Value Compression to Reduce Power in Data Caches

Carles Aliagas\textsuperscript{a}, Carlos Molina\textsuperscript{a}, Montse García\textsuperscript{b}, Jordi Tubella\textsuperscript{d} and Antonio González\textsuperscript{a}

\textsuperscript{a} Dept. d'Enginyeria Informàtica i Matemàtiques
Universitat Rovira i Virgili
Av. Països Catalans, 26
43007-Tarragona (SPAIN)
E-mail: [callagas,cmolina,mgarcia]@ctbc.urv.es

\textsuperscript{b} Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Campus Nord - c/ Jordi Girona, l-3 - Mòdul D6
08034 - Barcelona (SPAIN)
E-mail: [antonio,jordi]@ac.upc.es

Abstract. Cache memory represents an important percentage of the total energy consumption of today's processors. This paper proposes a novel cache design based on data compression to reduce the power dissipation of the data cache. The new scheme stores the same amount of information as a conventional cache but in a smaller physical storage. At the same time, the cache latency is preserved, thus no performance penalty is introduced. The benefits in terms of power reduction vary from 15.5% to 16%, and the reduction in die area ranges from 23% to 40%.

1. Introduction

Cache occupy a very important part of die area in most processors. Percentages of around 50% are relatively common. On the other hand, some authors [2] [6] have reported that caches may be responsible for 10% to 20% of total power dissipated by a processor.

The objectives of this work are twofold: to reduce the area of the cache and its power dissipation while maintaining the same amount of information. We present a novel cache architecture that provides significant advantages in terms of both these metrics.

The key idea is based on the observation that many values stored in the data cache can be represented with a small number of bits instead of the full 32-bit or 64-bit common representation used by most values. Using less bits to store the data, together with a simple encoding/decoding scheme, allows the processor to reduce the storage required for a given amount of data, and thus, reduces the area and power dissipation of the memory.

The rest of the paper is organized as follows. Section 2 discusses and evaluates schemes for compressing data values. The proposed cache architecture is presented in section 3 and evaluated in section 4. Section 5 outlines some related work and finally, section 6 summarizes the main conclusions of this work.

2. Data Value Compression

Data values have significant redundancy in their representation. In this section we quantify this phenomenon and discuss some different types of data compression that can be used to take advantage of this redundancy.

In general, we focus on 64-bit wide data values. This allows for a huge range of values to be represented. However, the significant bits of these values are relatively few [10]. Non-significant bits can be avoided by using two types of data compression: integer data compression and address data compression.

Integer data compression is applied when the data can be represented with a few bits and the rest of the bits are non-significant (all ones or zeros). For a 64-bit wide word
there are several formats of compression depending on where the significant bits are. Figure 1 shows the different types of data compression applied. We will only store in the cache the black part (significant bits). The arrow indicates sign extension and the “0..0” means that the field is all zeros.

Another type of data are addresses stored in memory. In this case, we apply address data compression. There are three sources of addresses: data pointers, code pointers and stack pointers. The compiler distributes the memory logical space in three major components: code, data and stack. For our binaries (see section 4), the following address distribution is used: 0x0140000000 for data addresses, 0x0120000000 for code addresses and 0x011FF0000 for stack addresses. These addresses are 64-bit wide, but only a part of their bits is variable. The rest of bits remain unaltered and are common to all addresses. We try to recognize the pattern of the common part and store only the variable part. Figure 2 depicts the components of the address data compression. We have experimentally observed that with 8 patterns we can represent 83% of total value addresses. These patterns are: from data addresses 0x1400, 0x1401, 0x1402, 0x1403, from code addresses 0x1200, 0x1201, 0x1202 and from stack addresses 0x11FF.

In order to evaluate the contribution of the 7 types of data compression described above (6 integer data compression plus the address data compression), the behavior of a 16KB data cache is analyzed for a set of benchmarks. The combined effects of these compression schemes is a reduction of 64 to 22 bits. Detailed information about the experimental framework is presented in Section 4. Figure 3 shows the effectiveness of each type of scheme. The dark grey bars indicate the size reduction of each type of compression averaged for all benchmarks. Light grey bar shows the best percentage observed for a individual benchmark. Last bar is the contribution of all types of compression. Note that close to a 70% of the values that enter the data cache may be compressed. Two types of compression have a major impact: type 1 integer data compression (close to a 45%) and address data compression (close to 20%). Note also that a particular type of compression that on average does not represent significant benefit, may be important for a specific benchmark.
3. Pattern Cache

The underlying concept of our proposal is based on the observation that a significant part of the data cache values may be compressed, as explained in the previous section. In this way, a new cache architecture that we refer to as Pattern Cache is proposed in order to reduce the storage area and power dissipation. Figure 4 depicts the proposed design.

The Pattern Cache is divided into two areas: the tag and the data area. The tag area or Location Table (LT) stores compressed values and pointers to the data area or Value Table (VT). LT is indexed as a conventional data cache. Each LT entry has two fields: a tag that identifies the memory address stored in each line, as in a usual cache, and the location field that determines for each 64-bit word in the line where and how to find its value. Each location field has two parts: type and compressed data. The first one determines how to find the value and the second one is a compressed data or a pointer to VT. On the other hand, a VT entry has a full 64-bit data value.

There are three ways of obtaining the data values: a) a value may be stored in the LT entry itself using an integer compressed form; b) a value may be stored partly in the LT entry and partly in the Address Table (AT) using an address compressed form; and c) the value may be stored in the Value Table (VT) in a non-compressed form. A subset of the location bits determines which of the above schemes is used, as detailed in Table 1. The rest of the bits are either a pointer or a compressed value.

<table>
<thead>
<tr>
<th>Type</th>
<th>Information</th>
<th>Type</th>
<th>Information</th>
<th>Type</th>
<th>Information</th>
<th>Type</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Pointer</td>
<td>0100</td>
<td>Data: type 4</td>
<td>1000</td>
<td>Address: prefix 0</td>
<td>1100</td>
<td>Address: prefix 4</td>
</tr>
<tr>
<td>0001</td>
<td>Data: type 1</td>
<td>0101</td>
<td>Data: type 5</td>
<td>1001</td>
<td>Address: prefix 1</td>
<td>1101</td>
<td>Address: prefix 5</td>
</tr>
<tr>
<td>0010</td>
<td>Data: type 2</td>
<td>0110</td>
<td>Data: type 6</td>
<td>1010</td>
<td>Address: prefix 2</td>
<td>1110</td>
<td>Address: prefix 6</td>
</tr>
<tr>
<td>0011</td>
<td>Data: type 3</td>
<td>0111</td>
<td>Not used</td>
<td>1011</td>
<td>Address: prefix 3</td>
<td>1111</td>
<td>Address: prefix 7</td>
</tr>
</tbody>
</table>

**Table 1. Compression Type Codification**

The behavior of the Pattern Cache may be summarized as follows:

1. Miss on Read or Write: the new line is brought from the upper level of the memory hierarchy. At this time, a compression test is performed to all its 64-bit words. Values that are found to be compressible are stored in a compressed form in the location field of LT. Otherwise, the value is tried to be stored in an empty entry of the VT.

The search for room in VT is simple. An associated bitmask with the same number of bits as the VT determines which entries of the VT are empty. When a value is inserted in the VT, its associated bitmask entry is set. On the other hand,
when the LT entry that points to that value is replaced, its associated bitmask entry is cleared.

A key issue of the Pattern Cache is that LT contains only lines whose all words can be either compressed or allocated in the VT. If a new line has some values that cannot be compressed and there are no empty entries for them in the VT, the line is stored in a fully-associative buffer that assists the Pattern Cache. We refer to this buffer as the Assisting Buffer (AB in Figure 4). The AB and the LT are searched in parallel for each memory reference. Note that LT and AB always contain different cache lines. In addition to store lines that cannot be stored in the LT, the AB also works as a victim cache for lines replaced from the LT [7].

2. Hit on read: if there is a hit in LT, the location field provides either the value in a compressed form or a pointer to the VT where the value will be found. If the value is compressed it is necessary to restore the full value. With Integer Data Compression (Figure 1) only sign extension and zero concatenation is necessary. With Address Data Compression three bits of the field type (Figure 4) identifies the pattern of the compressed data. It is necessary to access the Address Table (AT in Figure 4) to obtain the pattern to restore the full value.

3. Hit on write: there are four cases depending of the location of the replaced value and if the new value can be compressed or not. a) If the replaced value was in the VT and the new value cannot be compressed, the latter is stored in the location of the former. b) If the replaced value was in the VT and the new value can be compressed, the later is stored in the LT and the VT entry of the old value is invalidated. c) If the replaced value was in the LT and the new value can be compressed, the later is stored in the location of the former. d) If the replaced value was in the LT and the new value cannot be compressed, the value is tried to be stored in an empty entry of the VT. If there is no space in the VT, all the line is moved to the AB and the entries in the LT and the VT are invalidated.

The compression test requires a sequence of bit comparisons in order to determine the pattern. For the integer type compression, an AND gate to detect a set of ones and an OR gate to detect a set of zeros is required. For address data compression, an AND gate with some entries inverted is needed for each different pattern. We consider those gates and the AT table (8 entries of 13 bits) negligible in terms of area and power dissipation.

Our proposal tries to reduce the storage area required to maintain the same information as the baseline cache. The following section analyzes how this reduction affects the die area, access time and power consumption. Then, it is a dynamic study of the miss ration and power dissipation for a set of benchmarks is presented.

4. Performance Evaluation

4.1. Static Analysis

This subsection presents an evaluation of the cache area, access time and power consumption of the Pattern Cache. These evaluations are done by means of the CACTI tool version 3.0 [9]. The input parameters of the CACTI tool have been adapted to model the structure of the Pattern Cache. The assumed technology is 0.09µm.

CACTI is used to estimate the area of both the LT and the VT. The total area of the Pattern Cache is the sum of both. The LT has the same number of lines and associativity as the base cache. Note that the LT and the base cache have the same tag area, but the data area of the LT only stores the location field (pointers and compressed values). The location field has 22 bits for each word in a line. On the other hand, the VT is managed as
Figure 5. (a) Die area (cm²), (b) Access time (ns) and (c) Power dissipation (nJ) for 16KB a direct mapped cache with 8-byte line size, instead of the 32 bytes of the base cache. Finally, the AB a fully-associative cache with 16 lines. The baseline cache is a direct-mapped cache with a Victim Cache of 16 entries.

Figure 5.a shows the total die area of different cache configurations for a 16KB baseline data cache. Several Pattern Cache configurations are depicted. VTxx means a Pattern Cache with a VT capacity reduced to xx% of the baseline size (e.g., VT30 means a VT capacity reduced to 30%). Different colors in each bar represent the contribution of tags, location field and data. The rightmost bar represents the area of the Assisting Buffer (AB) that has to be added to all cache configurations, including the baseline.

As expected, the achieved area reduction is significant in all cases. For instance, VT30 achieves a reduction of 26% with respect to the baseline cache. Below, it is discussed how these reductions interact with the miss ratio. Statistics for other cache capacities ranging from 4KB to 64 KB are detailed in Table 2.

The access time of the various cache architectures was also evaluated with the CACTI tool. Figure 5.b shows the results for a 16KB baseline and related Pattern Cache architectures. Notice that, every cache configuration has three bars. First bar determines the time required for tag check. Second bar determines the time required to access the value (in a compressed form o through VT). Finally, last bar determines the access time to the Assisting Buffer or Victim Cache. In order to determine the total access time, the maximum of those three bars has to be considered. The main conclusion of this study is that the Assisting Buffer determines the access time. This is only true for caches of less than 128KB.

Finally, power dissipation has been evaluated in a similar way as die area and access time (only dynamic power is considered in this study). Since the power dissipation of memory structures depends on the area of the memory and the total number of bits read/written, the Pattern Cache provides significant benefits in terms of power reduction. Figure 5.c shows the power consumption for a cache hit. In particular, the Hit pattern corresponds to the power of an access in the Pattern Cache that hits in LT. Hit VT corresponds to the power dissipated by the Pattern Cache when the value is obtained from VT. This involves an access to LT plus an access to VT. Hit is the average consumption of the Pattern Cache considering the percentage of Hit pattern and Hit VT during the execution of programs. Simulations have shown that on average, 75% of the time the value is obtained from LT. Detailed savings for caches ranging from 4KB to
64KB are presented in Table 2. Note that the reduction in power is very significant across the whole range of capacities.

4.2 Dynamic Analysis

The simulation environment is built on top of the Simplescalar [4] Alpha toolkit that has been modified to model the Pattern Cache.

The following benchmarks Spec2000 have been considered: crafty, eon, gcc, gzip, mef, parser, twolf, vortex and vpr from the integer suite; and ammp, apsi, art, equake, mesa, mgrid, sixtrack, swim and wupwise from the FP suite. The programs have been compiled with the Compaq C compiler with -non_shared -O5 optimization flags (i.e., maximum optimization level). Each program was run with the reference input set and statistics were collected for 1000 million of instructions after skipping the initial part of initializations.

4.3. Analysis of Results

This subsection presents and analyzes the results obtained for different memory level hierarchy configurations in terms of die area, power consumption and miss ratio.

4.3.1. Miss Rate vs Power Consumption

In order to determine the performance benefits of the cache design proposed in this paper, the following scenario is considered: a first level data cache ranging from 4KB to 64KB. All baseline cache configurations are considered to be direct-mapped with a cache line of 32 bytes plus a Victim Cache fully-associative of 16 entries. To better understand the performance of our proposal, different Pattern Cache configurations are also evaluated. The LT is dimensioned to store the same number of lines as the baseline configurations, and the VT size is 40%, 30% or 20% of the data array of the baseline configuration. For all the configurations of the Pattern Cache, an Assisting Buffer of 16 entries is considered.

Figure 6 shows the power dissipation of the memory hierarchy. Each bar denotes the average power for different cache configurations, which is obtained through the following formula:

\[
Power = (\text{Hit} \times DLI\_Power) + (\text{Miss} \times (2 \times DLI\_Power + DL2\_Power))
\]

DLI\_Power is the average power of an access to the first level data cache. DL2\_Power is the power of an access to the second level cache. This cache is assumed to be 512KB. Note that this expression computes the total power by considering the power
of the first level data cache on hit, and the power of the first and the second level data cache on miss. A miss produces two accesses to DL1 (one for detecting the miss, and another for storing the values) and one access to DL2 to obtain the data.

As expected, the Pattern Cache outperforms the base model in terms of power consumption. The main contribution to this reduction is due to the LT, which provides the data in 75% of the hits. Note also that the VT reduction hardly affects power dissipation since decreasing the VT size increases miss ratio and more data must be provided by the second level cache.

From these numbers and the results in the previous section, we can conclude that the Pattern Cache is an effective architecture for first level data caches in terms of power, die area and access time. Table 2 summarizes all the statistics for each cache configuration being considered. For instance, a reduction of VT to 40% produces an average die area reduction of 19%, a power dissipation reduction of 16% and a very minor (1.5%) increase in the miss ratio.

5. Related Work

Zhang et al. [13] introduced frequent value locality. They observed that a few values represent a large fraction of memory accesses. Based on this observation, they propose the design of the FVC (Frequent Value Cache). The FVC only contains frequently accessed values stored in a compact encoded form and it is used in conjunction of a traditional Direct-Mapped Cache. Yang et al [12] present a similar cache design and evaluation called CC (Compression Cache) where each line can hold one uncompressed line or two cache lines that have been compressed to at least half of their lengths. A modification of the FVC is proposed in [11] in order to improve energy efficiency.

In our proposed architecture, the data array is portioned into two arrays in such a way that only the first data array is accessed for compressed values, which represent the majority of accesses.

Significance compression is used by Brooks et al. [3] and Canal et al. [5] to reduce power dissipation, not only in data cache but in the full pipeline. Brooks et al. present a hardware mechanism that dynamically recognizes whether the most significant bits of values are just sign extension and in this case, the functional units operate just on the least significant bits. Canal et al. propose a new significance compression scheme that appends two or three extension bits to each data value to indicate the significant byte positions. They propose a reorganization of the whole pipeline to take advantage of narrow operands in all pipeline stages. Other different data compression schemes are also
presented in [8] and [10]. The former explores different encoding schemes oriented to a large number of values, and dynamically adapts the encoding according to the changes in the frequency distribution of data values. The latter proposes a dynamic zero compression technique to reduce cache energy by taking advantage of the high occurrence of zero-valued bytes in the cache. Black et al [1] introduce a mechanism called block-based trace cache based on pointers to reduce replication in the trace cache.

6. Conclusions

We have introduced a novel data cache architecture called Pattern Cache. The proposed cache applies different types of compression to values in order to reduce the required storage. Simulation results show that in a data cache, close to 70% of the values of 64-bit ISA programs may be compressed to 22 bits. We have shown that the Pattern Cache significantly reduces power dissipation and die area with a minimum loss in terms of miss ratio while maintaining the same access time as a conventional cache.

7. Acknowledgments

This work has been supported by projects CYCIT 511/98 and ESPRIT 24942. The research described in this paper has been developed using the resources of the European Center for Parallelism of Barcelona (CEPBA) and the resources of the Robotics and Vision Group of Tarragona (RIVI).

References


