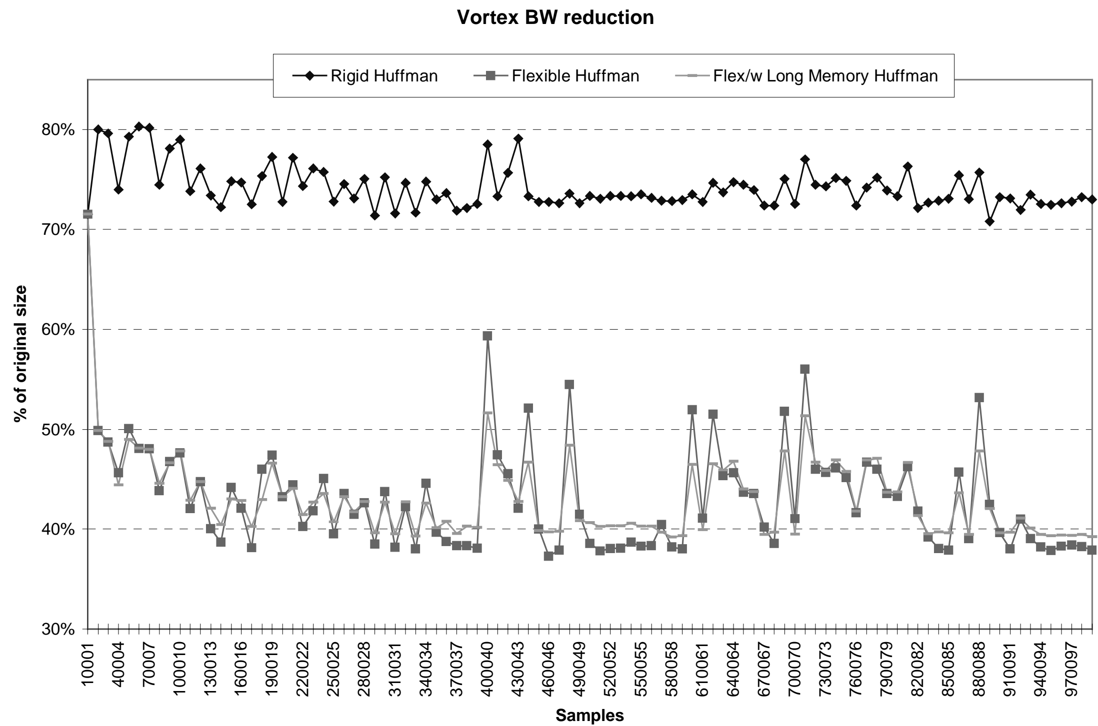


Figure 3.3 Dynamic Compression for Vortex

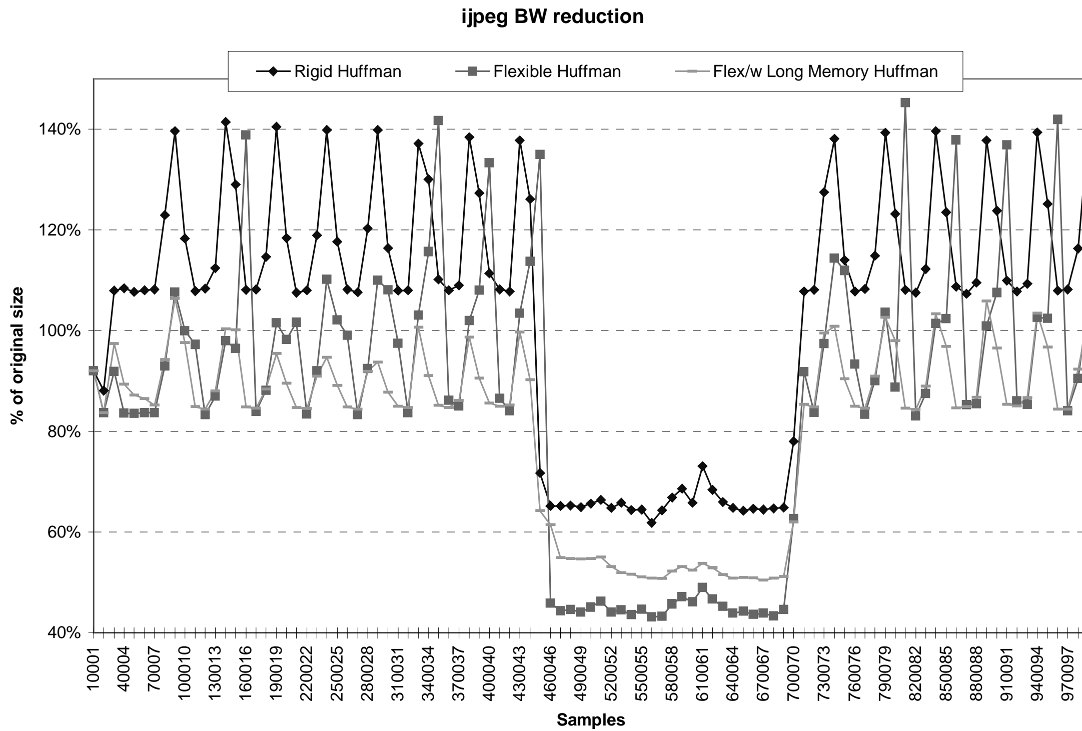


Figure 3.6 Dynamic Compression for Ijpeg

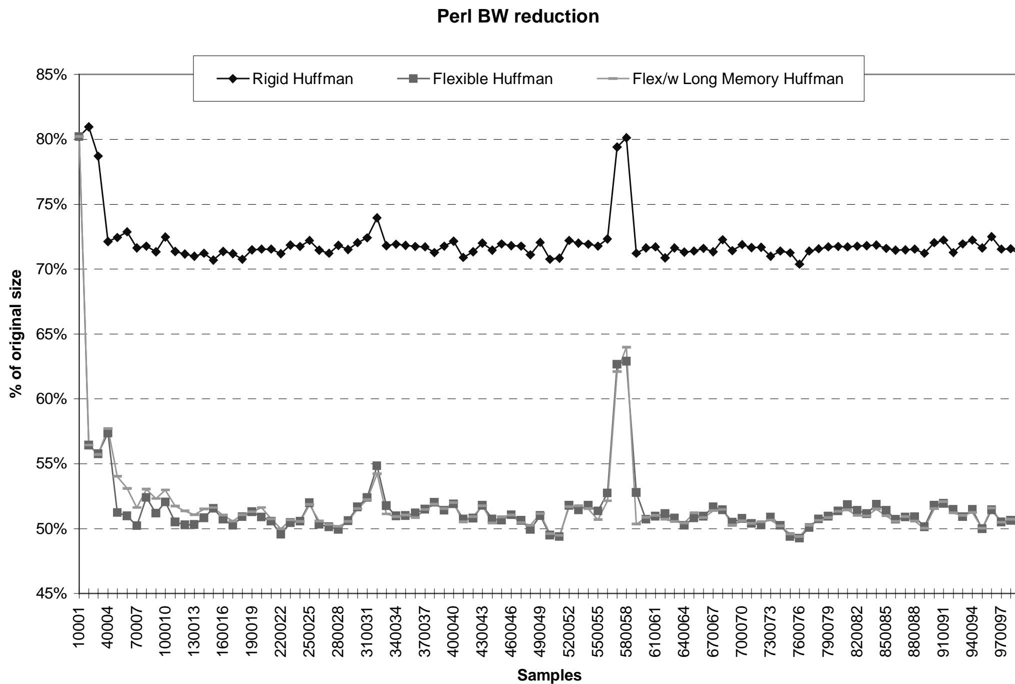


Figure 3.5 Dynamic Compression for Perl

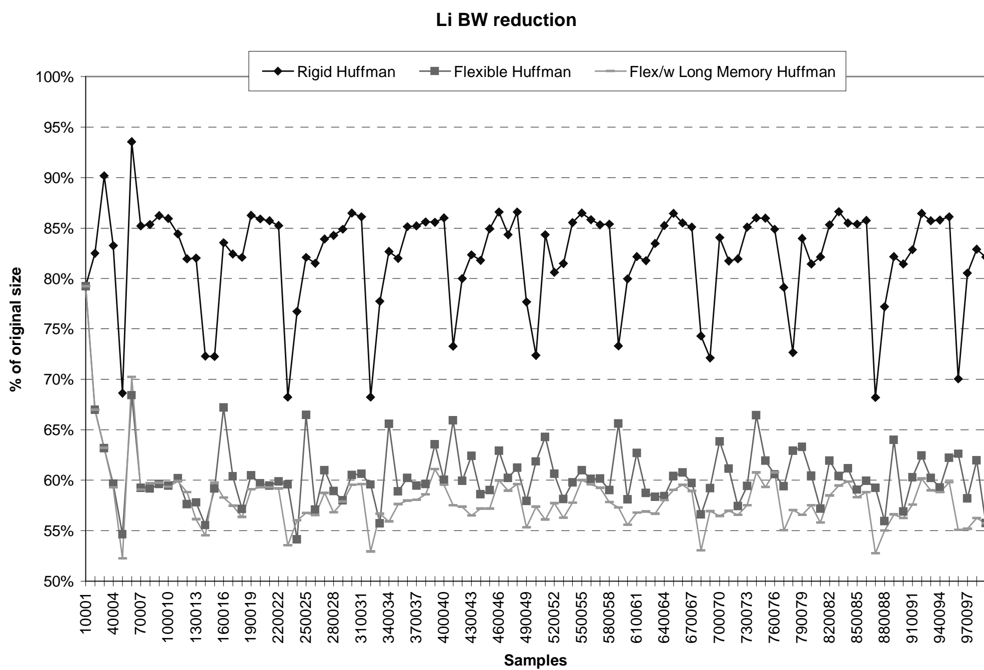


Figure 3.7 Dynamic Compression for Li

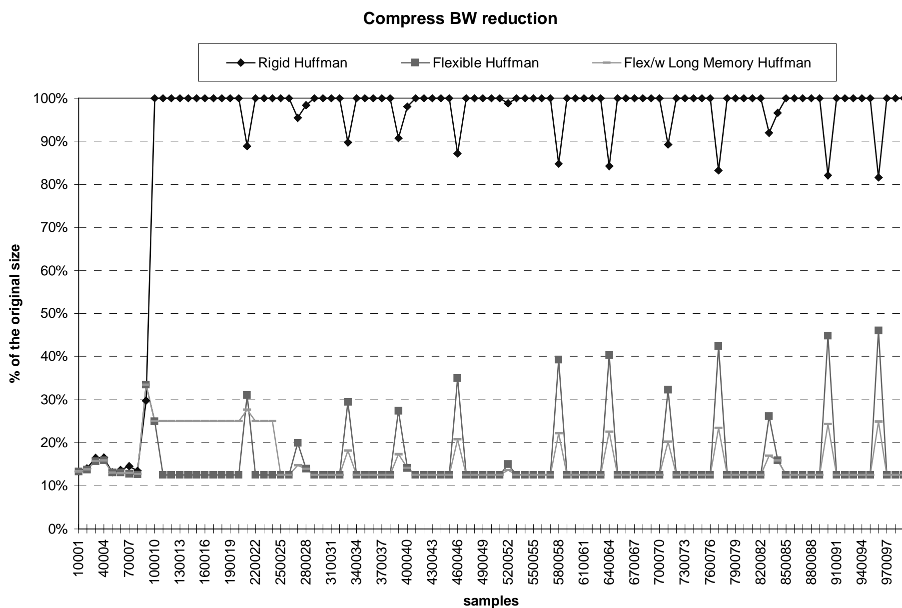


Figure 3.8 Dynamic Compression for Compress

The important detail for this experiment is that only original data request (regardless of granularity) have been served. Our assumptions for the performance of the three schemes turned out to be true. The Rigid Huffman performs well only at the initial stages of a program, when the initialized data is being accessed and then, when new values are generated, performs very

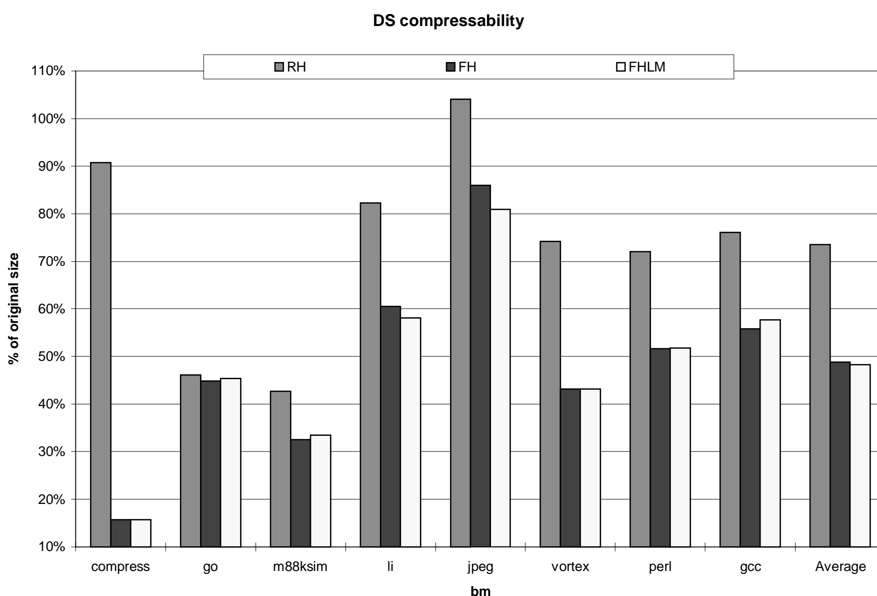


Figure 3.9 Summary of Data Segment Compressibility

poorly. It is very spectacular in the case of the *compress* benchmark. For *ijped* the compressed data size even exceeds the original data segment size. The reason for this increase is the fact that at initial region a precompressed (ijpeg coded) image is being loaded, so the data entropy is already high. Then, after a short period of time, where the image is being processed (and entropy of memory references is very low) it is stored back, causing new rise of entropy and fall of compression.

The Flexible Huffman takes every opportunity to adjust compression algorithm which may not always be the best choice, but it performs much better then the Rigid Huffman in

general. A good example of this behavior is the *go* benchmark. The performance of the Flexible Huffman with Long Memory is very similar to that of the Flexible Huffman and depends on benchmark, with overall results being slightly better (See Figure 3.9). Generally speaking, the Flexible Huffman is easier to implement in hardware than the Flexible Huffman with Long Memory since no care should be taken of the overflow support, which may prove to be the most practical approach. The summary of change in entropy of the stream before and after compression is presented in Figure 3.10. We can see that information density did increase,

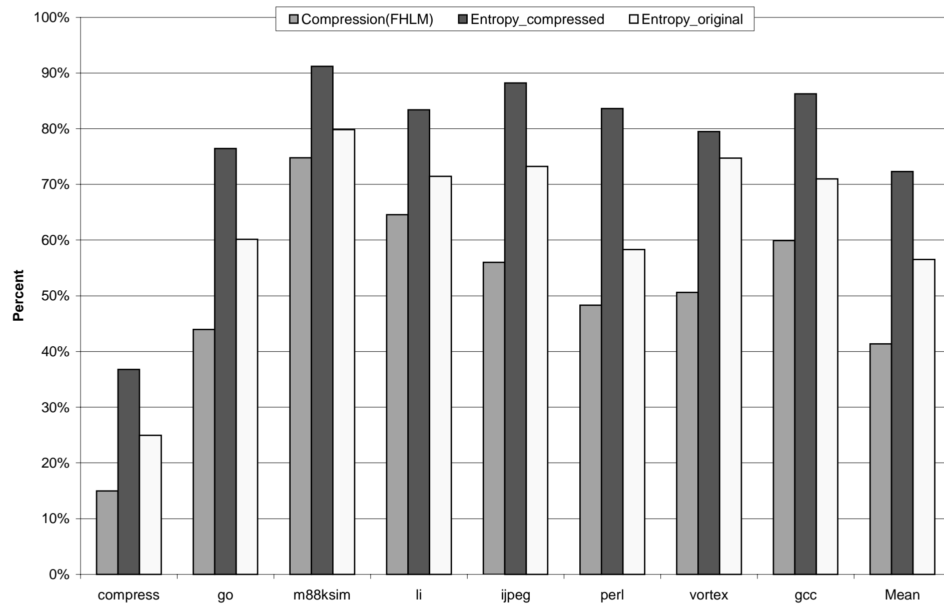


Figure 3.10 Entropy Change Due to Compression

but still remains substantially lower then the perfect measure. In the 4 we will propose a way to utilize this available slack to reduce power consumption of the data bus this code is being transferred on.

The final issue in this category is the selection of regeneration points. The importance of this issue becomes obvious when we recall that the contents of a caching structure must be purged every time the encoding changes. This will definitely hurt the overall performance and

could be considered as a catastrophic event. It is especially true if a software interrupt will be used to perform encoding regeneration. Generally speaking using an interrupt is the easiest approach to minimize the amount of hardware needed to support the flexible Huffman encoding. To prevent often regeneration from happening all we need to do is introduce a performance monitor. It could be easily implemented as a pair of counters with simple glue logic.

If the current compression is bellow certain threshold (for example 80% of original size have been used) we do not attempt to regenerate encoding and just keep on using the current

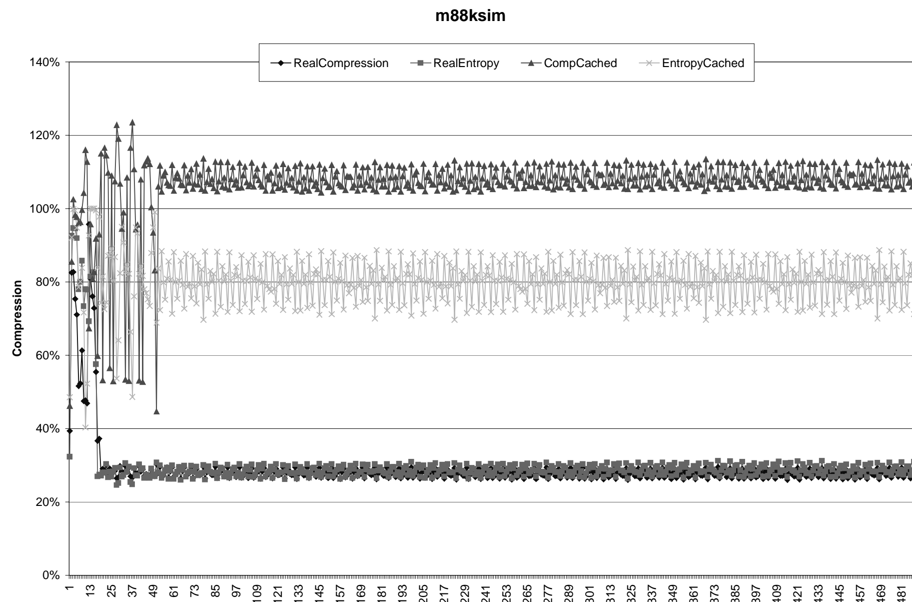


Figure 3.11 Dynamic Compression for M88ksim in presence of a Data cache

one. However, if the compressor is not doing a good job, overall cache performance will degrade any way and purging the storage will probably not hurt the performance any further. Table 4 demonstrates dependency between threshold value and performance degradation. 'Regeneration Frequency' refers to percentage of checkpoints, which actually caused algorithm regeneration. For the rest of the work 75% threshold is used.

Table 4 Compression degradation vs. threshold selection

Threshold	65%	75%	85%
Regeneration Frequency	35%	40%	70%
Compression Degradation	8-15%	< 3%	< 1%

3.2 Effects of the Data Cache on Data Compressibility

In the next set of experiments we increase the realism of the experimental setup by introducing a small data cache into the system. Now the author is attempting to investigate the change in behavior of the data stream between the data cache and the next level of memory hierarchy. Theoretically, we should see a significant difference in the compression algorithm performance. Instead of serving every load and store instruction issued by the processor, next level of memory hierarchy should respond to the stream of miss repair requests with a block of data. If there is no prefetch model is employed (like in our case, there is no explicit prefetch is done), the minimal block of data being transferred at a time is a cache block. The size of the cache block is fixed at four words (16 bytes).

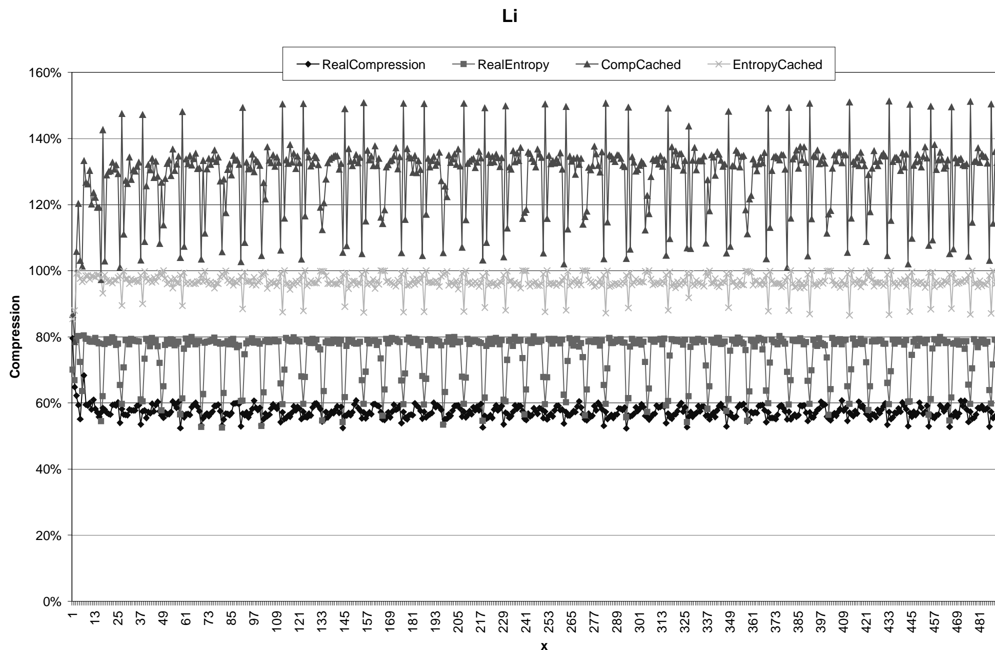


Figure 3.12 Dynamic Compression for Li in presence of a Data cache

Indeed, the data stream characteristics become nearly unrecognizable. Instead of the dynamic picture we have seen for the *m88ksim* and *li* previously (in Figure 3.1 and Figure 3.7) we go this new distribution (see Figure 3.11 and Figure 3.12), which is strongly inferior to the case without a cache. The perl's behavior also endured strong changes but remained manageable for redundancy-reducing algorithm (see Figure 3.13). The reader should note also an additional set of data in those figures labeled 'real compression' and 'real entropy'. In all the figures (Figure 3.11 through Figure 3.13) these 'real' label refers to the activity produced by a zero-sized cache ('real entropy' refers to the entropy of the data stream for this case).

This 'real' case is slightly different from the picture seen in Figure 3.1 and Figure 3.7 because this data stream is generated by the miss repair data requests that would be issued by a cache controller with zero storage array (which normally, as we just mentioned, requests one

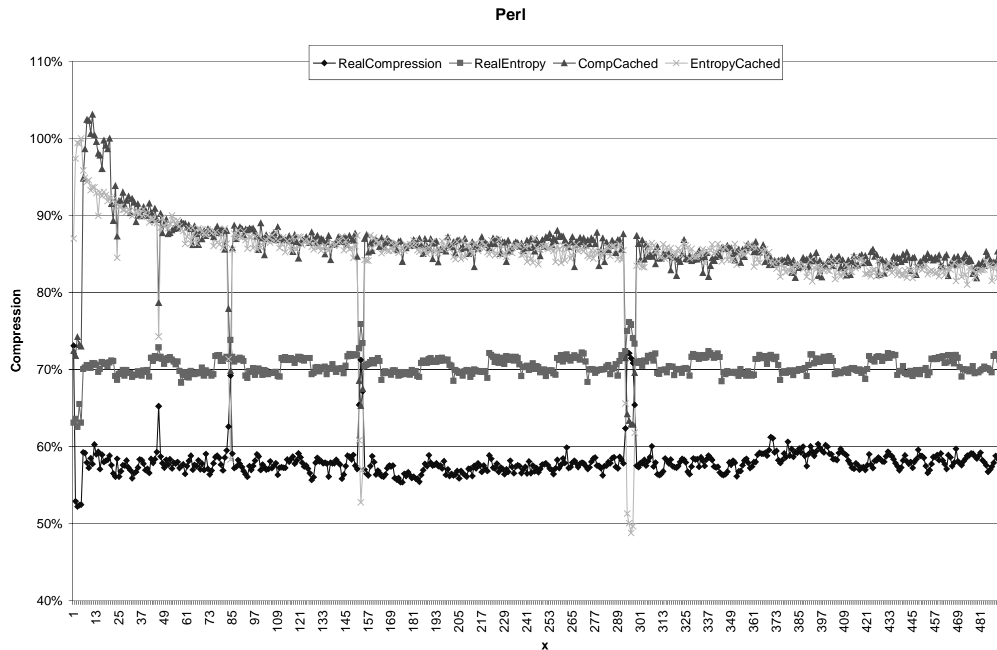


Figure 3.13 Dynamic Compression for Perl in presence of a Data cache

cache block at a time as oppose to a single word or byte at a time as assumed in the previous set of experiments).

The degradation in compression performance is quite understandable once we bear in mind that caching itself works by exploiting repetitiveness of references (time and spatial locality), which directly correlates to the stream redundancy. The data compression algorithm will be deprived of its primary resource – the redundancy. The data cache selected for this experiment is a small 4K direct mapped with Write Back, Write Allocate update policies. It seems that the Write Through and the No Write Allocate update policies would save the situation by allowing

enough traffic behind the cache, but it is not entirely true. After attempting to use the Write Through and No Write Allocate update policy it have been discovered that similar (or even lower) level of performance remains, while bus traffic increases unreasonably. The final results are summarized in Figure 3.14. The only compression scheme utilized in this experiment is the Flexible Huffman with Long Memory (as the best performer in the previous study). It is clear that the entropy of the data stream with the data cache present changes dramatically but still remains substantially large. We will investigate this matter further in Chapter 5.

Now that we see that a significant gain (of 50 % on average) could be achieved by

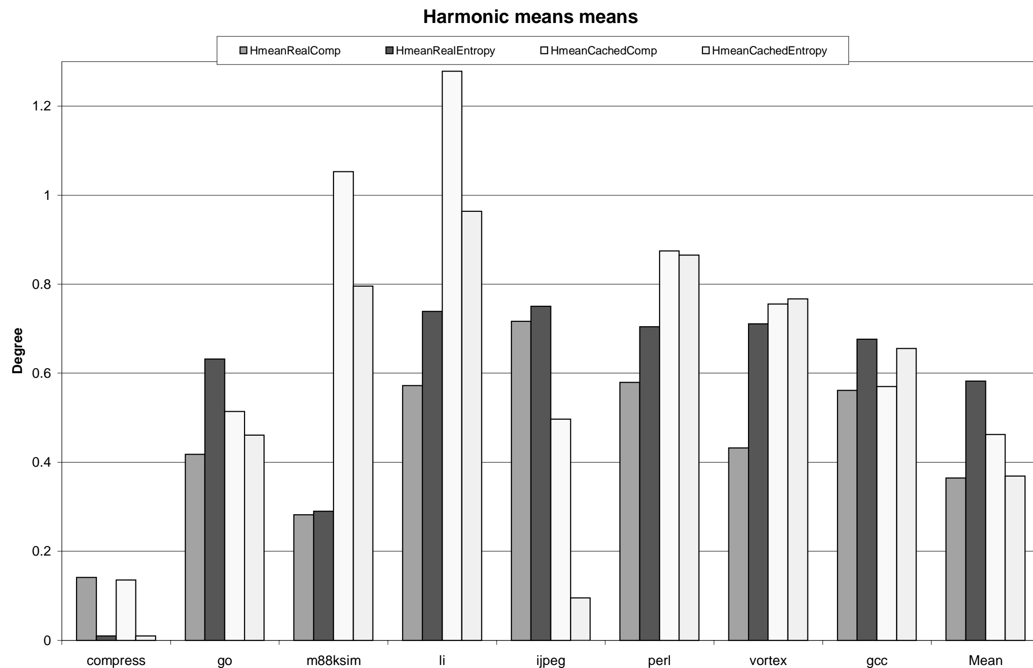


Figure 3.14 Effect of Data Cache on Data Stream compressibility

compressing the data segment let us find a way to utilize it. Unlike the compression of the code segment this technique is not limited to embedded system domain. The dynamic data compression could be used anywhere when space and bandwidth is an issue. It can reduce

memory bus traffic, increase data cache capacity (and hit ratio as a result) and reduce power consumption as a net effect. It can even be used in a multiprocessor environment to optimize inter-processor communications. The first area of application for the dynamic data stream compression is the system bus utilization and it is discussed in the next chapter.

4 System Data Bus Redundancy Utilization

4.1 Motivation and Experimental Setup

This chapter investigates the serious degradation from redundancy of the original data that can significantly benefit from exploration of the available redundancy: the system data bus [33],[34],[49]. The code segment is not the only portion of the program that contains high level of redundancy. In addition, the data segment suffers from excessive redundancy as well. In some instances the data stream, consisting of data produced and consumed by the program, is

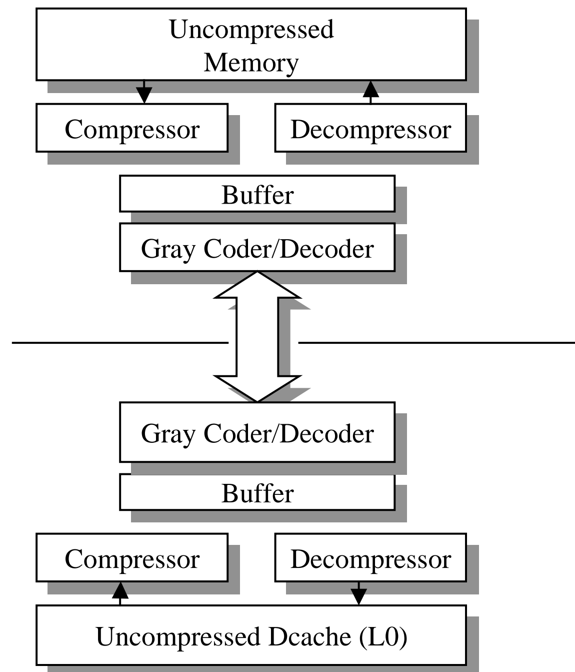


Figure 4.1 Traditional. Bus Encoding Experimental Setup

even worse. Moreover, the quality of the data stream strongly correlates to the region of the

program and is time variant. The instruction bus is viewed as a conductor for a dynamic stream of data with a random, nearly uniformly distributed sequence of values with a high level of redundancy. The bus is optimized for both shorter transaction cycle and lower switching activity without sacrificing the overall throughoutput. The reduction of the switching activity directly contributes to reliability (data integrity) of the bus and low power design. It is well known that Input/Output (I/O) circuits are one of the major power consumers in a system [33],[34],[35]. Their share of power dissipation could easily reach 30-40% and some times even can exceed 50 % of overall power consumption. When transactions on I/O bus compared with internal transactions, the former dissipate 100-1000 times more power [33]. This is happening due to their large capacitance (three orders of magnitude) when compared to internal circuits [33],[34],[49].

This problem is further intensified by the fact that due to the low information contents of

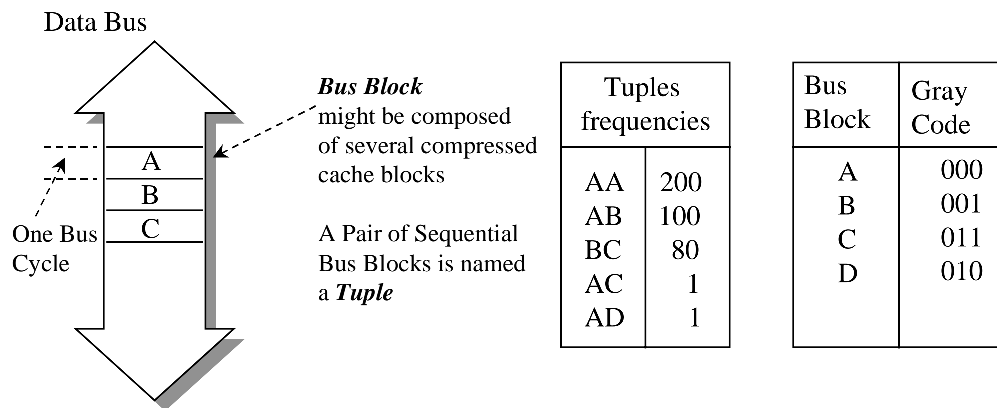


Figure 4.2 Bus Blocks and Tuples Structure

data being transferred through I/O subsystem, it ending up being used more intensive than really necessary. For instance, from the initial experiments it has been found that at some regions of the SPECInt95 programs [45] the system data bus transfers solid blocks of *zeroes* eighty percent

of the time.

Following the integral approach to the embedded system improvement in general, and the front end of it in particular, it is necessary to pay close attention to the I/O subsystem. By creative involvement of compiler and run time collected information, this situation can be improved on. It has been found that by removing redundancy from the data sent over the system

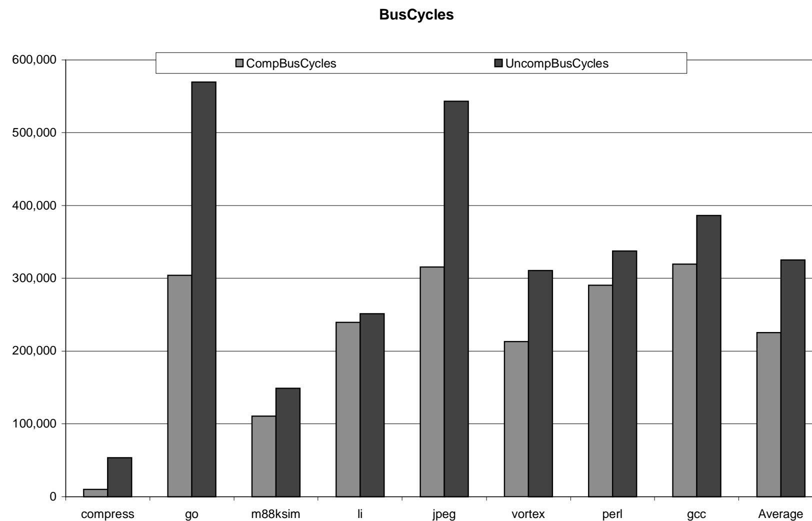


Figure 4.3 Busy Bus Cycles

bus the throughoutput of the bus was doubled. Nevertheless, if low power is the primal goal of this optimization, some additional coding is needed. By superimposing the Gray code [30] on the top of compression the ultimate goal of higher throughoutput with lower switching activity can be achieved.

Although power consumption is very hard to estimate statically, it is safe to assume that majority of the power is consumed when a bit flips (changes its value) on the I/O pad and the correspondent bus line. This fact means that power consumption for I/O circuits is in direct correlation with the information contents of the data being sent through the bus. So in the spirit

of the previous discussion, let us investigate the reduction of switching activity and the increase of utilization of the data bus by the reduction of redundancy in the data stream going through it.

The first set of experiments were conducted in an idealized environment where there were no additional structures other than the core processor, a very small L0 data cache, and the off-chip memory (see Figure 4.1). By 'small cache' (one cache line) we understand here some kind of a read buffer or a memory controller, and the only purpose of it in the current experimental setup is to model a real cache data request activity. As have been discussed in the previous chapter, the difference between no-cache and a tiny cache is how main memory is accessed on a data request. If the processor attempts to load a byte the memory is normally accessed for at least a cache block (given that the reference missed in the cache). If prefetching

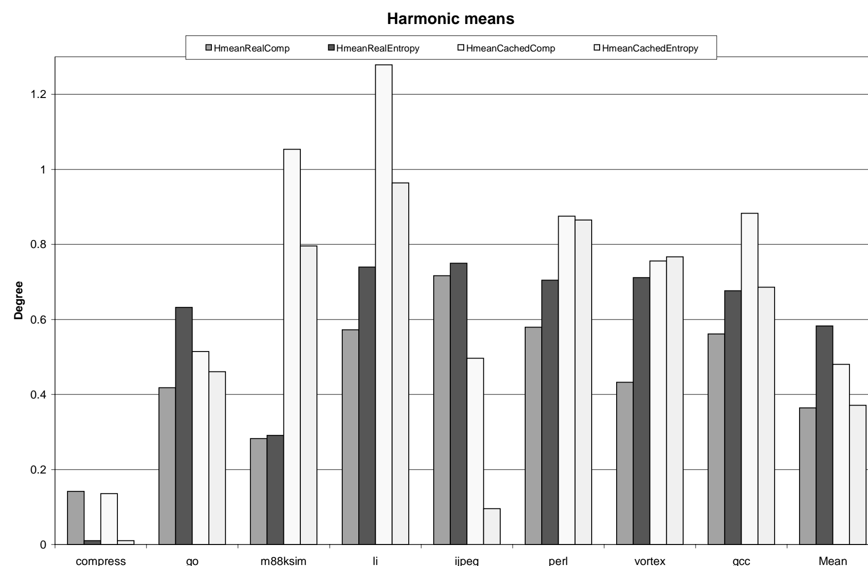


Figure 4.4 Entropy Changes due to Caching

is used multiple cache blocks could be delivered on a single miss. With the cache block set for 16-bytes, we can realistically model memory bus traffic without having the effect of a large

cache use. Neither the processor nor the rest of the memory hierarchy sustains any changes. Bus operations are completely transparent for them. The bus itself is modeled as a queue (FIFO structure) through which a sequence of *bus blocks* is transferred. It should be noted that those bus blocks might be unrelated to either of the logical blocks commonly assumed in the memory interface. In other words, a single bus block might include either multiple or fractional parts of a cache block mixed in random order (see Figure 4.2).

From now on for all the bus compression purposes we need only consider bus blocks.

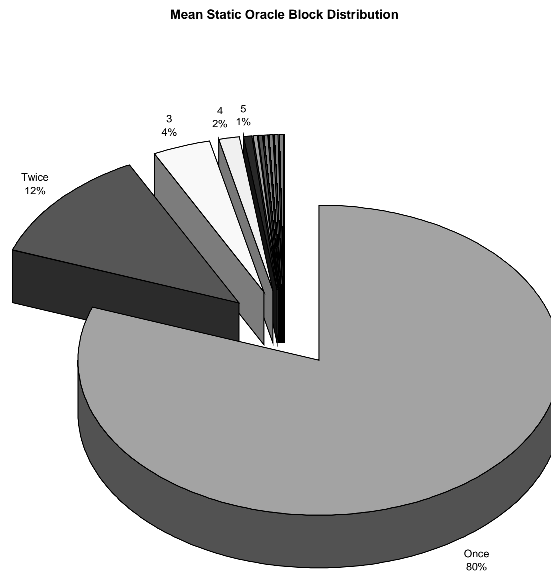


Figure 4.5 Oracle Block Distribution

The first set of experiments illuminates the effect of compression. We are trying to reduce the number of cycles that the bus remains busy. The compression algorithm used is the Flexible Huffman with Long Memory. Figure 4.3 summarizes this information. We can clearly see that certain benchmarks can significantly increase their bus utilization in time (throughoutput). For

example *go*, *ijpeg* and *compress* nearly double it. But there is one catch: with the reduction in the amount of time the data is being transferred reduced, but the amount of *information* remaining unchanged, resulting entropy increases and the switching activity multiplies. This effect could be illustrated in the following way (see Figure 4.6). In this figure two abstract distributions corresponding to two different transfers on bus are presented. One (S2) is presenting the transaction of data in the original form and derived from figures (Figure 3.1 through Figure 3.8). The other one (S1) is representing switching activity for compressed data segment being sent through the bus. The important quality is that shaded areas underneath the curves (S1 and S2) are of equal in size. The measure of this activity increase is experimentally confirmed and summarized in Figure 4.3 and Figure 4.4. Apparently sole compression of the data bus only leads to the time savings, but defeats the purpose of low power design. Instead of an even distribution of switching activity over a longer period of time, we will have a short burst of activity, increasing the peak power consumption (which ultimately could require a larger power supply).

This fact leads us to the conclusion that some additional encoding is needed if we still want to use bus compression. If compressed bus blocks are considered to be atomic units of transfer, we can attempt to establish a correlation between their *sequences* (order) and the switching activity. From the same Figure 4.4, we can also see that there is still a certain amount of redundancy available, even after compression has been performed. This leaves us some room for improvement. A natural choice to reduce switching between sequential states is to apply the Gray coding [30] to the compressed blocks. We should also mention that Gray coding imposes minimum amount of delay for both encoding and decoding. With this double encoding each cache block appears to be compressed in two *dimensions*: in space and in time with a net result

of higher entropy and low switching activity.

4.2 Data Bus Coding Algorithms

The Gray coding algorithm itself should be modified so that it may adapt to the constantly changing bus activity. The basic idea is simple: to make most common pairs of compressed bus blocks differ in a minimum number of bits (perfectly just one bit). Let us call a pair of compressed bus blocks a *tuple*. If the two sequential blocks are different, this is a real tuple, if

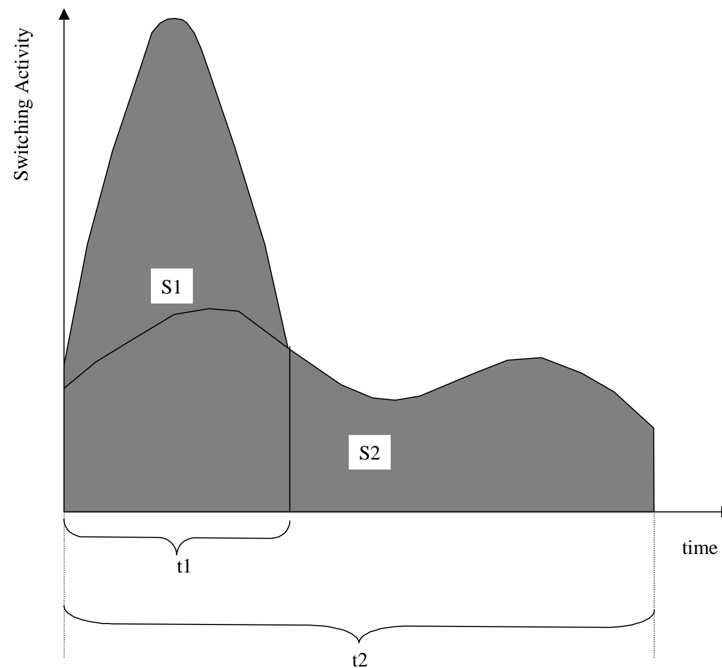


Figure 4.6 Density of the Switching Activity on Compressed Data Bus

they are the same, this is an empty tuple (see Figure 4.2). From now on we only interested in optimizing real tuples. There were several dynamic modifications proposed - Oracle Gray, Adaptive Age, Adaptive Infinite Tuples and Adaptive Limited Tuples.

The Oracle Gray is an idealized structure (not necessarily hardware implementable), which has full knowledge of all compressed bus block frequencies prior to execution. The list of all possible blocks is sorted in descending probability order and Gray codes are assigned to this list in the same order. This guarantees that the most frequent blocks will have the least bit 'distance' (differ in the least number of bits). Practical implementation of the Oracle Gray is

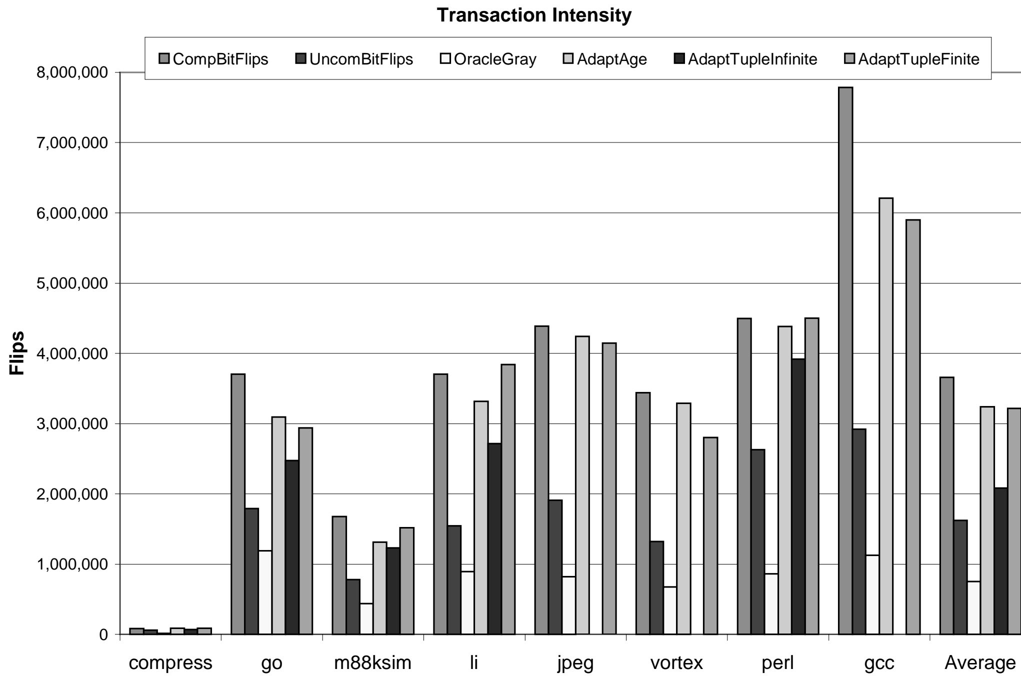


Figure 4.7 Transaction Intensity

only possible if accurate profiling of all possible data sets can be performed. In this case, we would know the probability of any common block to appear on the bus. Theoretically, there might be 2^{Bus_width} blocks, but as have been discovered, only about 5-10% of them are commonly seen and the total span is only 50-60% of the maximum number. Basically, it means that if we simply *enumerate* all possible blocks we should already see a large drop in switching activity. Furthermore, greater reduction is expected with the Gray encoding. The major

advantage of this method is the fact that both the encoder and decoder have a full table of all common blocks *prior* to execution. For an adaptive scheme, it means that original blocks should never been transferred in unencoded form. Figure 4.5 represents an example of blocks distribution. It gives an example of how many blocks have been seen once, twice and so on (this is the arithmetic mean across all benchmarks).

The next method, the Adaptive Age, attempts to model the Oracle encoding in a real time system with no prior profiling. After a compressed block is first seen (and transferred in its

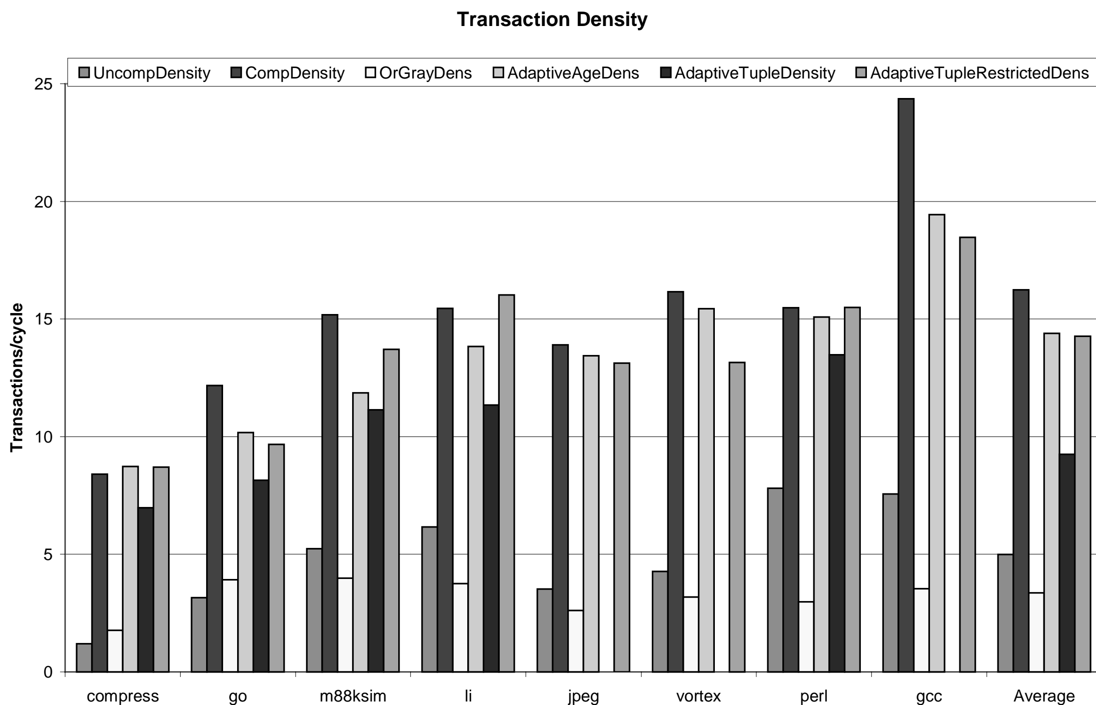


Figure 4.8 Transaction Density

original form) both the encoder and decoder create an entry in their table. The next time the block is seen, the code is sent instead of the original form. Every time the block is seen again, its frequency is increased, and at a certain period of time the Gray codes are recalculated with

these updated frequencies in mind. The recalculation point could be predefined (for instance every million transfers) or flexible. If flexible regeneration points are used, some monitoring hardware might be dedicated to detect the need for code regeneration. Additional overhead for such a system is an extra physical line needed to be added to the bus. This line indicates whether a Gray coded or an original block is being transferred. With multiple service lines already used on buses, some of the existing connections can be utilized.

The next two Gray encoders are trying to utilize frequencies of *tuples* being transferred as opposed to previously used frequencies of individual blocks (see Figure 4.2). The general idea is, instead of looking at individual blocks and their frequencies, to collect statistics on pairs of blocks. This method allows us to take away uncertainty on what block is being followed by what, but requires much more information to be kept around. The first method is the Adaptive Infinite tuples algorithm. It method is theoretical and assumes infinite storage for all possible tuples frequencies. This algorithm is only used here to estimate available performance gain. The second – Limited Adaptive Tuple – limits tuples frequency storage to a finite number: 1024 entries. Experimental results are summarized in Figure 4.7 and Figure 4.8. While the Figure 4.7 gives the absolute number of flips the Figure 4.8 normalizes them to the shortened (due to initial compression) period of time: compressed bus cycles. In either case, the Oracle Gray achieves a significant reduction of bus activity on top of compression. It even does better, on average, than the original uncompressed data stream. This fact basically means that if the Oracle Gray code conditions can be practically achieved (implemented in hardware with aid from the compiler), then the ultimate reward of shorter bus busy state time and lower power consumption can be achieved. On the other hand, all the ‘practical’ implementations suffer from the fact that each block should be sent through the bus at least once to be encoded in the future. Nevertheless all

of them do reduce the bit flip activity overhead from compressing the original data stream.

The final step in this study would be changing the overall conditions to something more realistic, such as increasing the L0 data cache size to 32KB. This is the memory interface setup most of existing systems are utilizing. As we described in Section 3.2, the problem is that both the data cache and the bus encoding mechanism (in this case) are utilizing the same basic quality of the data stream (redundancy/entropy). Since the cache is closer to the original source of the redundancy, the processor, it gets the best of it. The cache virtually prevents most common blocks from appearing on the bus. (Refer back to the discussion in section 3.2). Once the data stream is 'filtered' through the data cache, its entropy increases and the bus encoder has nothing to work with: all it sees are unique blocks that a rarely repeating. This conclusion conforms to our previous findings regarding the entropy of data filtered by a small cache (see Section 3.2). This observation might be considered a negative result in general case. But for some embedded systems, which do not use caching at all, or have very small caches, it might prove to be useful.

5 Compressed Data Cache Hardware Implementation

5.1 Motivation

This chapter deals with another key component of an embedded system, which can be significantly improved by reducing the redundancy of original data stream: the data cache.

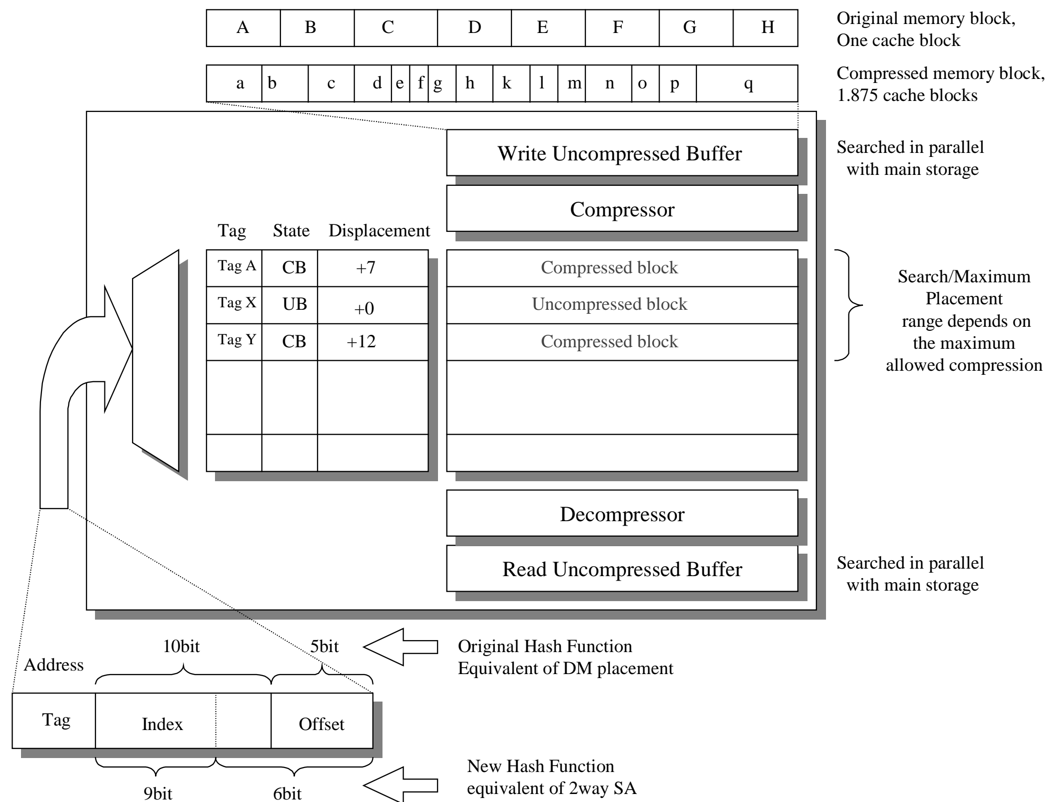


Figure 5.1 Compressed Data Cache Architecture

There has been little work in hardware design schemes for exploiting data value compressibility. This is understandable, since on-die data memory capacity today is not a performance-limiting

factor for general-purpose processors. However, this is not the case for special-purpose or embedded processors that often share the die with the rest of the system. In several recently announced embedded systems like the MIPS64 20Kc [52] and the IBM PowerPC 750x [53], on chip instruction and data caches occupies approximately 50% of the silicon area and consume a significant share of total power. In these situations, there is a need for very highly efficient use of data memory, or other alternatives to use smaller hardware structures while delivering a similar level of performance.

Moreover, since we found that the data cache is the primary consumer of the available redundancy of the data stream, the next logical step is to attempt to increase its performance the same way we did for the instruction cache – by compressing it. As have been mentioned before, though similar at the first glance, the actual coding conditions for code set and data stream are drastically different. The major difference is that instead of a *static* code segment, we are dealing with a constantly changing *dynamic* data stream, so we cannot apply static compression algorithm (with fixed frequency distribution). In addition to that, since the compressed data should be stored (statically) for some period of time prior to decompression, we cannot apply a truly adaptive compression algorithm either. With all this in mind, we have a new and unique set of contradictory problems to solve.

5.2 Compressed Data Cache Architecture

Following the detailed analysis of data stream compressibility in section 3.1, we can affirm that the compression algorithm that best fits this set of coding conditions is the Discrete Adaptive Huffman. Let us now define several basic assumptions for design of the proposed

compressed data cache:

- The cache is direct mapped in essence, but provides an *implicit associativity* (see explanation below);
- Both compressed and uncompressed data blocks can be stored in the cache at the same time;
- The smallest compressible block is a cache line; (no partial compression for fractions of a cache line);
- A block of data is stored in compressed form only if compression reduces its size;
- Hashing is a function of compressibility of the datum (implicit associativity); and,
- The cache uses write allocate/ write back update policies.

Let us begin by describing the implicit associativity mechanism. One of the basic

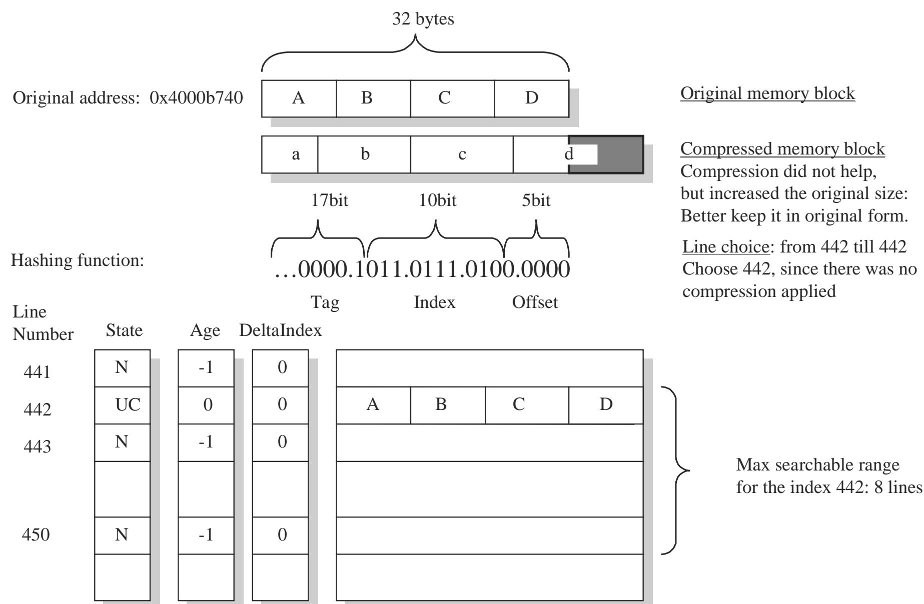


Figure 5.2 Block Placement Example - Expanded Block Placement

features of this cache is the variable hashing function. The whole design is built around of the

idea of variable compressibility of the available data, and the hashing function reflects it. There are three possible outcomes for an attempt to compress a block of data: its size is reduced, remains unchanged or is increased. Obviously the outcome depends on frequency of elements that make up that block. If we want to store compressed blocks of a random size, we need to provide a flexible mechanism for it. But the point is that we want to preserve precious storage within the data cache, so *only* blocks that *reduce* its size after compression are stored and placement policy is modified accordingly. In other words the more compressible a cache block is the more flexibility on its placement is allowed. This mechanism is best explained with an example.

In the Figure 5.2 we see an example of the case where compression attempt produced negative results - the cache block was increased in size after compression. This happens when a cache block contains rarely used bytes. Fortunately, those blocks are infrequently encountered as well. Nevertheless, in this case the block is stored in the compressed cache in its original

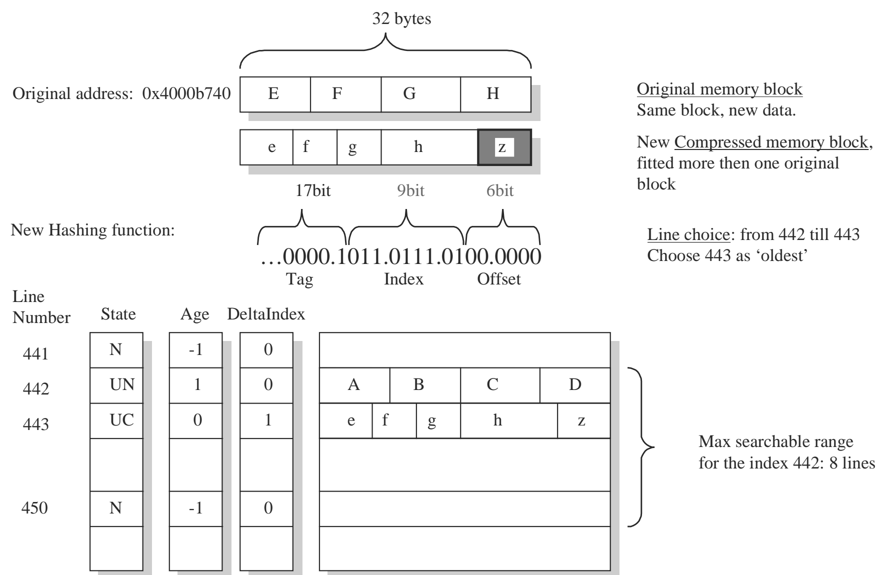


Figure 5.3 Block Placement Example - Reduced Block Placement

form (uncompressed), and the hashing function is equivalent to the one for the direct mapped cache (see Figure 5.2). A different approach is taken for blocks that are reduced in size after

compression (see Figure 5.3).

Before we proceed with this example, let us first define granularity level for the cache. The granularity level directly correlates to the smallest addressable unit in the cache, which normally is word or byte. Let us use a single byte as atomic unit in our case. Depending on the granularity level, when a single additional atomic unit is added to the compressed block (in Figure 5.3 the original size of *z* is multiple of a byte), the compressed block offset automatically 'increased' by one bit (in this example to six from five), and its index is 'reduced' by one bit (to

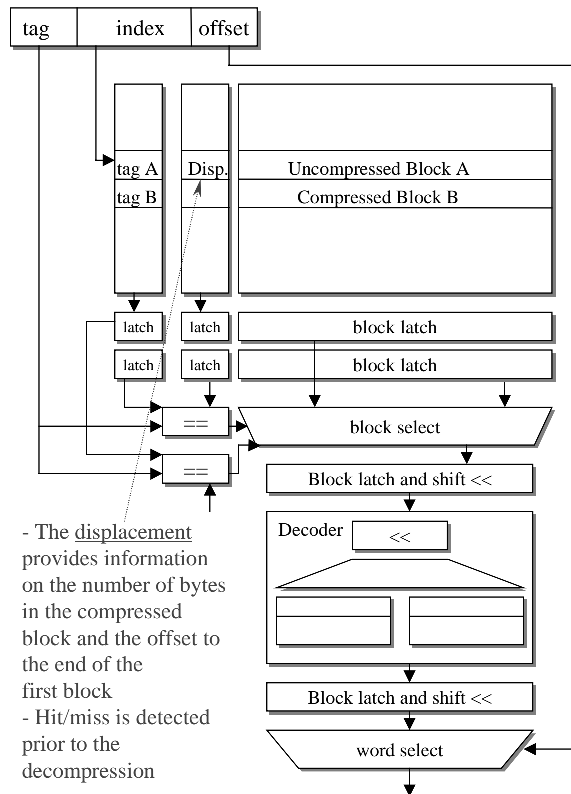


Figure 5.4 Read Pipeline. Multiple Set storage

nine from ten). This shorter offset in turn implicitly doubles the line selection choice – now this compressed block could be placed in either line 442 or the line 443 in main cache storage. This process is repeated until a compressed block's size equals or exceeds the original block size. By

this time, the compressed block might contain multiple cache blocks (up to the maximum available number allowed by the compression algorithm – in this example eight). Among the possible line candidates, the least recently used (LRU) is chosen. This process is named the *implicit associativity*.

As it has been mentioned before, for the byte-base compression algorithm, which is used in this set of experiments, the granularity is a single byte, so the best potential compression

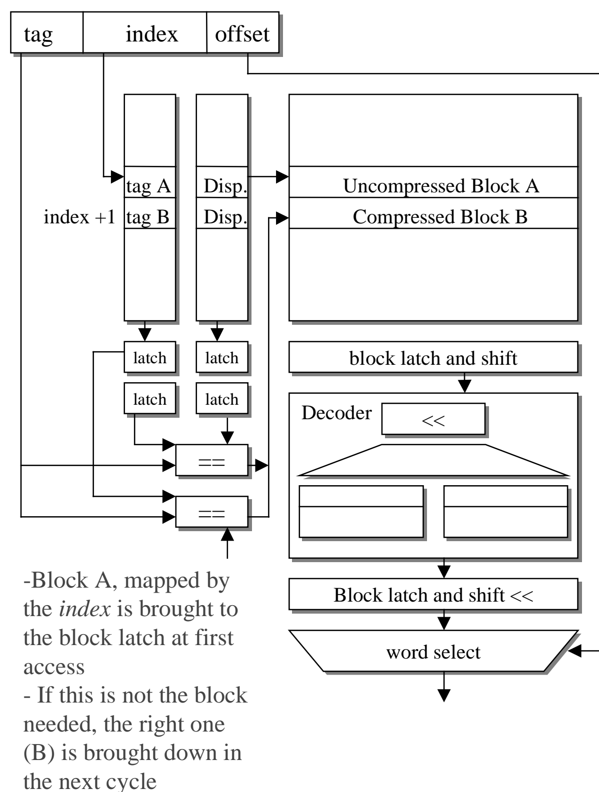


Figure 5.5 Read Pipeline. Two cycle access

occurs when a whole byte is represented with a single bit, i.e. by eight fold (8x). Given this information, up to eight cache blocks can be present in a single compressed cache line. This in turn means that a line can be placed anywhere within this eight line window with original (direct

mapping) address inside of it. Therefore, in order to find a single block, we need to search all the entries for all possible locations of the compressed cache block. In current case this would be equivalent to an eight-way set associative cache. If this search range ever appears to be a limiting factor for hardware implementation of the algorithm, the maximum compressibility could be limited, with corresponding degradation of performance (from 0.023 average miss ratio for the 8x unbounded compression down to the 0.031 for 2x restricted compression).

Figure 5.4 shows the example of a 2x limited compression and also presents a potential

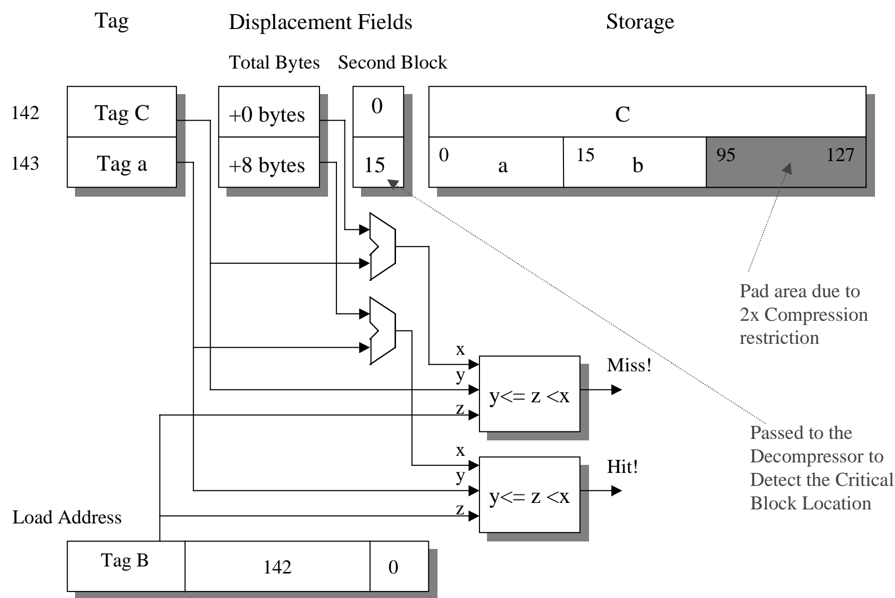


Figure 5.6 2x Restricted Compression Block Placement and Access

implementation of the cache read pipeline. If the read mechanism used in conventional two-way set associative cache is implemented, both cache storage and tag/displacement array can either be banked (the traditional implementation with hardware duplication) or dual ported. It is well known that multiporting of more then two presents an extreme design challenge and calls for custom design and layout. Because of this, 8x compression might have impractical hardware requirements for this scheme.

There are two tradeoffs that can be exploited in order to implement implicit associativity for multi-way compression, depending on target implementation: (1) if the access latency is

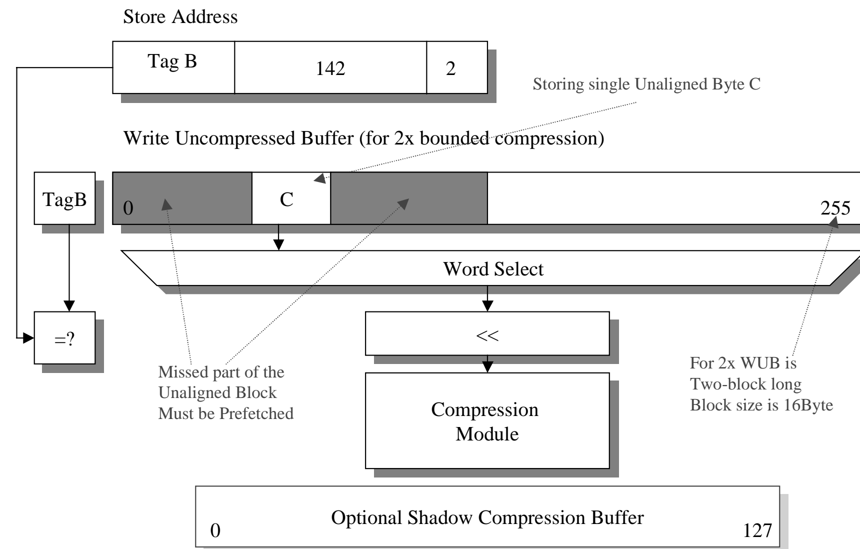


Figure 5.7 WUB Organization

critical, the scheme presented in Figure 5.4 can be used, or (2) if an extra cycle for *some* (not all) accesses can be tolerated, the scheme presented in Figure 5.5 can be used instead. In this scheme, access to an uncompressed block is done in one cycle, and access to a compressed block takes one or two cycles plus time for decompression. In either case, once a block is found and it is uncompressed, it could be delivered in the same way as a traditional cache does (see discussion below). If the block is compressed, it must be uncompressed first into the read buffer (uncompressed blocks are not placed there, see below). The critical block can be decompressed first using some additional information (6-bit displacement) which is stored along with the block tag in the displacement field (see Figure 5.6, the 'second block' field). If the next load hits in the same line or one of the following blocks, it is likely to be already uncompressed and resident in the read buffer. To improve access time, the read and write buffers are searched in parallel

with main cache storage. In this sense this scheme is similar to the open page policy in DRAMs.

Now the read and write sequences should be described in greater details. On a write to the cache (the processor stores a datum) the data does not goes directly into the compressed storage. It is first written into the Write Uncompressed Buffer (WUB) (see Figure 5.1 and

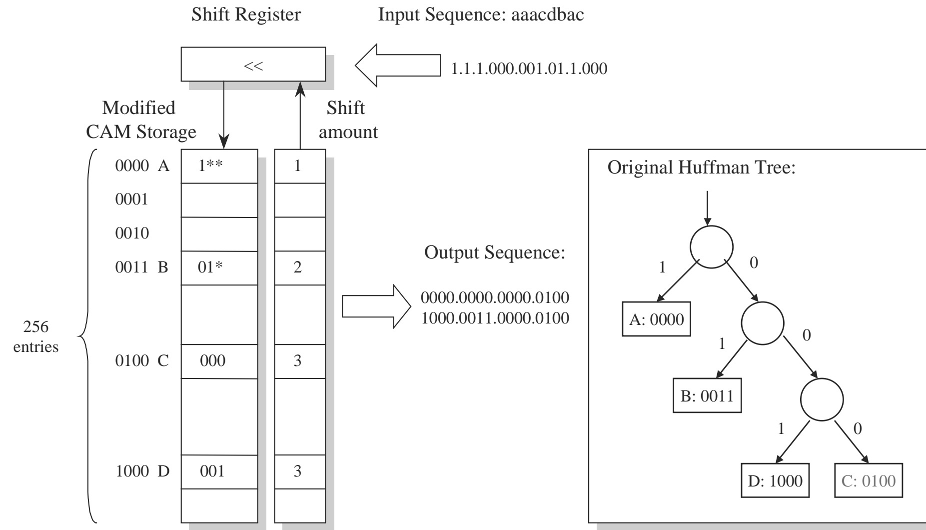


Figure 5.8 Logical Structure of the Reprogrammable Huffman Decoder

Figure 5.7). The WUB can hold up to eight (or whatever the compression limit is) sequential cache blocks. If the compression latency poses a delay on the write procedure, each block written into WUB can be speculatively pre-compressed and stored in a shadow compressed buffer. If the datum being stored by the processor is not aligned at the beginning position of a cache block, the missing portion of the block is prefetched (see Figure 5.7). The next write to the cache is checked against the block currently located in the WUB. If this datum is from the same block (which often is the case), it is accumulated in the WUB. Potentially up to eight cache blocks can be accumulated in the WUB. It is important to notice that there is no limitation on order and permutation of reads and writes since we have two independent

hardware paths in the cache to handle them.

Once the maximum capacity of the WUB is reached, or there was a datum from a different block encountered (miss in the WUB), the whole contents of the WUB is compressed, one block at a time. For each individual block the decision is made whether it is stored in compressed or uncompressed form. The only restriction for this process is the sequentiality. If for example, out of the eight sequential blocks in the WUB, the first two get compressed and the following two do not, while the remain four do, we will have three independent groups to be

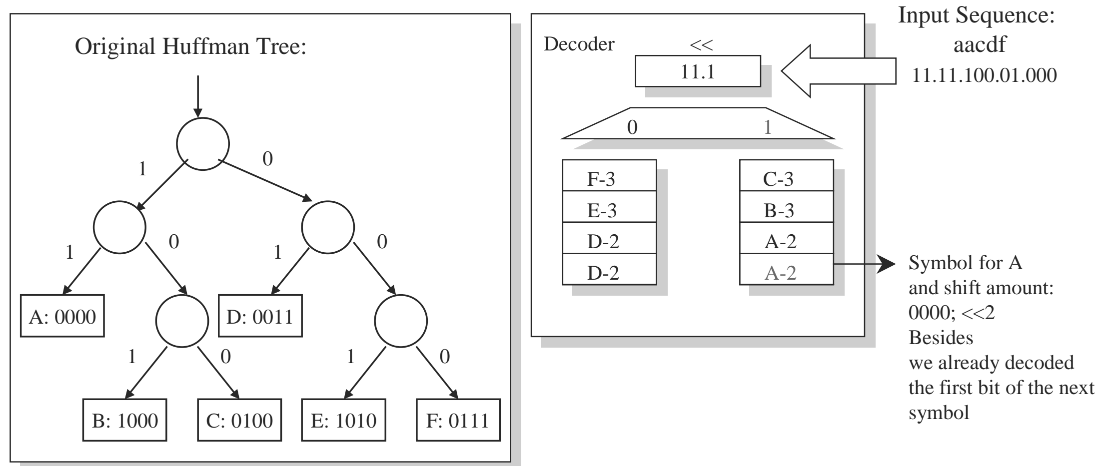


Figure 5.9 Dual Bank RAM Implementation of the Huffman Decoder

placed into the storage. The first and the last compressed groups will have a choice of placement (two lines each) while the uncompressed group will be placed at rigid location. So in this way all the content of the WUB is transferred into the compressed (main) cache storage. No stalls are produced by this process, so the store latency is zero cycles.

On a read from the cache (the processor executes a load), three separate locations within the cache are checked in parallel: the main compressed storage, the WUB, and the Read Uncompressed Buffer (RUB) (see Figure 5.1). The read uncompressed buffer holds the mostly recently read cache block in uncompressed form. This once again means that up to eight

consecutive uncompressed cache blocks can actually reside simultaneously in the RUB. If the read hits in either the WUB or RUB the access could be served in a single cycle (or whatever is equivalent hit time for direct mapped cache in the current setup). If the read hits in the main compressed storage access time depends on whether the data accessed is in compressed or uncompressed form. If the data uncompressed, the latency is the same as for conventional direct mapped cache. If the block is compressed, the latency might vary depending on decompression mechanism employed. The entire flow graph for this process could be found in the Appendix Table 3.

Now we should recall that the compression algorithm employed – the adaptive Huffman – have periodic adjustment phases, which change the frequency distribution used. On the catastrophic event when the compression algorithm is being adjusted, certain actions should take

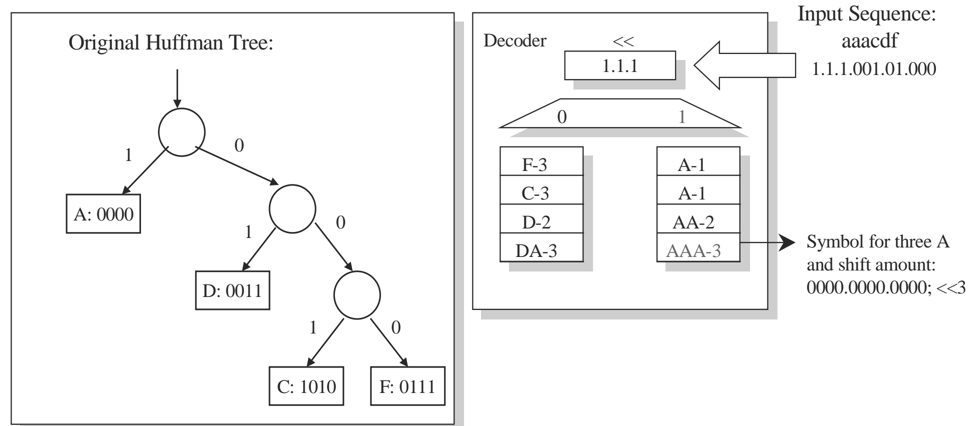


Figure 5.10 Multiple Symbol Decoding Example

place in the cache storage. As was mentioned before in Section 3.1 in most cases this is a rare event and we can dedicate a software interrupt to serve it. This interrupt will use the profile information accumulated since the last regeneration point and produce new and improved

distribution to be used by the compression algorithm. From an architectural standpoint, all previously compressed items should be discarded because new algorithm would be unable to interpret them. But, it does not mean we should completely purge the cache storage. First, just a portion of the data in the cache is actually compressed, so the uncompressed portion is not affected by compression algorithm change and could remain untouched. Second, the compressed part could be uncompressed with the old algorithm prior to its regeneration, and left at the appropriate location (according to the direct mapping placement algorithms). Regardless of that certain portion of resident cache blocks does get lost, but all these features allow lessening the impact of the compression algorithm regeneration on the overall cache performance.

5.3 Dynamic Decoding Structure

Now we have come to the most important part of the compressed cache design - the

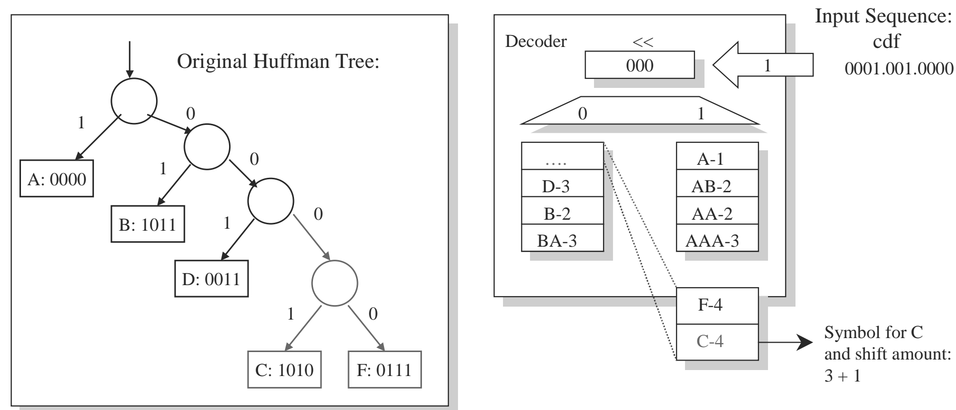


Figure 5.11 Biased Huffman Tree Example

decoder. Unlike the decoder for the instruction cache, this decoder needs to be reprogrammable, because the compression/decompression algorithm changes periodically. This fact means that no static and fast hardware structure could be built, so we need to propose a way to implement the adaptive Huffman decoder in reprogrammable form. In addition to that we cannot rely on pipelining to reduce the overall latency of access. Unlike for the code segment and instruction cache, the data cache reads could be spread apart in time with multiple cycles between them, so the pipeline hardly could be kept filled. Finally, it should occupy minimal physical size because otherwise the whole purpose of using a smaller smart cache instead of a large unsophisticated one might be defeated all together. Figure 5.7 presents the *logical/functional organization* of such a decoder as a priority decoder.

The Huffman code fragment, needed to be interpreted, addresses the RAM. The RAM storage entry holds the uncompressed symbol and shift amounts for each original code (Huffman

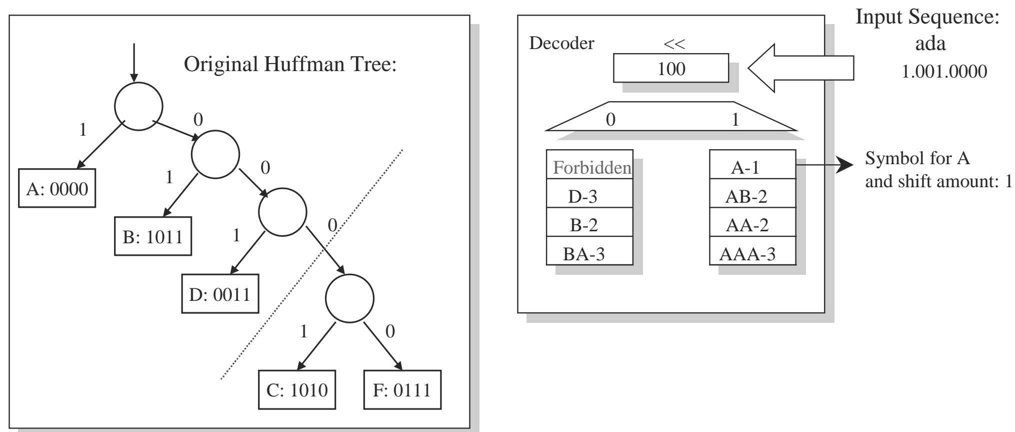


Figure 5.12 Restricted Huffman Decoder Structure

alphabet entry). The shift register is capable of holding up to $2^{LongestCodeSize}$ bits of the input encoded stream, which are then passed as an input to the left-to-right priority decoder. Match is

detected when all left-side bits for the RAM contents match the left-most part of the shift register. Once a match is found, a symbol is generated (based on the code location in the RAM – simply the line number), and the shift amount is used to update the shift register (so the decoded symbol is shifted out). The critical part here is that multiple cycles are needed to decode one cache block and a large physical size for the RAM storage and the priority decoder logic. With these points in mind, the following Dual Bank RAM implementation was proposed (see Figure 5.9).

In this scheme, the RAM storage is split in two banks of equivalent size. This separation corresponds to the splitting at the root of the Huffman tree. In Figure 5.9, the right bank corresponds to all of the Huffman codes that begin with one and the left bank for all that begins with zero. Obviously the storage could not be split any more, since the shortest Huffman code could contain just a single bit.

The advantages of such a structure include multiple symbols decoding in a single cycle

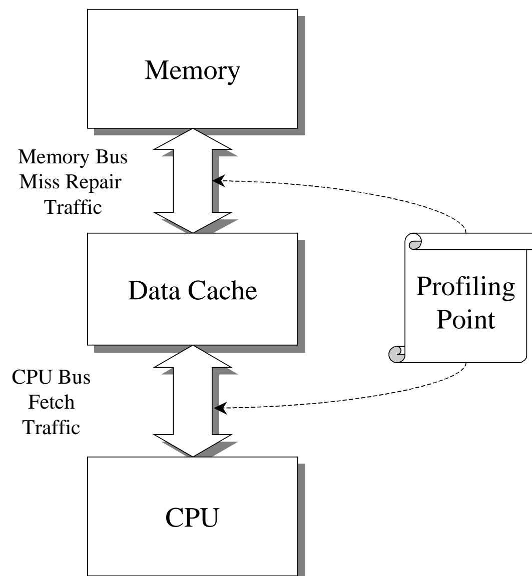


Figure 5.13 Profile Point Selection

and the fact that the critical ‘vertical’ search range is reduced in half. Figure 5.10 gives an example of three symbols being decoded in a single access to the dual bank RAM decoder. This case is far from purely theoretical - virtually all benchmarks in SpecInt95 at some regeneration period have a single dominating byte (usually zero), which outweighs the combined probabilities of all other symbols and gets encoded with a single bit.

Unfortunately, the opposite situation, where a tree is strongly asymmetric due to presence of an unlikely bit, is also possible (see Figure 5.11). In this case, we need more than

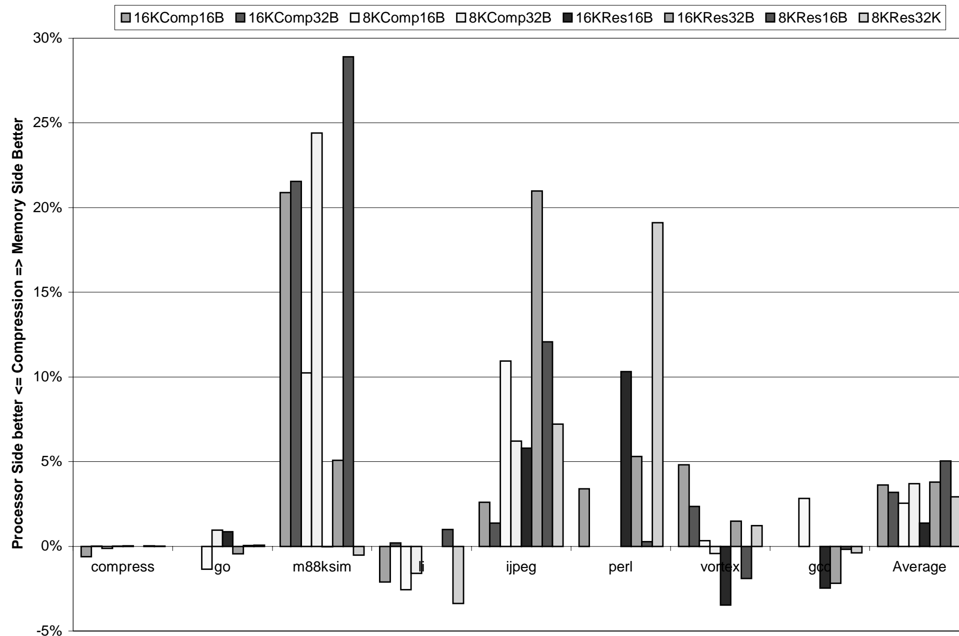


Figure 5.14 Compression Dependence on Profile Point Selection. Memory side vs. Processor Side

256 entries, potentially up to the $2^{LongestCodeSize}$ entries in the RAM storage. For a typical *LongestCodeSize* of up to 16 bits the RAM size gets unreasonable big.

Fortunately, there are several features specific to our setup that allows to optimize this decoder size. First of all, it is important to remember that codes are produced and consumed

locally in the cache. Second, according to the policy of not storing excessive codes, we are likely not to use long codes any way. From this, we are proposing the *Restricted Compression model*. The key feature of this compression model is the fact that only bytes whose Huffman codes are equal to or less then eight bits are compressed. All others are considered *forbidden*. The size limitation is chosen arbitrarily and could be varied. The biggest single advantage of using eight beet threshold is the fact that non of produced codes will exceed size of the original block, so no hardware 'adjustment' is needed to deal with longer codes. If a cache block contains a single forbidden byte, it is considered forbidden as well and is not compressed. This Restricted Compression model allows us to use only a 256-entry decoder. We guarantee that the encoder will not produce forbidden codes (the *LongestCodeSize* is now less then or equal to eight bit and total decoder size is fixed) (see Figure 5.12).

Just as have been outlined above, the decoder RAM is split into two banks (each one now is 128 lines long). The leftmost bit of the shift register selects the bank. The rest of the bits in the shift register (seven bits) serve as an address into the correspondent bank. As we showed before, the addressed RAM line might actually contain multiple codes (see Figure 5.10). Once the line is found, it produces one or *several* symbols and a *cumulative* (for the several codes being uncompressed) shift amount. With this implementation, the speed of decoding will vary with the data being decoded, and the most likely (and shortest) symbols will be decoded the fastest. To decode a 16-byte block we might need between two and 16 cycles. If the decoder is sub-clocked at half the cache clock time it translates into between one and eight cycles.

The encoder is actually 'the easy part', when compared to the decoder. Just as we mentioned earlier in the Section 5.2, since encoding is a rarely performed operation, it can get a dedicated operating system (OS) interrupt. This interrupt will generate new encoding and

update the changing part of the hardware decoder. This interrupt will have to perform all actions outline earlier including new distribution and new optimal algorithm generation along with selective purging of data compressed with old algorithm. It is important to note that no immediate *recompression* with the newly generated algorithm was considered at this time, but it is a possible option. Though the algorithm regeneration is a catastrophic event we can reduce its occurrences by dedicating minimum amount of hardware. This will have to be a small hardware profiler that is updated in parallel with data compression and holds the degree of compression since last regeneration point. It is important to notice that we still maintain minimum time granularity in asserting this hardware (every 1000 references for example). This will prevent us from multiple 'back to back' algorithm regeneration. Once the measured degree of compression falls bellow certain threshold, the interrupt to generate new encoding is generated. With the current threshold set at 70% of the original size, only five to twenty percent of checkpoints call for actual algorithm regeneration.

A much more important and interesting question is how to collect accurate profile information for all bytes in the input data stream. Generally this profile should give us the frequency of any byte appearance in the data stream since the last regeneration point and its accuracy defines effectiveness of compression in general. The straightforward solution would be to maintain an array of 256 counters and update them as we progress. This update is occurring in parallel with the fetch process and does not impose any time penalty. Nevertheless, the use of this array of counters equals an increase in the hardware budget. Let us investigate this issue in greater details.

5.4 Variations on the Compressed Data Cache Design

As have been just mentioned, an important option and interesting question in regard to

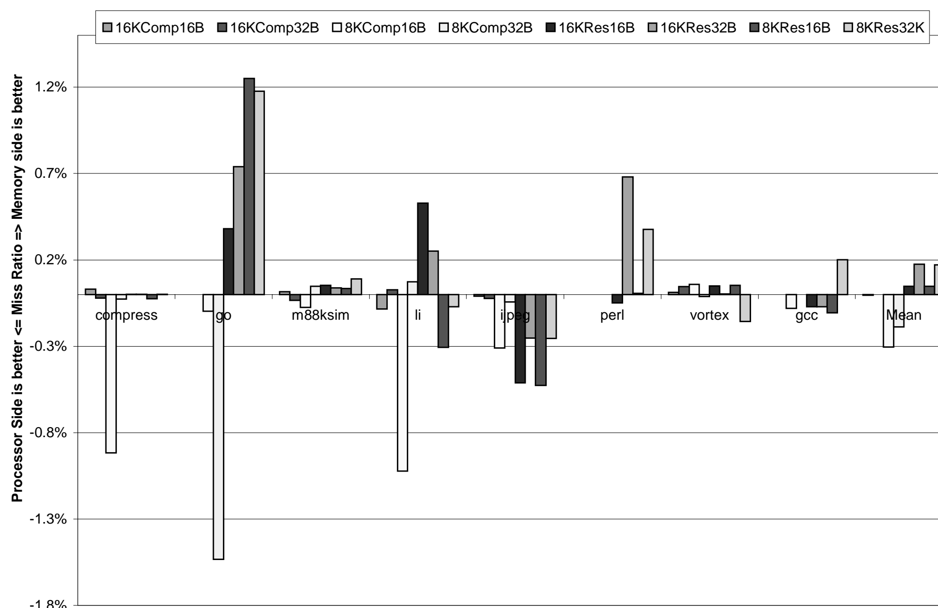


Figure 5.15 Miss Ratio Dependence on Profile Point Selection. Memory side vs. Processor Side

profile accuracy is where the statistics on the data stream are being collected. First it was briefly addressed in Section 3.1, and now it is time for the detailed discussion. As we remember from the code segment, there was no such issue as profile point selection. The code segment was statically available and optimized for static storage. Here we optimizing the dynamic stream for dynamic storage and the question is: which part of the stream is more representative.

The obvious choices are whether to collect statistics at the *processor* or *memory* side of

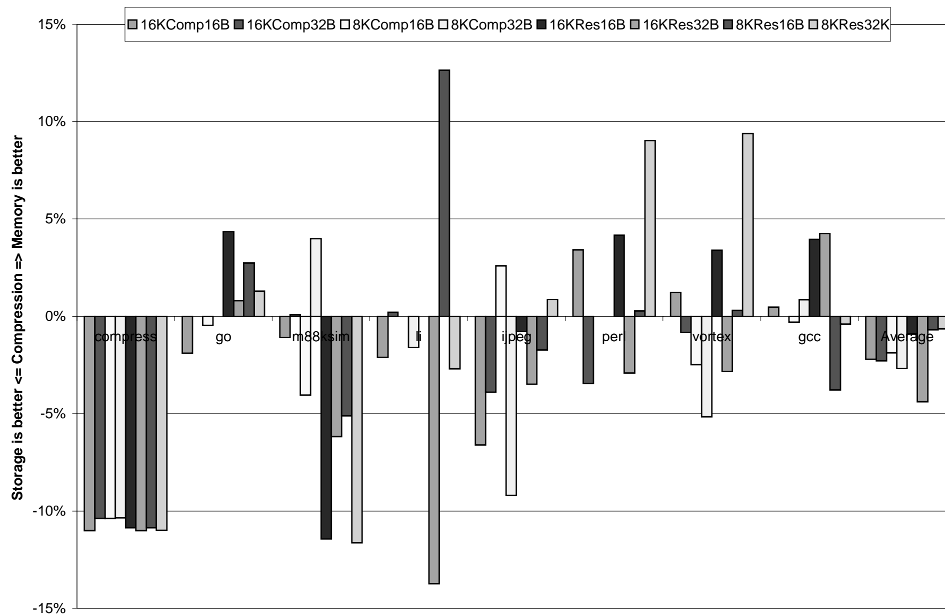


Figure 5.16 Compression Dependence on Profile Point Selection. Memory side vs. Storage Contents

the cache (see Figure 5.13). The processor bus side gives us a picture of which bytes are actually used by the processor. Theoretically, bytes more commonly used by the processor could be narrowed down. Then, once compressed those bytes get the best size reduction opportunity. Nevertheless, to profile the processor side stream, more references to the profiling hardware are needed. This imbalance does not impose immediate performance penalty, but in a long run might turn out to be a power issue. On the other hand, the memory side monitor only sees the stream of bytes in response to miss repair (we do not monitor write backs for obvious reason), which is normally less then ten percent of references of the processor side. It will give us slightly different distribution, which is more representative for bytes being not in storage when needed.

The Figure 5.14 summarizes the difference in degree of compression and the Figure

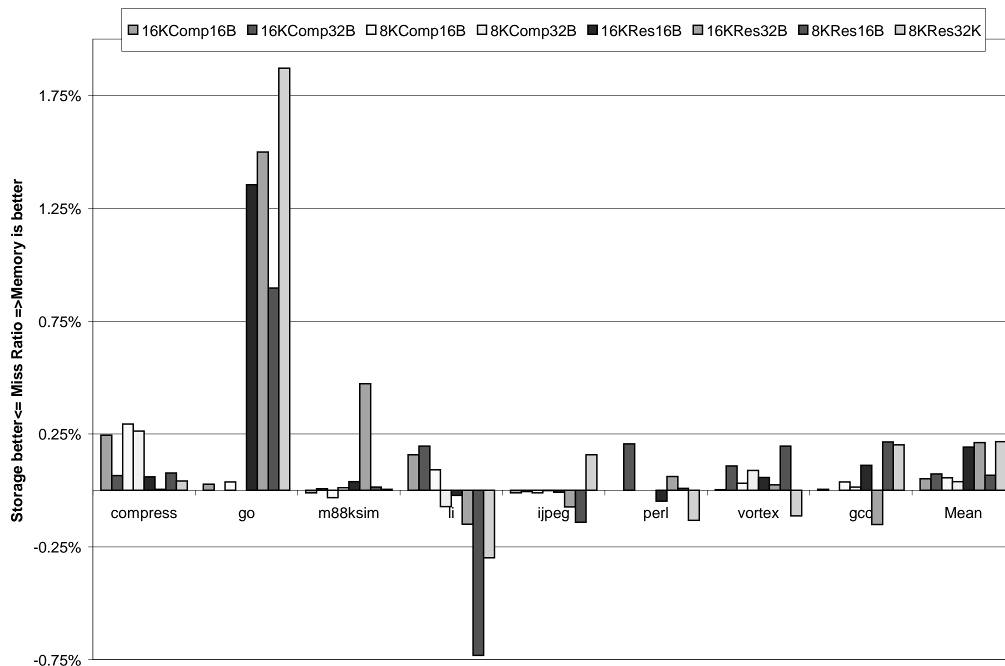


Figure 5.17 Miss Ratio Dependence on Profile Point Selection. Memory side vs. Storage Contents

5.15shows the difference in miss ratio based on where the data were collected. As we can see on average memory side is doing better in compression, and being less often accessed (comparing to the processor side) is clearly a better choice for implementation. This result in some way is counterintuitive. Nevertheless, once we analyze which blocks are resident in cache we will find out that blocks that are mostly accessed by the processor are also the longest residing in the storage. In addition they are also the ones that stay the longest in the read buffer in uncompressed form. On the other hand, blocks that are comparatively rarely accessed are causing the most of *cancellations* due to conflicts in the storage. If those blocks are compressed the most, those effects are lessened. A naïve but excellent example of this phenomenon would be pouring a cane of poppy seed into a *full* can of beans. It will fit almost entirely into the empty

space in-between the beans. And since the poppy seed is the one to go in and out of the can all the time, it better be small, so beans do not have to be removed.

Nevertheless, there is yet another, and in some way, more elegant solution for the profile collection. Generally we can generate the byte distribution based on the current *contents* of the data cache. It should represent a snapshot of activity during the period since the last code generation and it does not take any additional hardware to keep the statistics. As the OS interrupt goes through the storage and selectively purges the compressed blocks it must uncompress them first in order to leave the block that maps into the current line resident and

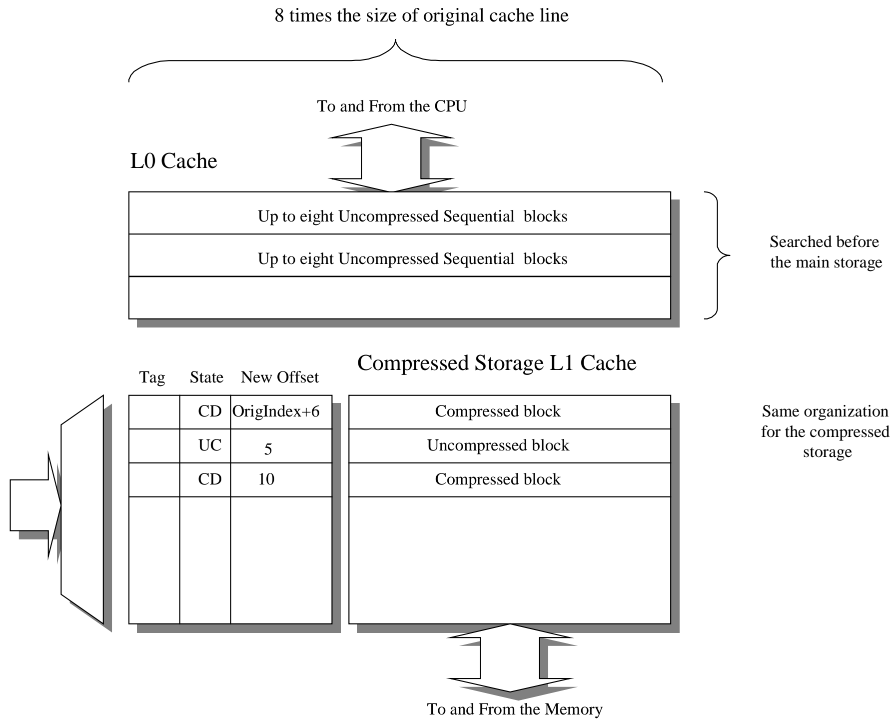


Figure 5.18 Two Level Compressed Data Cache

uncompressed (see Section 5.2). At the same time, the desired byte profile could be performed. This distribution should be blind to frequency of byte usage by processor. It might turn out to be either advantageous or degrading. As could be seen from the Figure 5.16 after experimenting

with all the three options, the assumption about low representativeness of the processor side stream turned out to be true – for most of benchmarks storage profiling gives better compression. Nevertheless once we look at Figure 5.19 we can see that overall miss ratio is virtually unchanged. It might mean two things. First is the possibility that we already have near optimal profiling with memory side monitoring. The second possibility is that it is not as much important *how much* we compress but rather *what* we compress. But for the reason of the smaller hardware budget and lower power consumption we should recommend the storage profiling as a better solution.

The next option needed to be described is the sizes of read and write buffers. If one

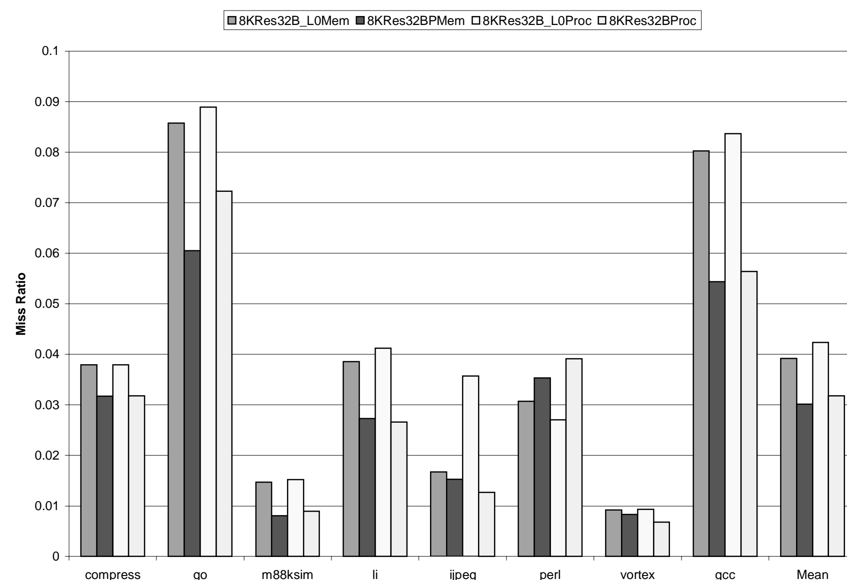


Figure 5.19 Miss Ratio for Two-Level Cache compared with the Original Implementation

carefully analyses the structure of the original compressed data cache (Figure 5.1) it could be seen that the read and write uncompressed buffers play significant role in the operation of this cache. In essence, they could be viewed as a small L0 cache split into two parts: read and write branches. The question is what is going to happen if we significantly increase the size of those

buffers. In fact we would probably want to merge both buffers and slightly reorganize overall design.

The Figure 5.18 presents the revised design. Both the read and write uncompressed buffers are merged into one L0 storage array. Each line in the L0 can hold up to eight *sequential* memory blocks. The fact that resident blocks are sequential is rather important here. It allows easy compressed block formation, but results in low hardware utilization, since one hundred

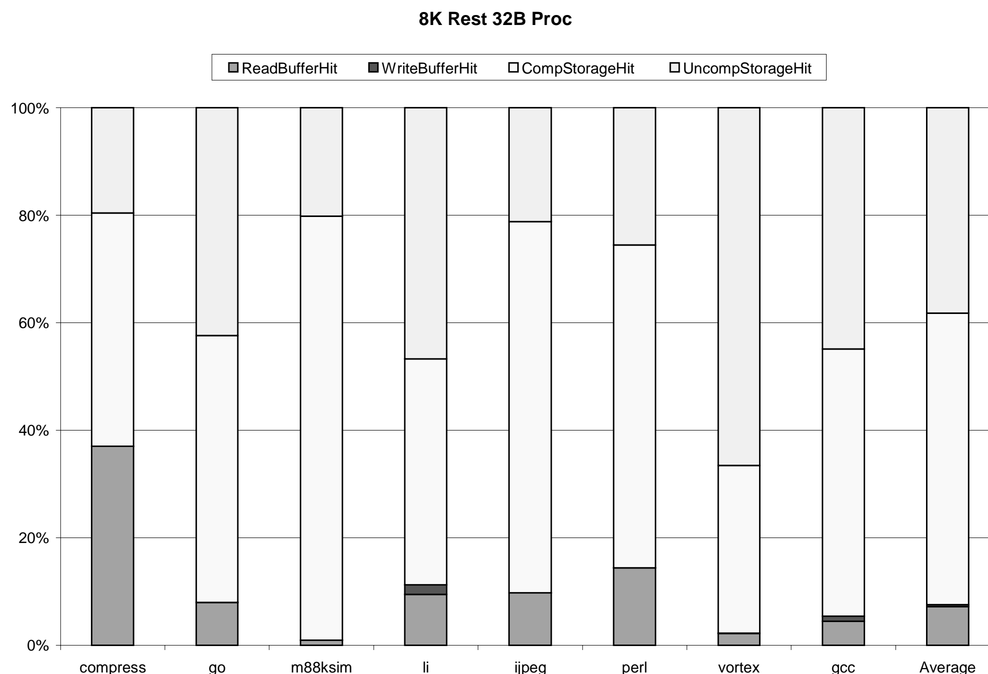


Figure 5.20 Reference Hit Breakdown for the Original Compressed Data Cache

percent compression is an infrequent event. The L0 filters all accesses to the compressed cache storage – both write and read first go through it and only then access the main storage. The size of L0 is set to 32 lines. In this configuration, a byte frequency profile is collected between L0 and main storage, with additional monitoring of the processor side of L0. This is done to guarantee the proper distribution for code regeneration – we do not want build our compression

solely on the storage contents or memory side. The reasons for that were described in great details in 4 and includes the change in the data stream entropy filtered by a caching structure.

The first set of experiments for the two-level cache revealed its low performance and

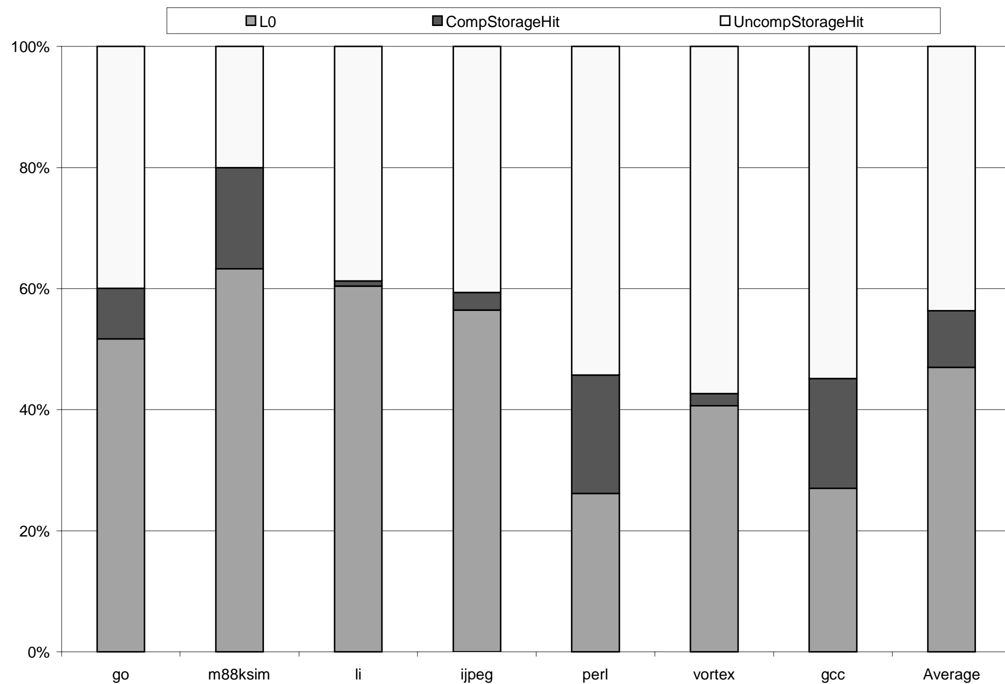


Figure 5.21 Reference Hit Breakdown for the Two-Level Data Cache

practicality (see Figure 5.19). In this figure, one of the configurations of the original compressed data cache is compared to the two-level cache. For the bigger hardware budget we got slight *degradation* of performance for most of the benchmarks. The only exception is the perl benchmark, which exhibits high degree of spatial locality, so as a result, most of the used data fit into the L0 cache.

In order to further analyze this degradation of performance in two-level cache we need to look at internal cycle distribution of both original and the two-level caches. In Figure 5.20 the ReadBufferHit part of the bar corresponds to all the references that hit in the uncompressed read buffer (the read from the same block that was read recently and still resides in the read buffer). The WriteBufferHit part corresponds to all of the references that hit in write uncompressed

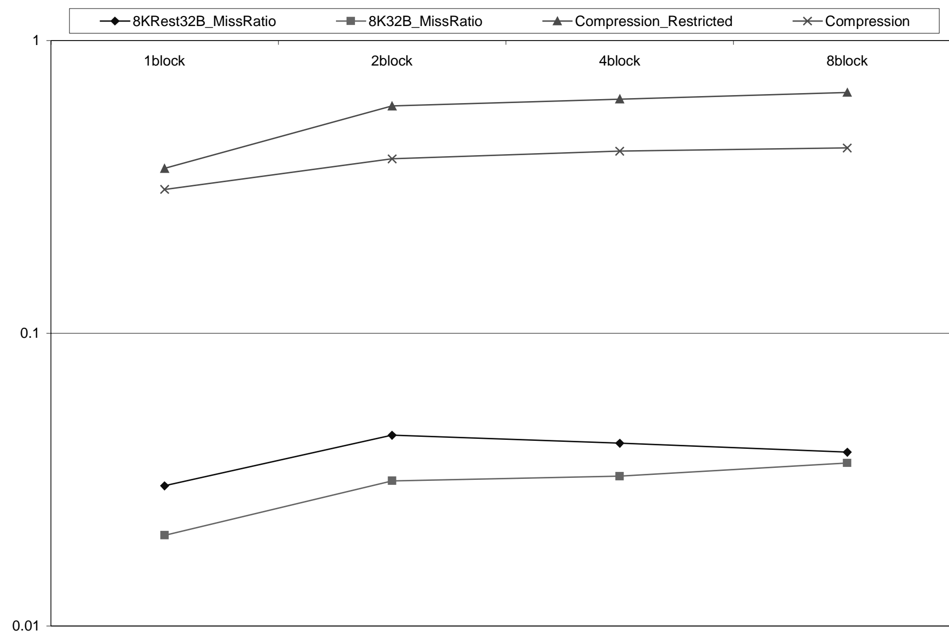


Figure 5.22 Two Level Cache Size Variation (Logarithmic Scale)

buffer (a write followed by a read from the same address before the block got a chance to be compressed). The other two sections correspond to the main compressed storage. Since a reference could hit in either compressed or uncompressed block we must differentiate. As we can see from Figure 5.21, majority of the references hit in the compressed storage. This fact is the main reason for the performance increase of compressed data cache comparing to the uncompressed one.

If now we analyze similar distribution for the two-level cache, we will see completely

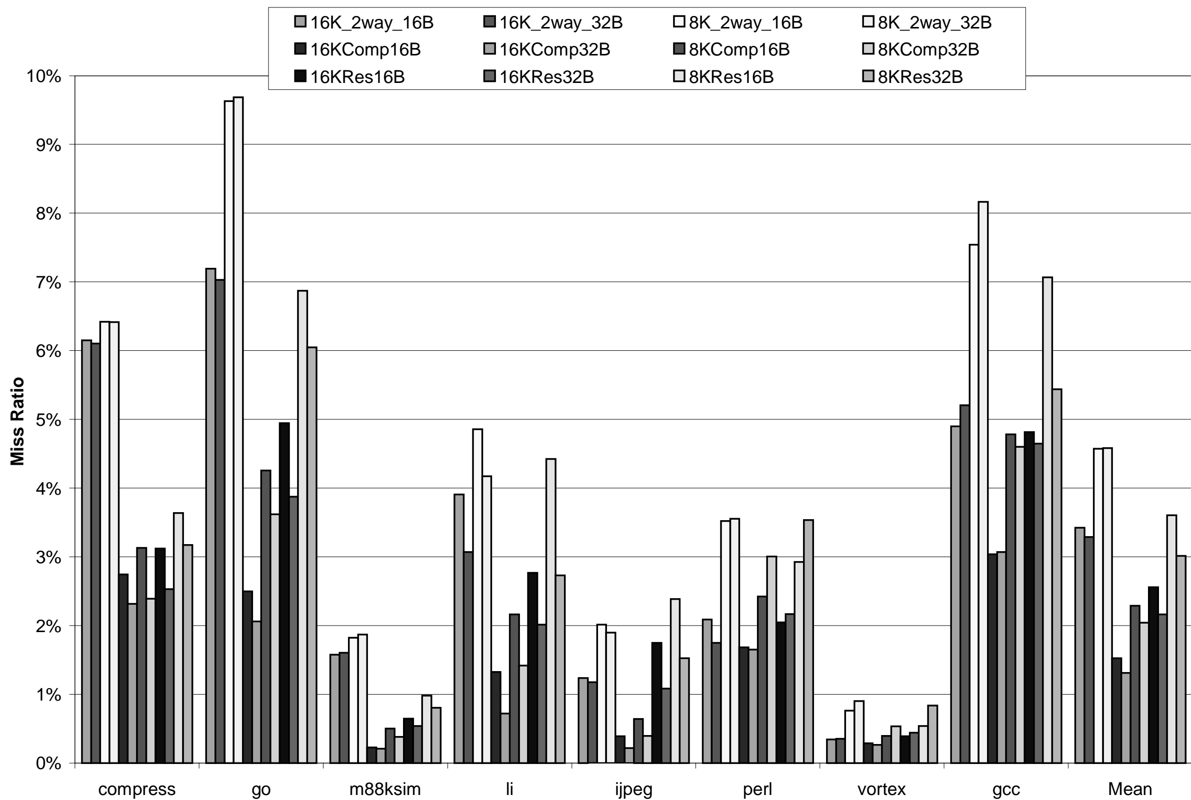


Figure 5.23 Miss Ratio Comparison between Compressed and Uncompressed Caches

different picture (see Figure 5.21). Now the majority of references hit in the L0 storage, which effectively filters out references to the main storage. As a result small portion of data does get compressed, and it could be seen that only a small fraction of the references that got to the main storage actually hit in the compressed block.

The final variation in the L0 experiment is varying the size of the L0 cache from one to eight lines. The results for mss ration and degree of compression are summarized in Figure 5.22. It can be clearly seen that with increase of the L0 size the compression performance degrades quickly (just as in the case with bus compression). The miss ratio drops at two-block L0 size and then slowly improves as the L0 size increases since L0 now serves majority of

references. It is also should be clear that overall performance per hardware unit uniformly degrades, since increase in L0 size equals total hardware budget increase with minimal return.

5.5 Final Configuration for the Compressed Data Cache

Once we have iterated over several design options we can come up with a proposal for the final and optimal solution for the compressed data cache design. With everything mentioned

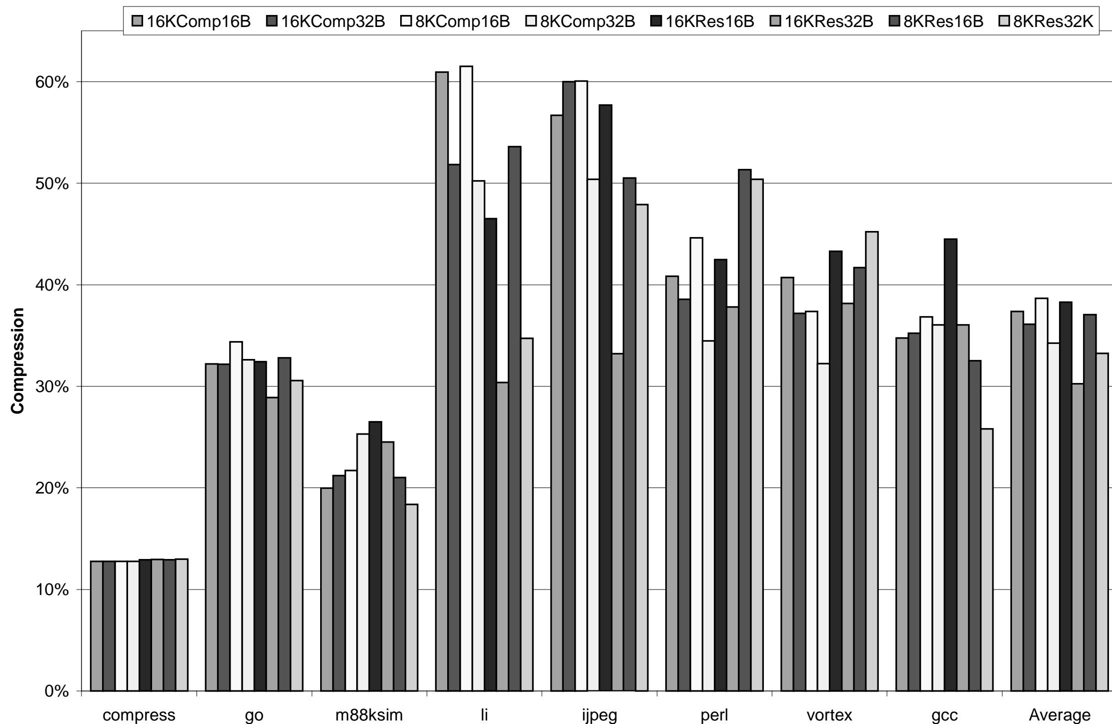


Figure 5.24 Absolute Dynamic Compression for Storage Profile Scheme

earlier the cache that uses original configuration and storage based profile collection with a choice of storage size between eight and 32 Kbytes and block size of 16 bytes could be considered near optimal. It is very important to note that it is only optimal for the current set of

benchmarks. For a different type of application this configuration should be revised. Also some of the applications (like streaming data processors) might be better without data cache at all.

The final simulation results for the best configuration are presented in Figure 5.23 and Figure 5.24. The compressed cache is compared against a similarly sized uncompressed two-way set associative cache. From the results we can see that an eight Kbytes compressed cache with restricted compression model on average performs as the traditional 16 Kbytes two-way set

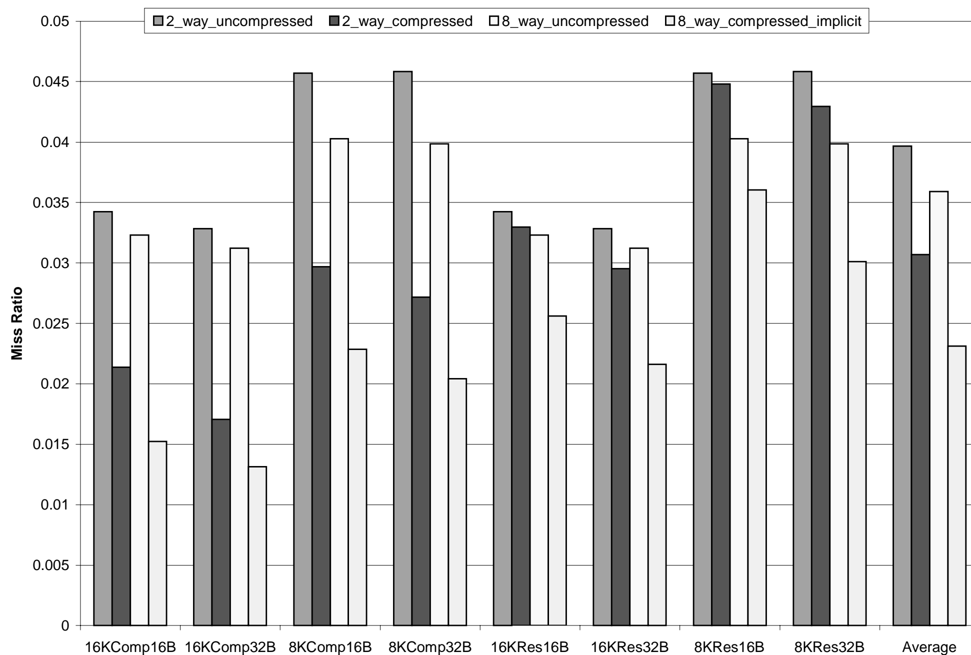


Figure 5.25 Entropy Miss Ratio Summary

associative cache. For more sophisticated compression model, the miss ratio difference easily reaches a two-time reduction for the same size of storage. In order to prove the statement made in Section 2.5.1 about entropy miss ratio the following set of experiments was performed. The compression algorithm was limited to maximum of 2x compression rate and 2way-like placement policy. The comparison between correspondent 2 and 8 way set associative traditional cache vs. 2x and 8x limited compressed cache performed. The results are

summarized in Figure 5.25.

From everything said so far it could be conclude that even with restricted compression model the compressed data cache can perform better then a similar uncompressed cache, and intelligent choice should be made for every specific architecture whether it can benefit from compression of data cache.

6 Conclusions and Future Work

The main purpose of this work is to study the available redundancy and potential compressibility of an embedded processor program in general and find ways to use it. As a result, we can reduce both static and dynamic program size, which results not only in increasing overall performance but also in smaller code ROM size and lower power consumption. It is important to reemphasize that ultimate goal of higher performance and smaller program at the same time was achieved. The system with compressed encoding and redesigned instruction cache actually runs faster than the system with the original configuration and native code.

If was also shown that there is no single engineering solution to the problem, but rather a combination of several different techniques aimed at the same goal. Part of this approach is to use compiler as extensively as possible. For example we use it to extract the pipeline decoder logic for an embedded processor in software at system development time. By doing this we facilitated flexible approach to the design of this decoder, and as a result permitted the compression. In essence the compiler is employed not only for removing complex decisions from run time into compilation time (VLIW approach) but also for dictating the overall system architecture according to the implementation. This customization is performed in light and transparent fashion and only requires modification of instruction and/or data cache while the rest of the system remains unchanged.

As for the instruction fetch pipeline design, by Huffman compressing and Tailor coding the ISA of the original program significant code segment size reduction was achieved. This

work also detailed the design of instruction fetch mechanisms for both this compression schemes, and then discusses their performance and cost tradeoffs. Some interesting results were found. In particular, the degree of compression for the ROM doesn't necessarily translate into an improvement in instructions delivered per cycle. Experiments found that when the missprediction penalty of the added Huffman decoder stage was taken into account, the Tailored instruction sat architecture approach produced a higher performance. Nevertheless, pipeline performance is not always the central goal of embedded systems. Methods like the Full Huffman compression scheme that operates at instruction cache hit time still achieved median performance advantage over the baseline, while providing significant ROM size savings.

Next close attention had been paid to the data stream compressibility and system data bus design as well. The amount of redundant data stored in expensive caches and transmitted through tight bottleneck of instruction fetch and data path is hardly tolerable. First we address the dynamic data stream redundancy by optimizing the system bus between CPU and memory. The main conclusion from that study was that if there is no caching structures are present in the system, significant gain could be achieved from coding the bus. This gain is expressed in both higher throughoutput and lower power consumption. But if there is even a small cache used on the CPU side, hardly any optimizations are possible.

Next we turned to the data cache design. By reducing redundancy of data stored in the data cache we break an age-old capacity limit of cache storage and defined a new entropy capacity limit. With the modern applications considered the effective capacity of a cache can easily be doubled (expressed in reduction of miss ratio) by partially compressing the cache storage. The main challenge encountered in compressed data cache design was the adaptive decoder for compressed data. Since the data cache could hardly be pipelined (or just would not

benefit from it) the latency for decompression of data is critical. Two different approaches to the reconfigurable data decompression has been advised – full and restricted compression scheme. While the full compression scheme utilizes every opportunity for compression and generally yields near optimal results, the restricted model is much simpler. Because of this simplicity it is more practical to implement in real hardware, so once again, similarly to the code segment compression we can see the tradeoff between degree of compression and complexity of the decoder. Next some variations in the design of the compressed data cache have been considered. The main and the most interesting conclusion was that multi-level caching where first (zero) level is uncompressed and the next one is, would defeat the purpose of cache compression in the first place.

Regardless of the comprehensiveness of the study, many questions remain open, and some new perspectives are unrevealed. For the code compression, it is important to consider different compression algorithms and variations of the ones used. It is also very important to consider the use of different atomic fetch units like superblocks and possibly treegions. These blocks should significantly reduce the overhead due to address translation table, since it will require fewer entries. If these blocks will ever be used in the compressed data cache, better branch predictors will become necessary. Otherwise the pollution of cache will become unavoidable which will defeat the purpose of compressing it in first place.

And last but not least, after the findings in data cache decoding technology we might try to use a restricted compression model in compressed instruction cache as well in order to simplify decoding. It is also might be possible to collapse the extra pipeline stage needed for decompression and shorten the instruction fetch pipeline. For a more general view creation of a unique compression algorithms based on a unique and complex cost function for each

application is interesting.

For data compression the future work should include different decoders construction and compiler optimizations to aid high data entropy and reference locality. Generally speaking if compiler technology would be perfected to the point where no low entropy data transfers were needed the current research would become obsolete. All those issues are reserved and suggested as a future work.

References

[1] Andrew Wolfe, Alex Chanin "Executing Compressed Programs on An Embedded RISC Architecture", *In Proceedings of 25th International Symposium on Microarchitecture*, 1992

[2] D.A. Huffman "A Method for the Construction of Minimum-Redundancy Codes", in Proceedings of the IRE, Vol. 4D, pp. 1098-1101, Sep. 1952

[3] Clifford Liem, "Retargetable Compilers for Embedded Core Processors", Kluwer Academic Publishers, 1997.

[4] William .A. Havanki "Treegion scheduling for VLIW processor" MS thesis. Dept. ECE North Carolina State University, Raleigh NC, 1997

[5] Sanjeev Banerjia, William A. Havanki, Thomas M. Conte "Treegion scheduling for highly parallel processor" *in Proceedings of Euro-Par'97* (Paris, France) 1997

[6] William A. Havanki, Sanjeev Banerjia, Thomas M. Conte "Treegion Scheduling for Wide Issue Processors" *in Proceedings of the 1997 4th International Symposium on High-Performance Computer Architecture (HPCA-4),* (Las Vegas), Feb. 1998.

[7] Thomas M. Conte, Sanjeev Banerjia, Sergei Y. Larin, Kishore N. Menezes, Sumedh W. Sathaye, ``Instruction fetch mechanisms for VLIW architectures with compressed encodings". *In Proceedings of the 29th International Symposium on Microarchitecture (Paris, France),* pp.201-211, Dec. 1996.

[8] Sanjeev Banerjia, Kishore N. Menezes, Thomas M. Conte "NextPC Computation for Banked Instruction Cache for VLIW architecture with a Compressed Encoding"

Technical report Dept. of ECE, North Carolina State University, Raleigh, NC 27695-7911, June 1996.

[9] Jens Ernst, William Evans, Christopher W. Fraser, Steven Lucco Todd A. Proebsting "Code Compression" In *Proc. of the '97 International Conf. on Programming Language Design and Implementation*, (Las Vegas, NV) 1997

[10] M. Game, A. Booker "CodePack: Code Compression for PowerPC Processors", IBM Microelectronics Division, RTP NC.

[11] Thomas M. Conte, "The TINKER Machine Language Manual" North Carolina State University, Raleigh NC 27695-7911, 1995.

[12] K.D. Cooper, N. McIntosh "Enhanced Code Compression for Embedded RISC Processors" In *Proc. of the '99 International Conf. on Programming Language Design and Implementation*, (Atlanta, Ga) 1999

[13] C.W. Fraser "Automatic Inference of models for Statistical Code Compression" In *Proc. of the '99 International Conf. on Programming Language Design and Implementation*, (Atlanta, Ga) 1999

[14] J.E. Smith "A Study of Branch Prediction Strategies" *Proc. 8th Ann. Int'l. Symp. Computer Architecture*, 1981.

[15] Stan.Y. Liao, Srinivas Devadas, Kurt Keutzer "Code density optimization for embedded DSP processors using data compression techniques" In *Proc. of 16th Conference on Advanced Research in VLSI*, (Los Alamitos, CA) 1995.

[16] Joseph A. Fisher "Trace Scheduling: A Technique for Global Microcode Compaction" *IEEE Transactions on Computers*, Vol. C-30, No. 7, July 1981.

[17] M. Kosuch, A. Wolfe "Compression of Embedded System Programs" *IEEE*

International Conference on Computer Design, October 1994.

[18] M. Benes, A. Wolfe, S.M. Nowick "A High-speed Asynchronous Decompression Circuit for Embedded Processors" in *Proc. of the 17th Conference on Advanced Research in VLSI*, (Los Alamitos, CA) 1997.

[19] M.K. Rudberg L Wanhammar "New Approaches to High Speed Huffman Decoding" in Proc. of *ISCAS*, 1996.

[20] D. Alpert, D. Avnon "Architecture of the Pentium Microprocessor" *IEEE Micro*, vol. 13, pp. 11-21, June 1993.

[21] Vinod Kathail, Michael Schlansker, Bob R. Rau "HPL PlayDoh architecture specification" Technical Report HPL-93-80 HP Labs, Palo Alto,CA 1994.

[22] Wen-Mei W. Hwu, Scott A. Mahlke, W.Y. Chen, Pohua P. Chang, N.J. Warter,R.A. Bringmann, R.G. Ouellette, R.E. Hank, T. Kiyohara, G.E. Haab, J.G. Holm, D.M. Lavery "The Superblock: An effective structure for VLIW and Superscalar compilation" *The Journal of Supercomputing*, vol 7, Jan 1993

[23] J.A. Storer, T.G. Szymanski "Data Compression via Textual Substitution" *Journal of the ACM,* 29(4) pp. 928-951, October 1982.

[24] J. Kin, M. Gupta, W.H. Mangione-Smith "The Filter Cache: An energy efficient memory structure" in *Proc. 30th International Symposium on Microarchitecture*, Raleigh NC, Dec. 1997.

[25] Charles Lefurgy, P. Bird, I. Chen, Trevor Mudge "Improving Code Density Using Compression Techniques" in *Proc. 30th International Symposium on Microarchitecture*, Raleigh NC, Dec. 1997.

[26] S. Segars, K. Clarke, L. Goudge "Embedded Control Problems, Thumb, and the

ARM7TDMI" *IEEE Micro*, October 1995.

[27] K. Kissell "MIPS16: High-density MIPS for the Embedded Market" Silicon Graphics MIPS Group, 1997.

[28] Texas Instruments "TMS320C2x User's Guide", January 1993

[29] C. E. Shannon "A Mathematical Theory of Communication", The Bell System Technical Journal, Vol. 27, pp.374-423,623-656, July, October 1948.

[30] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery "Numerical Recipes in C The Art of Scientific Computing", Cambridge University Press. Second Edition 1997

[31] J. D. Gee, M. D. Hill, D. N. Pnevmatikatos, A. J. Smith "cache Performance of the SPEC92 Benchmark Suite" IEEE Micro 13:4, pp. 17-27, 1993

[32] M. J. Flynn "Very high-speed computing systems" in Proc. IEEE 54:12, December 1966

[33] Mircea R. Stan, Wayne P. Burleson "Low-Power Encodings for Global Communication in CMOS VLSI" IEEE Transactions on VLSI Systems, Vol.5, No, 4, Dec. 1997

[34] Mircea R. Stan, Wayne P. Burleson "Bus-Invert Coding for Low-Power I/O", IEEE Transactions on VLSI Systems, Vol.3, No, 1, Mar 1995

[35] M. Pedram, "Power Minimization in IC Design", *ACM Transactions on Design Automation of Electronic Systems* Vol.1, No.1, Jan 1996

[36] A. V. Aho, R. Sethi, J. D. Ullman "Compilers. Principles, Techniques, and Tools" Addison-Wesley Publishing Company.

[37] S. S. Muchnick "Advanced Compiler Design and Implementation" Morgan Kaufmann Publishers

[38] C. L. Su, C. Y. Tsui, A. M. Despain "Saving Power in the Control Path of Embedded Processors", IEEE Design Test Comput., vol 11, 1994

[39] J. Ziv, A. Lempel, "A universal Algorithm for Sequential Data Compression," IEEE Transactions on Information Theory vol 23, 1977.

[40] David W. Hammerstrom E. S. Davidson "Information Content of CPU Memory Referencing Behavior" *in Proc. 4th Annual Symposium on Computer Architecture* 1977 pp.184-192.

[41] Musoll, E.; Lang, T.; Cortadella, J. "Working-Zone Encoding for Reducing the Energy in Microprocessor Address Buses," IEEE Transactions on Very Large Scale Integration Systems, vol. 6(4) 1998

[42] IBM "CodePac PowerPC Code Compression Utility," User's Manual Ver. 3.0 IBM 1998

[43] Charles Lefurgy, Trevor Mudge "Fast Software-managed Code Decompression," in Proc. of Computer and Architecture Support for Embedded Systems pp. 139-143 October 1999

[44] Charles Lefurgy, Eva Piccininni, and Trevor Mudge "Reducing Code Size with Run-time Decompression," *Proceedings of the 6th International Symposium on High-Performance Computer Architecture (HPCA)* January 2000

[45] SPEC CPU 95, Technical Manual, August 1995

[46] Haris Lekatsas, Wayne Wolf "Random Access Decompression using Binary Arithmetic Coding" Technical Report Princeton University.

[47] Yukihiro Yoshida, Bao-Yu Song, Hiroyuki Okuhata, Tako Onoye, Isao Shirakawa " An Object Code Compression Approach to Embedded Processors" in *Proc. 30th International Symposium on Microarchitecture*, Raleigh NC, Dec. 1997.

[48] Charles Lefurgy, Eva Piccininni, Trevor Mudge " Evaluation of a High Performance Code Compression Method", in *Proc. 32th International Symposium on*

Microarchitecture, Haifa Israel, Nov. 1999

[49] D. del Corso, H. Kirrmann, J. D. Nicoud "Microcomputer Buses and Links" *New York Academic,* 1986

[50] S. Y. Larin and T. M. Conte, "Compiler-driven Cached Code Compression Schemes for Embedded ILP Processors,", in *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, (Haifa, Isreal), Nov. 1999.

[51] M. H. Lipasti, C. B. Wilkerson, J. P. Shen, "Value Locality and Load Value Prediction" in *Proceed. of the 7th International Conference on Architecture Support for Programming Languages and Operating Systems,* October 1996

[52] V. Peng "MIPS64 20K: Family of Processors and Core Designs", in *Proc. of the Embedded Processor Forum*, San Jose CA 2000

[53] P. Sandon "PowerPC 750CX: High Performance with Integrated Multilevel Caching", in *Proc. of the Embedded Processor Forum*, San Jose CA 2000

[54] R. W. Hamming, "Coding and Information Theory," Prentice-Hall, Englewood Cliffs, NJ, 1980

[55] Ian H. Witten, Radford M. Neal, John G. Cleary "Arithmetic coding for data compression" *Communications of the ACM, Volume 30, Number 6, p. 520-546,* June 1987

7 Appendix

Table 1. Cache study cycle count assumptions summary. Note that Base and Tailored do not employ a buffer, which is why Buffer Hit/Miss have no effect

			Base	***Tailored***	***Compressed***
Next Block	Cache Hit	Buffer Hit	1cycle	1cycle	1cycle
prediction		Buffer Miss	1cycle	1cycle	1+(n-1)
Correct	Cache Miss	Buffer Hit	1+(n-1)	2+(n-1)	1cycle
		Buffer Miss	1+(n-1)	2+(n-1)	3+(n-1)
Next Block	Cache Hit	Buffer Hit	2cycles	2cycles	1cycle
prediction		Buffer Miss	2cycles	2cycles	2+(n-1)
*Incorrec*t	Cache Miss	Buffer Hit	8+(n-1)	9+(n-1)	1cycle
		Buffer Miss	8+(n-1)	9+(n-1)	10+(n-1)

Table 2.

TEPIC Instruction set Summary.

Integer ALU Operation

1	1	2	5	5	5	2	8	5	1	5
T	S	OPT	OPCODE	Src1	Src 2	BHWX	Reserved	Dest	L1	PREDICATE

Integer Compare-to-Predicate Operation

1	1	2	5	5	5	2	3	5	5	1	5
T	S	OPT	OPCODE	Src1	Src2	BHWX	D1	Reserved	Dest	L1	PREDICATE

Integer Load Immediate Operation

1	1	2	5	20	5	1	5
T	S	OPT	OPCODE	Src1	Dest	L1	PREDICATE

Floatin Point Operation

1	1	2	5	5	5	1	6	3	5	1	5
T	S	OPT	OPCODE	Src1	Src2	S/D	Reserved	tssL/U	Dest	L1	PREDICATE

Load Operation

1	1	2	5	5	2	2	1	2	3	5	5	1	5
T	S	OPT	OPCODE	Src1	BHWX	SCS	Res	TCS	Reserved	Lat	Dest	Rsv	PREDICATE

Store Operation

1	1	2	5	5	5	2	2	11	1	5
T	S	OPT	OPCODE	Src1	Src2	BHWX	TCS	Reserved	L1	PREDICATE

Branch Operation

1	1	2	5	5	5	16	5
T	S	OPT	OPCODE	Src1	Counter	Reserved	PREDICATE

0 … 39

Table 3

Compressed Data Cache Control Flow Chart