Benefits of Value-Range Cache: A Limit Study

Ramya K. Narayana

University of Wisconsin-Madison
Electrical and Computer Engineering
1415 Engineering Drive
Madison, WI 53706

Submitted in partial fulfillment of the M.S. Degree in Electrical and Computer Engineering
Project Option
Jan 19, 2004

Abstract

The ever increasing gap between CPU and the memory system performance encourages us to look at various ways to utilize memory more efficiently. Alternative ways of storing data or compressing data in main memory provides an important alternative. Many different compression techniques have been suggested before. In this report, we look for the occurrence of a pattern in main memory that we define as *value-range*. A *value-range* data represents consecutive bytes of memory that have the same value, with three parameters: the start address, length and the value. We examine the frequency of such data in memory by taking a memory snapshot at some time during the execution of a benchmark. Our observation shows that such data occur quite often in main memory. Then we go on to save such data in a cache that we call the *value-range* cache. Thus, this cache contains data that has the same value in consecutive bytes, represented in a compressed form with a start address, length and value. On an L2 miss we check if the required data occurs in the *value-range* cache and if it does we satisfy the L2 miss from the *value-range* cache. Only data that miss in the *value-range* cache are sent to the main memory. We implement the *value-range* cache with infinite capacity for our limit study. This enables us to find out the maximum possible benefit from this scheme. We find that for most SpecInt2000 benchmarks less than 5% of the L2 cache misses can be satisfied with *value-range* cache. This scheme improves the IPC of most benchmarks by less than 3%. We discuss the possible reasons for this meager performance benefit and suggest alternatives that could be explored to improve performance.

1.0 Introduction and Motivation

Performance of modern computer systems is limited, to a large extent, by the huge delays in accessing data from memory. Moreover, advanced computer applications continue to use more and more memory, this growth rate in memory requirements is expected to continue. In general, increasing the memory size leads to larger delays in accessing data from the memory. These conflicting demands make it necessary to look at ways to use memory more efficiently. Compressing data in memory is one such option. Different compression techniques offer alternative ways of storing data in memory.

Different forms of code compaction have been investigated from the early days of computing [14]. Thumb architecture [15] from Arm has shown the commercial appeal of compact code for embedded systems. In a workstation environment, however, code represents only a fraction of the overall memory usage, and so any efficient implementation of compression must address both code and data. Main memory compression has been suggested in [16-18,21-22] and with high performance hardware compressors [18-20] it has been shown that it has the potential to substantially improve system performance [17,18].

In order to recommend a data compression scheme it is important to understand the characteristics of data in the main memory at different points of time during the execution of a program. Past research in this area studied the contents of memory for different applications. [6] measured the list structure of five Lisp programs and found substantial regularities in the data. Based on these regularities a more space efficient representation of the data was discussed. [1] analyzed the characteristics of memory-data from 10 Unix applications. They defined memory-data as any information stored in main memory during the execution of an application, including both code and data. They found that memory-data contains a large portion of zeroes and that zeroes often occur in contiguous runs. Also, integral power-of-2, -1 and low values have greater than average probabilities. They present the compression ratio results by applying two compression algorithms [7,8]. They show that memory-data typically compresses to half using these methods. These results encourage us to explore main memory compression opportunities with SpecInt2000 benchmarks. Section 2.0 presents the characteristics of data in the main memory for the SpecInt2000 benchmarks.

After establishing the potential for data compression, many different proposals have been suggested to utilize this property to improve the performance of the computer system. Below, we look at a few such proposals to understand their merits and problems.

Jang-Soo Lee et.al. [2] explore the potential of an on-chip cache compression technique, which can both reduce cache miss ratio by increasing the effective memory space, and improve the off-chip bandwidth by transferring data in compressed form, if main memory is also managed in compressed form. The results presented from trace-driven simulation show that their approach can provide around 7%-90% decrease in the on-chip cache miss ratio as well as a 9%-95% decrease in the data traffic over the conventional memory systems depending on the SPEC95 benchmark programs. However, their approach has at least two potential problems. First, when the processor requests a word within a compressed data block stored in the compressed cache or main memory, the compressed block has to be all decompressed on the fly and then the requested word transferred to the processor. This decompression time has a critical effect on the memory access time and offsets the compression benefits. Another problem associated with their compressed memory system in that compressed blocks can be generated with different sizes depending on the compression efficiency. Therefore, the length of any compressed block can even be longer than that of its source block in the worst case. In addition, when a compressed block is decompressed, modified, and re-compressed, its new compressed block cannot be stored in the old position in the compressed memory if its length is longer than that of the old compressed block.

Krishna Kant and Ravi Iyer in [3] focus on designing simple compression schemes that helps reduce the amount of information transferred between the processor caches and the memory subsystem. This compression is primarily geared towards improving the performance and efficiency of the transfer medium (busses, links etc). They evaluate the potential of the basic compression techniques for two commercial workloads – SPECweb99 [23] and TPC-C [24]. They show that simple compression schemes show significant promise for reducing address bus width but only moderate benefits for data bus reduction.

Daniel Citron and Larry Rudolph in [5] show that if a data value is used, there is a high probability that closely related values will be used. They present a technique to increase bus width. A physically narrow bus is widened by using caching techniques. All information transferred across the bus (address, data and instructions) is divided into two. The low order part is sent straight over the bus and the high part is compacted into a smaller value. This compaction is done by storing the high order part in a BUS-EXPANDER [5]. By having corresponding BUS-EXPANDERS situated as an interface between various computing devices to the bus, it is possible to send indices into the BUS-EXPANDERS instead of the full value. This can almost double the effective bandwidth. They show that for almost all applications this enables sending

2n bits of information over an n bit wide bus. However, the main problem with this scheme seems to be with compacting floating point values.

Matthew Farrens and Arvin Park [4] explore another interesting idea. They shows that when address reference streams exhibit high degrees of spatial and temporal locality, many of the higher order address lines carry redundant information. It suggests that by caching higher order portions of address references in a set of dynamically allocated base registers, it would be possible to transmit small register indices between the processor and memory. Apart from other benefits, this scheme can minimize the information redundancy and potentially reduce the bandwidth constraints problem significantly. However, this scheme may introduce extra delay in the processor's critical path.

Given the above proposals, in this report we propose a new form of data compression, whose main advantage is that there is no decompression cost. We also propose a cache that can store this compressed data on the chip. This cache will be accessed for data that encounter a miss in the L2 cache. Thus, this scheme could provide multiple benefits. It could reduce the data traffic between the main memory and the CPU. It can improve the average miss latency by satisfying at least some L2 misses from the proposed cache. However, the cache has a fully associative structure. Searching for an entry in the cache depends on the number of entries. Hence it is essential to limit the size of this cache to a small value to make is practically useful. This report presents a limit study for this proposal and there by shows the maximum potential benefits of this scheme, given unlimited resources.

The rest of this paper is arranged as follows. Section 3.0 presents the details of our scheme. Section 4.0 presents the experimental setup and the changes made to the simulator to collect the results. Section 5.0 contains the results and its analysis. Finally, Section 6.0 has the conclusions and presents future work in this area.

2.0 Main Memory Data Characteristics for SpecInt2000 benchmarks

Previous work in the group [9], has shown that many store misses transfer uninitialized heap data. Intuitively we felt that it is possible that the data value in this heap might have all zeros. If this uninitialized data has all bytes with the same value then, potentially, this data can be compressed. Caching such data and storing it on the chip would result in removing the losses due to cache misses for these data.

The main idea of this proposal is to find data that has the same value in consecutive bytes. Such a consecutive stream of bytes in memory can be represented with the start address, length

and the value. We call such data "*value-range*" data. Before giving a formal definition for *value-range* it is important to define a related term called the "minimum length of *value-range* data". The *minimum length of value-range data* is defined by the user and can be any positive number. Let this quantity be defined by Lmin. For a given value of Lmin, *value-range* data is defined as data where at least Lmin consecutive bytes of memory have the same contents. Thus a *value-range* data can be defined with three numbers:

Start Address (SA)

Length (L), where L is the number of bytes having the same value and

L >= Lmin

Value (V), where V is the value in each byte of *value-range* data

Figure 1. shows examples of *value-range* data and their representation.

Let us consider 32 bytes of main memory. Let the minimum length of *value-range* be 4 bytes. For simplicity let us assume that the contents of a byte of memory can have either the value 0 or 1. Below we show the contents in the 32 bytes of memory and recognize the possible value ranges.

Address	A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15	A16
Content	0	1	0	0	0	0	1	1	1	1	1	1	0	1	1	1	0

Address	A17	A18	A19	A20	A21	A22	A23	A24	A25	A26	A27	A28	A29	A30	A31
Content	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0

From the definition of *value-range*, we know that, in this case if 4 or more consecutive bytes of memory have the same contents then it would be recognized as a *value-range* entry. Hence in the above example we have the following *value-range* entries.

1. SA = A2; L = 4; V = 0
2. SA = A6; L = 6; V = 1
3. SA = A16; L = 5; V = 0
4. SA = A23; L = 9; V = 0

Figure 1. Example of *value-range* data. Shows an example of 32 bytes of memory and recognizes the possible *value-range* data. The minimum length of *value-range* is set to be 4.

In order to recognize the potential for this scheme it important to find the frequency of occurrence of this phenomenon in main memory. Therefore, a static snapshot of memory was analyzed to recognize the number of *value-range* data available as any time. We wanted to find the distribution of *value-ranges* of all lengths. Hence, the entire memory was scanned to recognize *value-ranges* of all possible length. Figure 2., shows how value ranges of different lengths is distributed in the main memory for different SpecInt2000 benchmarks. For easier

representation lengths of *value-range* has been divided into powers of two. Therefore if we have 19 consecutive bytes with the same value we represent it as a value range of length 16 (2**4), i.e., the nearest power of 2. Since we want to place the *value-range* cache to satisfy misses from the L2 cache, we believe that potential benefits can be obtained only from *value-ranges* that are of length greater than or equal to the size of a L2 cache line. *Value-ranges* with length less than a L2 cache line cannot satisfy the L2 miss. Given this requirement, it seems reasonable to assume that *value-ranges* of length less than 32 bytes would not be useful for our purposes. From Figure 2. it is clear that, at a given point, all the SpecInt2000 benchmarks have more than 50% of its memory with data that have *value-range* characteristics with a length of 32 bytes or more. Thus if 50% of memory can be captured with *value-range* data we must be able to satisfy quite a few L2 misses from the *value-range* cache. Thus the presence of large number of *value-ranges* of length 32 bytes and higher encourage us to study the potential benefits that can be obtained by compressing and caching this form of data. Our proposed scheme is explained in the Section 3.

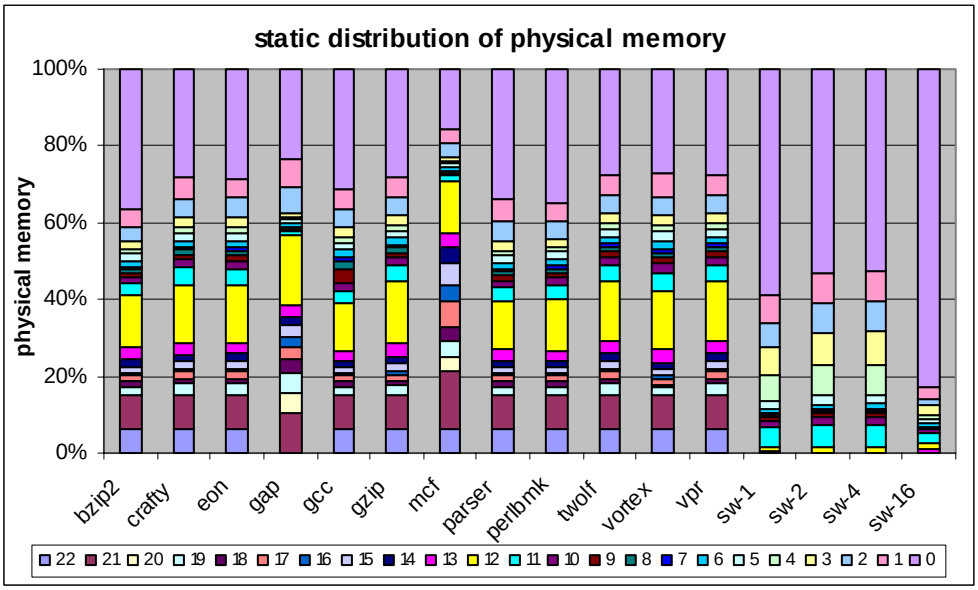


Figure 2: Value-range distribution in main memory: Shows the *value-range* data of different lengths available from the static snapshot of the main memory.

The four commercial benchmarks considered in Figure 2., have very few *value-ranges* with lengths greater than 32 bytes. For all these benchmarks such *value-ranges* seem to constitute only about 10% of the total memory.

3.0 Value-Range Cache Scheme

Section 2.0 shows that large parts of main memory have data that has *value-range* characteristics with length of 32 bytes or more. It is possible to represent the *value-range* data with its starting address, length and value. Thus, *value-range* data of length 1024 bytes starting at address A and containing the value B in each of the 1024 bytes can be represented as a set of three numbers (A,1024,B). Thus 1024 bytes of data in the main memory can be represented in a compressed form with three numbers that represent the starting address, length and the value. If this data is stored in a cache after the L2 then some misses in L2 cache could be satisfied from this cache. Thus our *value-range* cache is composed of entries where each entry has three parts, the starting address, length and the value in each byte of the value range. A miss from the L2 is checked against the *value-range* cache. If the *value-range* cache can satisfy this request then we avoid the delay incurred by accessing main memory.

L2 has a miss to the address X. SAi, Li and Vi represents the parameters of a *value-range* cache entry. The algorithm presented below is applied to all the entries of the *value-range* cache till there is a hit. If these conditions are not met by any entry in the *value-range* cache then we have a miss.

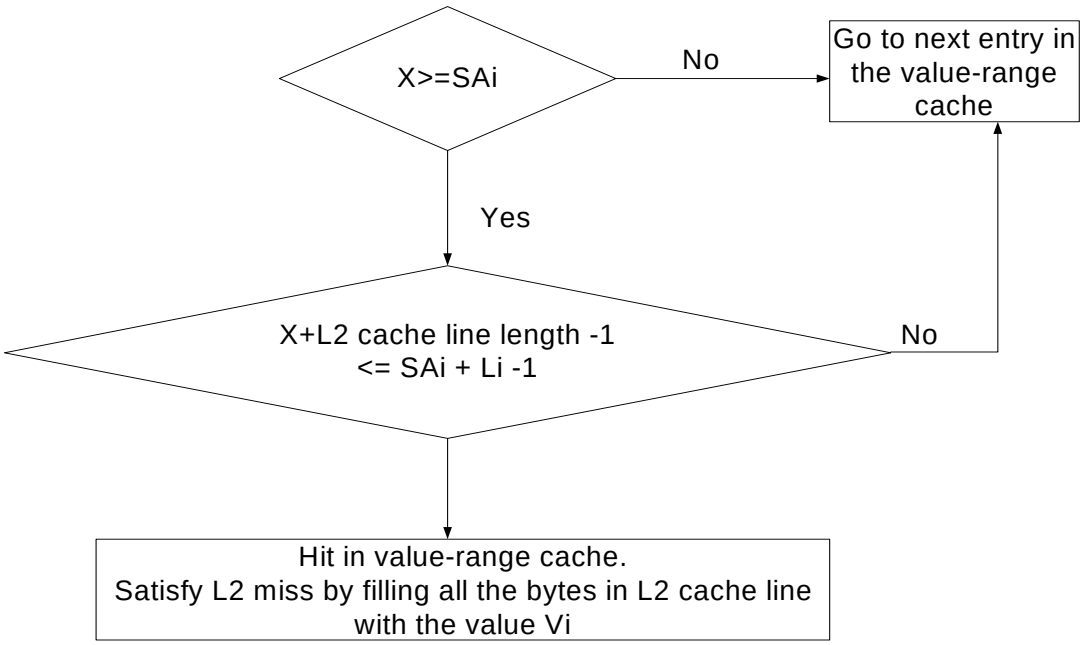


Figure 3. Shows the algorithm to search for an address in the *value-range* cache.

Figure 3. shows the algorithm used to search for an address in the *value-range* cache. Suppose the each entry of the *value-range* cache is represented as follows start address (SAi), length (Li) and value (Vi). The subscript i varies from 0 to the number of entries in the value range cache. Thus, if a L2 cache miss happens for address X the entire *value-range* cache is searched to find an entry that satisfies the following conditions.

For every i from 0 to the (number of entries in the value range cache –1) check if

1. X >= SAi
2. X + length of L2 cache line - 1 <= SAi + Li -1

If conditions 1 and 2 are satisfied for a value of i then satisfy the L2 miss by filling each byte of the L2 cache line with the value Vi. If conditions 1 and 2 are not satisfied then check these conditions for the next value of i. If the conditions 1 and 2 are not satisfied for any value of i then the required line of memory does not exist in the *value-range* cache and therefore the miss should be satisfied by the main memory.

The number of bytes required to store each entry of the *value-range* cache would be

No. of bytes required to store a start address +
No. of bytes required to store the length of the *value-range* +
No. of bytes required to stored the value.

Due to the way we define a *value-range* 1 byte is required to store the value. From, Figure 2., the maximum length of value range seems to be 2**22, hence at most 22 bits would be required to store the length. This means that the length and the value of an entry in the *value-range* cache need a maximum of 4 bytes. The total size of the *value-range* cache will depend of the number of entries present. Table 1. shows the amount of compression achieved for *value-range* entries of different lengths with our scheme. Table 1. does not provide compression ratios for *value-ranges* of length more than 1024 bytes. This is because *value-range* data of length more than 1Kbyte does not occur frequently. Even if we encounter such data it is easy to see that their compression ratio would be very small. Another important observation from Table 1. is that as the lengths of the *value-range* data increases, its compression ratio decreases. Hence longer *value-range* data is expressed in a more compressed form with our scheme.

If 32 bits are required to access a byte of memory, and if the minimum length for *value-range* is set to be 32 bytes, the maximum length of *value-range* can be represented in 3 bytes. Then each *value-range* entry can be represented in (4 (start address) + 3 (maximum length of *value-range)* + 1 (value)) bytes i.e., 8 bytes. Thus the compression achieved for value-ranges of various lengths is given below:

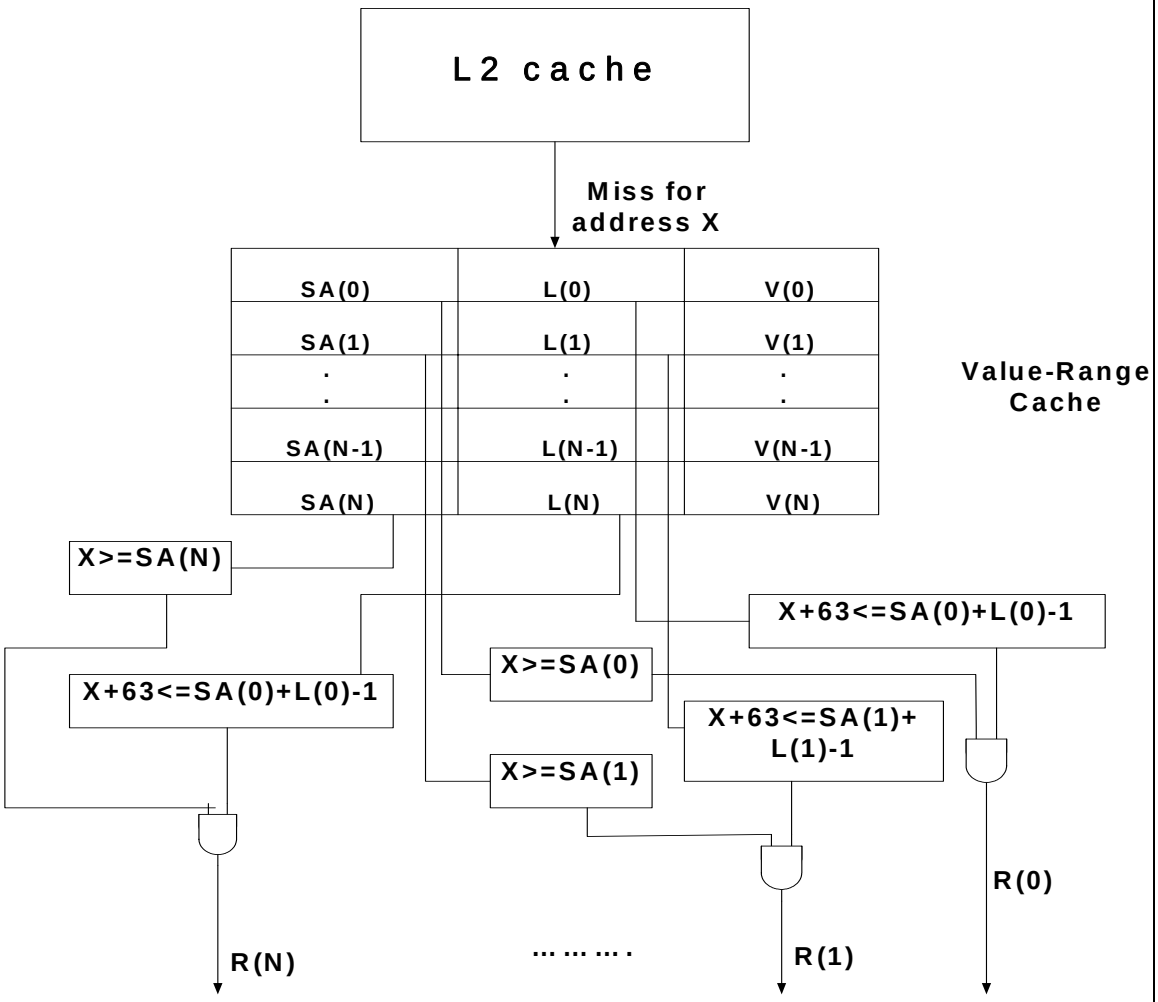
Value-range length (In bytes)	Compression ratio	*Value-range* length (In bytes)	Compression ratio
32	8/32 = 0.25	256	8/256 = 0.03125
64	8/64 = 0.125	512	8/512 = 0.015625
128	8/128= 0.0625	1024	8/1024 = 0.0078125

Compression ratio is defined as the number of bytes required to represent the data in *value-range* cache / number of bytes required to represent the data in the regular uncompressed form in memory.

Table 1. Shows the compression achieved by representing *value-range* data with our scheme.

Figure 3. Shows graphically the arrangement of entries in the *value-range* cache and the methodology for searching addresses in the value range cache. Figure 3. also explains how *the value-range* cache works with the L2 cache. It is obvious from Figure 3. that in order to get reasonable performance from the *value-range* cache it might be required to do parallel searches among all its entries to locate an address. Such parallel searches might not be possible if the number of entries in the *value-range* cache is very large. Hence it is important to keep the number of entries in the *value-range* cache to a small value.

Let us consider a *value-range* cache with N+1 entries. Each entry will consist of three parts the starting address (SA), length of the *value-range* (L) and the value of the *value-range* (V). The entries in the *value-range* cache are shown below. The number in the brackets shows the number of the entry. Also let the size of a L2 cache-line be 64 bytes.



If any R(i) is true, where i varies from 0 to N, then there is a hit in the value range cache. L2 request is satisfied by the *value-range* cache by copying the corresponding value, V(i), in all the 64 bytes of the L2 cache-line. If for no value of i, between 0 and N, R(i) is true, then there is miss in *value-range* cache also. This request must be then sent to the main memory.

Figure 3. Search methodology in the *value-range* cache.

4.0 Experimental Setup

For the purpose of our study we used the PHARMsim simulator developed in the Electrical and Computer Engineering department of the University of Wisconsin-Madison. PHARMsim is a Power-PC based simulation infrastructure using the SIMOS-PPC [11,13] and SimpleMP [12] simulators. PHARMsim is a detailed full-system simulator that supports all the instructions (system-mode and user-mode) in the PowerPC instruction-set architecture. The relevant machine parameters used for our study is provided in Table 2.

For our study we used the SpecInt2000 benchmark suite. All the benchmarks were run for 100 million instructions after they had completed the first 500 million instructions. This allows the results to be free of the distortions due to start up phase.

For studying the benefits of the *value-range* cache some important decisions need to be made. We would consider each of these questions in this section and explain the decisions that we made.

The first important question to consider is when should the memory be scanned to get information on the *value-ranges* present in memory. Also, it is important to decide how much memory should be scanned at one time. We decided that it is best to scan a memory page when it encounters the first TLB miss. This would enable us to capture all *value-range* data in the cache and check future references to data in this page with the entries in the *value-range* cache to access potential hits. Also, it was decided that at one time it is enough to capture all *value-ranges* in one memory page. A memory page was 4Kbytes long in our machine. Scanning more than one memory page at a time might be expensive and also not required. With our scheme we might only miss the first reference to a *value-range* entry in a page. The size of line in L2 cache was set to be 64bytes. Therefore, *value-ranges* of length less than 64 bytes were ignored. Also the *value-range* cache has to be updated with every write-back to memory so that its contents represent the correct data. The *value-range* cache has to be updated with DMA transactions to keep its contents consistent.

Because of the way PHARMsim is designed, incorporating a cache after L2 required complex changes to the coherence protocol. In particular a simpler protocol had to be designed for the *value-range* cache to make it consistent with the rest of the system. Because the PHARMsim emulates a multiprocessor system, maintaining consistent data in the *value-range* cache posed significant problems. We required a new set of states for the entries in the *value-range* cache that would remain consistent with various transactions in the memory system. An

attempt to build a *value-range* cache with these requirements has been unsuccessful and requires further correction. Since we were mainly interested in doing a limit study with the *value-range* cache and we were limiting our studies to uniprocessor systems that run the SpecInt2000 benchmarks we decided to approximate the above situation with an easier version. The model that was finally implemented is presented below.

The *value-range* cache implemented in PHARMsim currently performs as follows. Currently, the *value-range* cache is assumed to be of infinite size and can have as many entries as required. On a TLB miss a scan process is called that scans the corresponding physical memory page and finds all *value-ranges* of length more than and equal to 64 bytes. All such *value-ranges* are saved with their starting address, length and byte value in the *value-range* cache. When a L2 miss can be satisfied by an entry in the *value-range* cache it is sent back to the L2 cache with a latency of 1. Misses in the *value-range* cache are fetched from memory which takes 350 cycles; where 100 cycles is the latency required to access the DRAM and 250 cycles is the latency due to the network. Since the *value-range* cache is expected to be placed after the L2 cache it should ideally not encounter either of these delays. Updates to the contents of the main memory also affect the contents of the *value-range* cache. Thus memory-updates could result in adding new entries, removing the current entries or updating the existing entries. Updating the existing entries could involve one of the following. Changing the length of the value range, sometimes this could also mean changing the starting address. It could also involve changes the value in the *value-range* entry. Some updates might result in changes that combine two entries in the *value-range* cache into a single entry of longer length etc. It is important to maintain the *value-range* cache consistent so that at every instant of time it correctly represents the value ranges present in the memory-pages that have encountered a TLB miss as shown in Figure 1.

Out-of-order Execution	8-wide fetch/issue/commit, 128-entry RUU, 64-entry load/store queue, instruction fetch queue size: 16
Memory System	Trace buffer size: 100, L1 I$: 32KB(2-way, 64B line size), 1-cycle latency, L1 D$: 32KB(2-way; 64B line size), 1-cycle latency, L2 Unified: 256KB(4-way; 64B line size), 12-cycle latency, Main Memory: 100-cycle latency, Data n/w: 250-cycle latency
Functional Units	8 integer ALUs, 2 integer mult/div, 4 floating-pt ALUs, 4 floating-pt mult/div

Table 2. Machine configuration

With the *value-range* cache setup as detailed above we collected two sets of results. The first set examines the number of L2 misses that can be captured with the *value-range* cache. The second result gives the improvement in IPC because of the value range cache when compared to the base IPC.

5.0 Results and Discussion

Table 3. shows the total number of L2 misses in the base machine configuration and the number of these misses that can be captured in the *value-range* cache. Figure 4. shows the percentage of L2 misses captured by *value-range* cache.

Benchmark	Simulated Instructions	No. of L2 misses in the base machine	No. of L2 misses captured by the *value-range* cache
bzip2*	100 M	3263425	134030
crafty	100 M	62041	3884
eon	100 M	3879	49
gap	100 M	37936	60008
gcc	100 M	200519	5840
gzip	100 M	174051	6915
mcf	100 M	2088751	260
parser	100 M	408151	18184
perlbmk	100 M	58234	282
twolf	100 M	122564	9848
vortex	100 M	255215	12607
vpr	100 M	2389	60

Table 3. Number of L2 misses in the base machine and the number of L2 misses captured by *value-range* cache

*bzip shows the misses encountered by running the benchmark from the start to 600 million instructions.

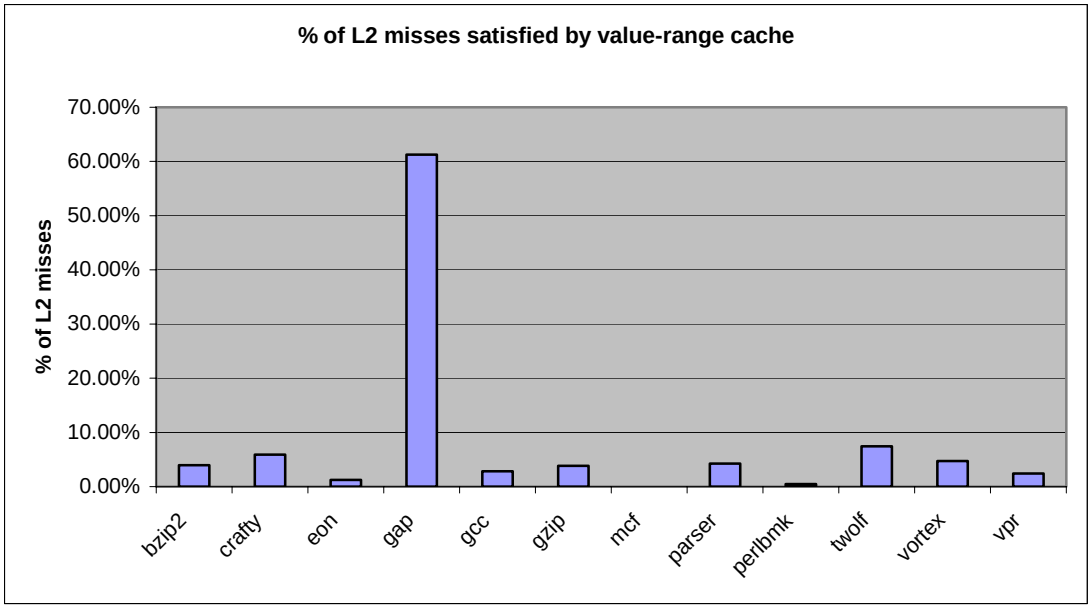


Figure 4. Percentage of L2 misses captured by the *value-range* cache

Figure 4., shows that most SpecInt2000 benchmarks can satisfy less than 5% of their L2 misses from the *value-range* cache. Only gap seems to be able satisfy nearly 61.27% of its L2 cache misses from the *value-range* cache. This result is surprising considering the fact that Figure 2. shows that nearly all benchmarks had about 50% of their memory with value-range data of 64 bytes or longer. Hence by extension it may be assumed that nearly 50% of all accessed memory-pages would have *value-range* data of length 64 bytes or longer. We would therefore expect to cover many more L2 misses with the *value-range* cache assuming fairly uniform access in the memory pages. However, this does not seem to be true. It could be due to a couple of factors. One could be that *value-range* data is not distributed uniformly over the entire memory. It is possible that main memory pages that are accessed by the L2 cache have much fewer *value-range* entries than the pages that were never accessed. Another explanation could be that most of the L2 accesses in a page are to those parts that have *value-range* data of lengths less than 64 bytes.

Table 4., shows the IPC of the base machine and the IPC after implementing the *value-range* cache. Figure 5., shows the percentage improvement in IPC due to *value-range* cache.
Figure 5., shows results as expected from Figure 4. In the given situation the only factor that contributes to improvement in IPC is the fact that L2 misses that have a hit in the *value-range* cache avoid the 350 cycles latency involved in accessing the main memory. Of all the SpecInt2000 benchmarks gap shows the maximum improvement in IPC. This can be attributed to the fact that 61.27% of the L2 misses do not encounter 350 cycles of delay due to *value-range*

cache. gap has a 23.64% improvement in IPC. All other benchmarks also behave as expected. Because of the small percentage of L2 misses that can be satisfied from the *value-range* cache almost all benchmarks show less than 3% improvement in IPC.

Benchmark	Simulated Instructions	Base IPC	IPC after implementing the *value-range* cache
bzip2	100 M	0.4365	0.4471
Crafty	100 M	1.4846	1.4985
Eon	100 M	1.7590	1.7592
Gap	100 M	0.9248	1.1434
Gcc	100 M	0.7924	0.8012
Gzip	100 M	0.8593	0.8745
Mcf	100 M	0.2018	0.2018
Parser	100 M	0.4806	0.4943
Perlbmk	100 M	1.3342	1.3347
Twolf	100 M	1.5338	1.5398
Vortex	100 M	0.7156	0.7294
Vpr	100 M	1.8583	1.8584

Table 4. Base IPC and IPC after *value-range* cache has been implemented

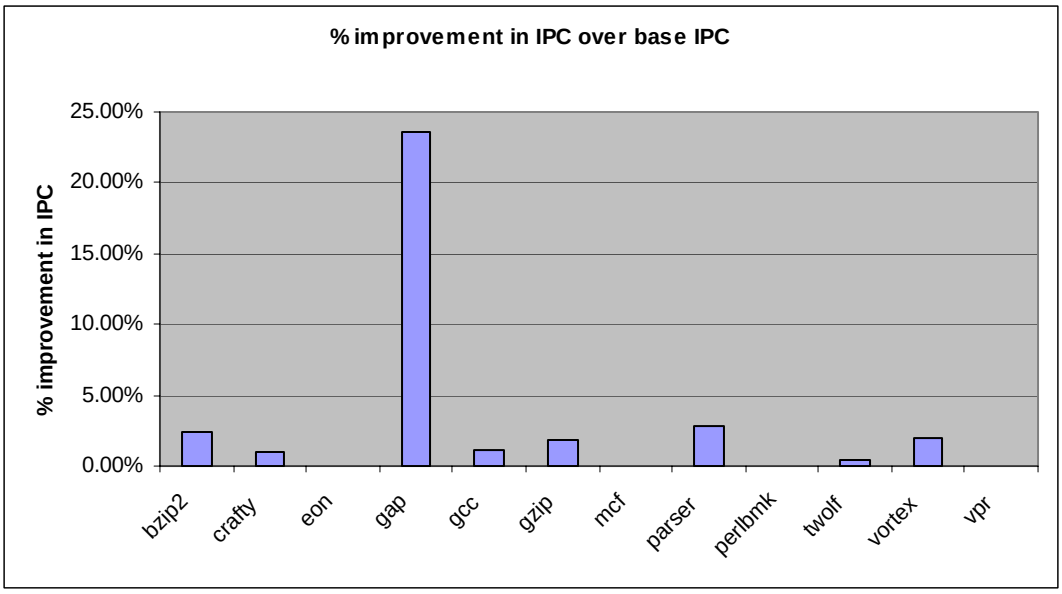


Figure 5. Percentage improvement in IPC from the base machine on implementing a *value-range* cache.

Although all benchmarks benefit to some extent from the *value-range* cache, the results are not very promising considering the fact that the *value-range* cache has infinite capacity and therefore can satisfy all L2 misses that has *value-range* property. This result might be indicative of the true nature of data accessed by these benchmarks and hence may present a fundamental limitation. However, we suspect that the results could be better if the benchmarks are run from the beginning and for a longer set of instructions.

The *value-range* cache does not come free. There would be some cycles lost in scanning a memory page and finding the *value-range* data present in it. Cycles would also be lost in transferring these data over the data network to where they would be stored after the L2 cache. According to the current design all entries in the *value-range* cache have to be searched before ensuring that there is a miss in the *value-range* cache. Given this condition it seems impractical to consider *value-range* caches with more than 1k values. Because searching in caches having more than 1k entries may reduce the benefits by incurring many cycles to search the entire cache. Also, a larger value range cache would imply more area.

However, it might be useful to access the benefits from the *value-range* cache with a limited number of entries. A previous study of a similar nature conducted in our group indicated that for most benchmarks about 3% of all *value-range* entries satisfy about 50% of all the L2 misses. It further showed that about 90% of all misses could be satisfied with 15% of all *value-range*

entries. Because these results were not taken in the experimental conditions used for this report they are not presented here. Nevertheless, they are important results and encourage us to explore if limiting the size of the *value-range* cache to a small value can give us almost the same benefit as with an infinite cache. This study might also be extended to find how accessing patterns in *value-range* data is related to its various characteristics e.g., length. For example, most hits in the value-range cache might be to entries that have longer lengths. Another study might be to find how the different accesses to memory are spatially related. It might be possible that after the first reference to data in the main memory all future references are to addresses with greater value. Hence *value-ranges* in those addresses are more important than others. If any such analysis enables us to design a *value-range* cache with very few entries that gives nearly as much benefit as the *value-range* cache of infinite capacity then this idea might be applicable in practice.

Another factor is that the definition of *value-range* is very rigid. It requires every byte of the L2 cache-line to have the same contents. We thought this might be the case as there are many uninitialized misses to the heap [9] and they might have all zeros. However, this assumption does not seem to provide us with the desired benefits. Therefore, alternative compression schemes can be considered. Presently the contents of the entire cache line is being represented by a single byte, therefore we require all the bytes in the cache line to have the same contents. However, since integers are usually represented in 2 bytes our case cannot capture integers with low non-zero values, as every other byte would have a non-zero value. Such values might occur often, as indicated by [1]. It might be useful to look into this aspect too. However one obvious disadvantage of this scheme would be that we would be using more bytes to represent a *value-range* data. But if lengths of *value-range* data are quite large then increase in one byte for the representation should not affect the efficiency of the cache too much.

6.0 Results and Conclusion

A major part of my research in this project was spent in learning to work with PHARMsim. PHARMsim in its current form is a complex system and one has to be very careful before incorporating changes to it. However, we justified the use of PHARMsim for our case because it would give us more realistic results. I spent nearly two semesters trying to understand PHARMsim and incorporating the *value-range* cache after the L2 cache. Since PHARMsim is a multiprocessor simulator, I was required to implement the *value-range* cache with a protocol that would let its data be consistent. Even though we were using the simulator with one processor it required the cache to work well with multiprocessor system. Hence the *value-range* cache had to

provide ideal responses for requests that might occur on the bus. In particular the DMA request was critical, as it would affect the correctness of the program execution in our case. In spite of trying of cover all issues that might arise, results collected later showed that because of *value-range* cache the benchmarks were executing a different set of instructions from their original set. Therefore, to avoid these complications we decided to design the *value-range* cache at the memory level and send data satisfied by it at a latency of 1 to the L2 cache. The results from this implementation of *value-range* cache are provided below.

On the first TLB miss to a page the entire page is scanned to find *value-range* data of more than 64 bytes. All such data is captured with three parameters: start address, length and value, in the *value-range* cache. All L2 misses are first checked with the *value-range* cache. If there is a hit in the *value-range* cache then this data is sent to L2 with a latency of 1 cycle. Data that misses the L2 cache encounters 100 cycles latency in accessing the DRAM and 250 cycles latency in the data network. Currently our cache has infinite capacity. We found that this cache is able to satisfy 2-7% of the L2 misses for most benchmarks. Because the data from the *value-range* cache is satisfied much faster than from the main memory this results in IPC benefits for most benchmarks. However the IPC benefits do not seem to be significant considering the fact that the value range cache has infinite capacity. This report, however, provides only the limit study. It might to be useful to see if the performance decreases significantly when the size of the *value-range* cache is limited. If this is not the case, it might still be a useful idea to implement, considering the fact that no benchmark shows a decrease in IPC.

References

[1] M. Kjelso, M. Gooch, S. Jones, “Empirical study of memory-data: Characteristics and Compressibility”, IEEE Proceedings on Computers and Digital Techniques, Vol 145, No 1, Jan. 1998, pp. 63-67.

[2] Jang-Soo Lee, Won-Kee Hong, and Shin-Dug Kim, “Design and Evaluation of a Selective Compressed Memory System”, Proceedings of International Conference on Computer Design, Oct. 1999, pp. 184-191.

[3] Krishna Kant and Ravi Iyer, “Compressibility Characteristics of Address / Data Transfers in Commercial Workloads”, Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads, Feb. 2002.

[4] Matthew Farrens, Arvin Park, “Dynamic Base Register Caching: A Technique for Reducing Address Bus Width”, Proc. Of 18th annual Intl. symposium on computer architecture, May 1991, pp 128-137.

[5] Daniel Citron, Larry Rudolph, “Creating a Wider Bus Using Caching Techniques”, Proc of First Intl. symposium on high performance computer architecture, Jan 1995, pp 90-99.

[6] D. W. Clark and C. C. Green, “An empirical study of list structure in Lisp”, Commun. ACM, 1977, 20, (2), pp. 78-87.

[7] M. Kjelso, M. Gooch and S. Jones, “Design and performance of a main memory hardware

data compressor", Proceedings of 22nd Euromicro conference, September 1996, (IEEE Computer Society Press), pp. 423-430.
[8] T. Welch, "A technique for high-performance data compression", IEEE Comput., June 1984, pp. 8-18.
[9] Jarrod A. Lewis, Bryan Black and Mikko Lipasti, "Avoiding Initialization Misses to the Heap", Proceedings of 29th International Symposium on Computer Architecture (ISCA-29), May 2002.
[10] Harold W. Cain, Kevin M. Lepak, Brandon A. Schwartz, and Mikko H. Lipasti, "Precise and Accurate Processor Simulator", Workshop on Computer Architecture Evaluation using Commercial Workloads, in conjunction with HPCA, February, 2002.
[11] Mendel Rosenblum. Simos full system simulator. http://simos.stanford.edu.
[12] Ravi Rajwar and Jim Goodman. Simplemp multiprocessor simulator. Personal communication., 2000.
[13] Tom Keller, Ann Marie Maynard, Rick Simpson, and Pat Bohrer. Simos-ppc full system simulator. http://www.cs.utexas.edu/users/cart/simos.
[14] C.C., Foster and R. Gonter, "Conditional interpretation of operation codes", IEEE Trans. Comput., 1971, 20, (1), pp. 104-107
[15] S. Segars, K. Clarke and L. Groudge, "Embedded control problems, thumb, and the ARM7TDMI", IEEE Micro, October 1995, pp 22-30.
[16] A.W. Appel and K. Li, "Virtual memory primitives for user programs", Proceedings of ASPLOS IV, 1991, pp. 96-107.
[17] F. Douglis, "The compression cache: Using on-line compression to extend physical memory", Proceedings Usenix winter technical conference, 1993, pp. 519-529.
[18] M. Kjelso, M. Gooch and S. Jones, "Design and performance of a main memory hardware data compressor", Proceedings of 22nd Euromicro conference, September 1996, (IEEE Computer Society Press), pp. 423-430.
[19] C. Lee and R. Yang, "High-throughput data compressor designs using content addressable memory", IEE Pro.-Circuits Devices Syst., 1995, 142, (1), pp. 69-73.
[20] J. Cheng and L. Duyanovich, "Fast and highly reliable IBMLZ1 compression chip algorithm for storage", Hot Chips VII, August 1995, pp.155-165.
[21] B. Abali, H. Franke, S. Xiaowei, et.al., "Performance of hardware compressed main memory", The Seventh International Symposium on High-Performance Computer Architecture, 2001, pp. 73-81.
[22] R. B. Tremaine, T.B.Smith, et. al., "Pinnacle: IBM MXT in a memory controller chip", IEEE Micro, March-April 2001, pp. 56-68.
[23] "SPECweb99 Design Document", available online on the SPEC website at http://www.specbench.org/osg/web99/docs/whitepaper.html.
[24] Transaction Processing Performance Council, TPCBENCHMARKTM C Standard Specification, http://www.tpc.org/, Jan 2000.