# Value Compression for Efficient Computation

Ramon Canal[1], Antonio González[12] and James E. Smith[3]

[1]Dept of Computer Architecture, Universitat Politècnica de Catalunya
Cr. Jordi Girona, 1-3, 08034 Barcelona, Spain
{rcanal, antonio}@ac.upc.edu
[2]Intel Barcelona Research Center, Intel Labs-Universitat Politècnica de Catalunya
Cr. Jordi Girona, 27-29, 08034 Barcelona, Spain
antonio.gonzalez@intel.com
[3] Dept. of Electrical & Computing. Engineering, University of Wisconsin-Madison
1415 Engineering Drive, 53706 Madison-WI, USA
jes@ece.wisc.edu

**Abstract.** A processor's energy consumption can be reduced by compressing values (data and addresses) that flow through a processor pipeline and gating off portions of data path elements that would otherwise be used for computing non-significant bits. An approach for compressing all values running through a processor is proposed and evaluated. For the SpecInt2000 benchmarks the best compression method achieves energy savings of more than 20 percent and a peak power reduction of 18 percent.

## 1 Introduction

In recent years, energy consumption has become a critical design constraint in microprocessor design and will likely remain so well into the future. Energy is important not only because of battery-life related issues, but also because of heat dissipation and thermal constraints. In current CMOS technology, most energy consumption occurs during state transitions in the underlying circuits [3]. This dynamic energy consumption is proportional to switching activity, as well as load capacitance and the square of the supply voltage. Thus, an important energy conservation technique is to reduce switching activity by "gating off" or inhibiting switching in portions of logic and memory during clock cycles when they are not being used.

In addition, the importance of static energy consumption is rapidly increasing with each microprocessor generation and will soon become as important as dynamic energy consumption. To reduce static energy consumption important techniques include minimizing circuit complexity and powering-down components that are not in use.

Value compression is a mechanism that is in a sense orthogonal to the more commonly used schemes that gate off or power off entire subsystems. With value compression the effective width of a subsystem is narrowed by turning off only certain bit (or byte) positions –usually higher order bytes, while leaving logic corresponding to the other bit (or byte) positions turned on. Value compression works because many values do not require the full precision supported by the data path. For example, the integer value *one* commonly occurs, but clearly does not require 32 (or 64) bits to encode it. Consequently, some value can be stored or manipulated in *compressed* form. For storage, value compression can be applied to individual data items, and for arithmetic and logical operations it is typically applied to both input operands. In either case, only a portion of storage or logic is required and energy is saved by turning off the unused portion(s).

In this paper we analyze several value compression mechanisms that are applied to the entire datapath. The paper is organized as follows. Section 2 describes the general principles and implications behind value compression. Section 3 lists related work. In Section 4, a comparison of several value compression schemes is performed. Finally, the main conclusions are presented in Section 5.

## 2 General Principles

As the name suggests, value compression reduces the number of bits used for representing a given value. When using value compression, data is typically represented with a number of data bits, plus some extra format bits that indicate the specifics of the compression method used. To date, most work has focused on compression of non-floating point data; extensions to floating point awaits further research.

Value compression can be used in several structures that make up a processor's datapath. These include data and instruction caches, integer functional units, register files, and branch predictors. Fig 1 contains data that indicates the compressibility of data values read/written in registers as SpecInt 2000 benchmarks are run on a 64-bit Alpha processor. This distribution shows a large potential for the value compression mechanisms because a large percentage of the values are narrow. For example, 40% can be represented in one byte (are between -128 and 127). The peak at 5 bytes is due to the memory addresses which are typically 5 bytes long in the Alpha architecture.

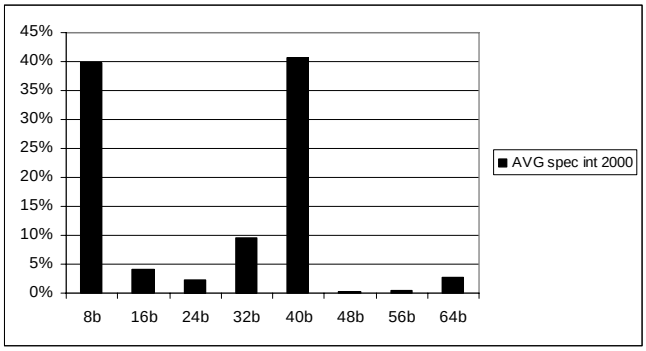


**Fig 1.** Data size distribution for the SpecInt2000

A good value compression method must take advantage of this data distribution, and, at the same time, incur a low overhead when compressing and decompressing. Although value compression can help reduce the energy consumption for performing certain functions, it is important that the overhead of compressing and decompressing does not affect the overall performance and the energy consumption. Thus, a good compression scheme should strike a good balance between the compressibility of the values and the extra performance and energy costs of the mechanism.

Researchers have proposed three basic methods for value compression. The first, *size compression*, was suggested in the preceding paragraph and compresses values according to their size (i.e., the minimum number of bytes in 2's complement notation) [1][8][9][10][11]. With size compression, one or more format bit(s) indicate the number of significant bytes. The second mechanism uses one format bit per byte to indicate whether the byte is zero or not [12]. This method, *zero compression*, can take advantage of zero bytes in any position, not just in high order positions as with size

compression. The last mechanism, *significance compression*, uses one format bit per byte to indicate whether a byte is a sign-extension of the previous one [4], and the least significant byte is always uncompressed.

The following table includes several value configuration formats that we consider in this paper. Other configurations have been analyzed and give significant smaller performance.

| Value compression method | Classification of the values | Extra bits |
|---|---|---|
| Size 8-64 | 8 bits or 64 bits | 1 |
| Size 16-64 | 16 bits or 64 bits | 1 |
| Size 32-64 | 32 bits or 64 bits | 1 |
| Size 40-64 | 40 bits or 64 bits | 1 |
| Size 8-16-32-64 | 8 bits, 16 bits, 32 bits or 64 bits | 2 |
| Size 8-16-40-64 | 8 bits, 16 bits, 40 bits or 64 bits | 2 |
| Significance 8-16-24-32-40-64 | Bytes 2,3,4,5 sign extended one byte, or byte 6 extended by two bytes. | 5 |
| Significance 8-16-24-32-40-48-56-64 | Bytes 2,3,4,5,6,7,8 sign extended one byte | 7 |
| Zero 8-16-24-32-40-64 | Bytes 2,3,4,5 can be zero or bytes 6 through 8. | 6 |
| Zero 8-16-24-32-40-48-56-64 | Any byte can be a zero | 8 |

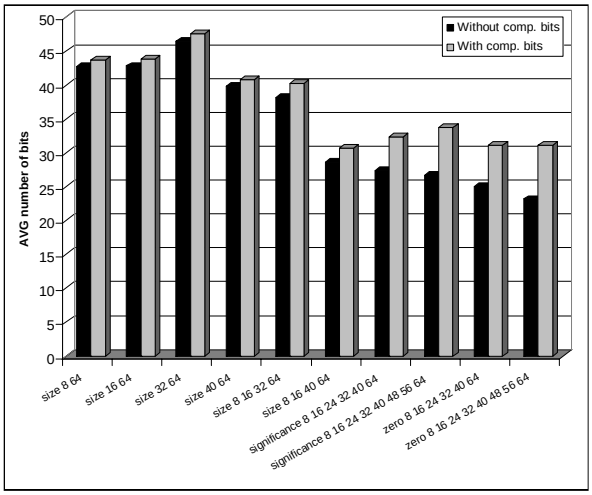


**Fig 2.** Average Data Size for the SpecInt2000

An initial study of the average compressed value size using the schemes listed above is shown in Fig 2. The average size was computed as the average of the number of bytes for each access to the register file, data cache, functional units, and the rename buffers. The first column shows the average data size without the format bits, and the second column shows the average size with the format bits. On average, ignoring the format bits, the *zero compression* mechanism achieves the best compression (23 bits for the configuration where every byte can be compressed). However, when the format bits are included, the best scheme is the *size compression* mechanism with an average of 30 bits per value (for the configuration in which the values are compressed to 8, 16, 40 or 64 bits).

This initial data indicates that any of the three proposed schemes can perform well (they reduce the effective data-width from 64 bits to 30 bits). In the next section we

describe several methods for using value compression for subsystems belonging to a processor's datapath. Then we analyze the energy consumption for the three value compression mechanisms when used as processor-wide compression techniques.

## 3 Related Work

Most of the work on value compression has targeted just one structure of the pipeline. In earlier work, [4] we proposed ways of using significance compression across all pipeline stages of an in-order, single-issue processor. Significance compression is also performed in main memory, and as compressed values flow through the pipeline the format bits control the gating off of unused storage and functional unit bytes. Nevertheless, that work is for a 32-bit ultra low power machine (i.e. performance is not a concern). The work in [4] is extended to 64-bits and uses compile-time mechanisms in [5]. Other work in value compression tends to focus on specific processor blocks or pipeline stages, as described below.

### 3.1 Processor Front-End

The primary functions performed in a processor's front end are instruction caching and branch prediction. Simple zero compression was proposed for the instruction cache [12], resulting in a 10% reduction in the energy consumption of the cache.

To the best of our knowledge there have been no published results on value compression to reduce energy requirements of branch prediction. However, in Section 4, we show performance figures of applying the zero compression mechanism of Villa et al. [13] and the significance compression method of Canal et al. [4] to branch predictors. The power savings during branch prediction comes from compressing values held in the branch target buffer (BTB).

There has also been a proposal for value compression while performing value prediction. Sato and Arita [11] split the structure that keeps the predicted values into two similar structures, where one holds byte-wide data and the other holds 64-bit data. This structure is shown to be beneficial for energy saving because most of the instructions' output-value widths do not change and a large portion of them (as shown in the data width distribution in Fig 1) are narrow.

### 3.2 Processor Back-End

In the processor back-end, we begin with the register file where Fig 3 depicts a simple value compression mechanism. For simplicity, the compression bits have been depicted in a separate structure. Before accessing the register file, the compression bits are read so that the access to the register file can be reduced to the specified bytes.

Canal et al. [4][5] propose dynamically compressing values so they are stored and retrieved along with their compression bits as shown in Fig 3. Brooks et al.[1], Loh [8] and Nakra et al. [9] propose similar techniques for exploiting narrow width operands to reduce functional unit energy requirements and, at the same time, to increase performance. Their techniques pack instructions that use narrow operands so that they can be executed in a single ALU (i.e. one 64-bit adder can compute four 16-bit additions). The differences between the various approaches lie in the ways the

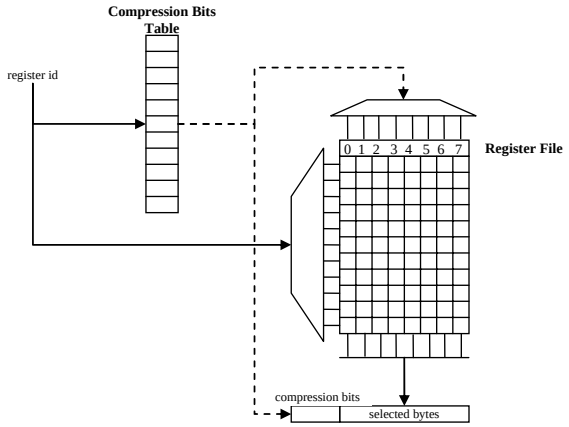


**Fig 3.** A register file with value compression capabilities

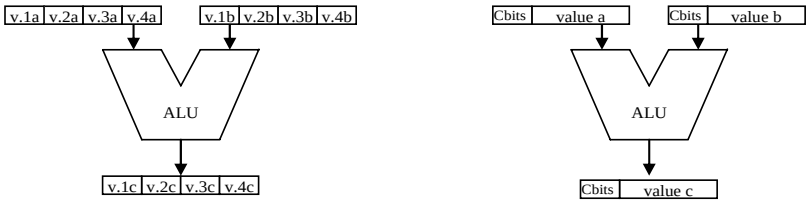


**Fig 4.** (a) ALU with packing capabilities, (b) ALU with value compression capabilities

narrow widths are obtained. Brooks [1] introduces hardware that dynamically detects the widths of operand values. Loh [8] extracts the data-width from a data-width predictor and thus a recovery mechanism is needed in case the prediction is wrong. Finally, Nakra et al. [9] set the width at compile-time. In this research [1][8][9], the register file is modified in two possible ways: either by incrementing the number of read and write ports to the banks of the register file holding the low-order bytes; or by replicating the lower part of the register file.

The implications for the functional units (FUs) result in two alternatives: Brooks [1], Loh [8] and Nakra [9] extend the FUs with the capability of executing multiple narrow-width instructions (see Fig 4a). On the other hand, Canal et al. [4][5] extend the functional units so that the FUs can operate with compressed values and generate the compression bits (see Fig 4b). In terms of implementation of these alternatives, Choi et al. [6] present several FU implementations that turn off the portions of the FU that compute the high-order bits when these are just a sign-extension of the least significant ones (the boundary between the high-order and low-order bits is analyzed and set in their work).

### 3.3 Data Cache

Several value compression methods have been proposed for reducing energy consumption in the memory subsystem. Most of methods are focused on on-chip caches. The data-cache has been shown to be one of the more power-hungry structures in a microarchitecture [7][12]**.** Fig 5 shows a data cache enhanced with value compression capabilities.

Typical implementations compress and decompress data when it is moved between the first and the second level caches. The same compression mechanisms can be used in all the memory hierarchy [13], and more sophisticated schemes [12] can be used in lower levels of the memory hierarchy for achieving higher compression ratios at the expense of some increase in latency -- not critical in lower memory levels. Several compression mechanisms have been proposed: *zero compression* [13] eliminates the bytes that are set to zero; *active data-width* [10] compresses the values to certain ranges (6,14,24 or 32 bit); a *frequent value cache* [15] has a list of most frequent values for the high-order bits (32 bits); and the last scheme analyzed is the *significance compression* [4] which eliminates the bytes that are a sign-extension of the previous one.

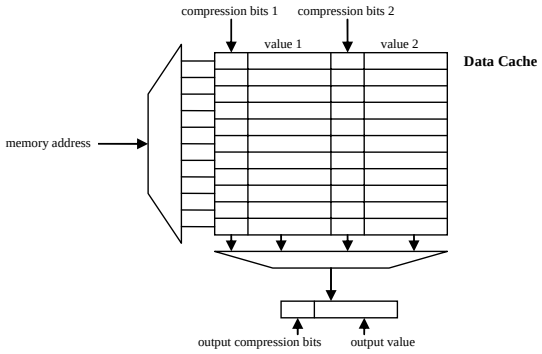


**Fig 5.** Data Cache with value compression capabilities

Villa et al. [13] propose an encoding where one bit per byte indicates whether the byte is null (zero). When the data is accessed, the compression bits are read first in order to just perform the activation of the parts that have a value different from zero. Okuma et al. [10] propose dividing the cache into several sub-banks where each sub-bank keeps a portion of the value (32-bit wide in their case). For each memory access, just the sub-banks with significant data are accessed. In their case, one sub-bank holds the lowest significant six bits, the next sub-bank holds the following 8 bits, the third sub-bank keeps the next 10 and the last bank holds the last (most-significant) 12 bits. This compression scheme needs two bits per word and is very similar to the more general one analyzed in this paper under the name of *size compression*.

## 4 Value Compression Comparison

In this section, we analyze the three value compression mechanisms (*size compression*, *zero compression* and *significance compression*) in terms of power. Starting from the overall processor energy reduction, we analyze some of the more interesting structures: data caches, instruction caches, register file, functional units and branch predictor. At the end, we consider the behavior in terms of peak power of the value compression mechanisms. Note that there are no performance (IPC) results because the compression mechanisms have no effect on performance. Thus, the results presented on energy reduction can be directly translated to Energy-Delay and Energy-Delay square metrics.

### 4.1 Experimental Framework

The Wattch [2] toolset is used to conduct our evaluation. The main architectural parameters of the assumed out-of-order processor are given in Table 1. We use the programs from the SpecInt2000 suite with their reference inputs. All benchmarks are compiled with the Compaq-Alpha C compiler with the maximum optimization level. Each benchmark was run to completion.

**Table 1**: Machine parameters

| Parameter | Configuration |
|---|---|
| Fetch Width | 4 instructions |
| I-cache | 64KB, 2-way set-associative. 32-byte lines, 1-cycle hit time, 6-cycle miss penalty. |
| Branch Predictor | Combined predictor of 1K entries with a Gshare with 64K 2-bit counters, 16 bit global history, and a bimodal predictor of 2K entries with 2-bit counters. |
| Decode/Rename width | 4 instructions |
| Max. in-flight instructions | 64 |
| Retire width | 4 instructions |
| Functional units | 3 intALU + 1 int mul/div3 fpALU + 1 fp mul/div |
| Issue mechanism | 4 instructions Out-of-order |
| D-cache L1 | 64KB, 2-way set-associative. 32-byte lines, 1-cycle hit time, 6-cycle miss penalty |
| I/D-cache L2 | 256 KB, 4-way set associative, 64-byte lines, 10-cycle hit time.16 bytes bus bandwidth to main memory, 100 cycles first chunk, 2 cycles interchunk |
| Physical registers | 96 |

### 4.2 Energy Savings

In addition to the average data size (shown in Fig 2), several other factors such as switching activity are important when computing dynamic energy reduction. Although storing more compression bits results in wider structures, the activity of these wider structures is what determines energy consumption, not the size. Thus, it can be the case that a wider structure has less activity than a narrower one. In this section, we give results for the best performing schemes.

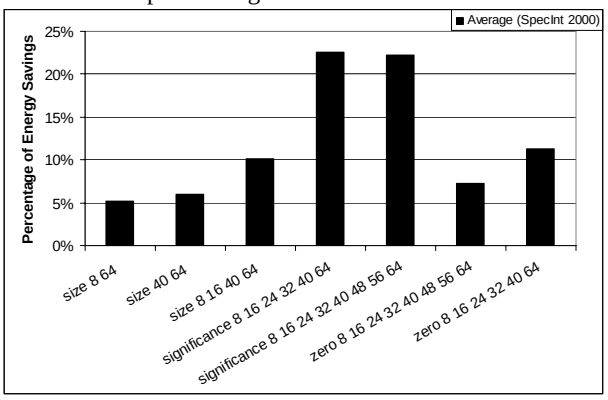


**Fig 6.** Processor Energy Savings

The energy savings of the mechanisms analyzed in this work are given in Fig 6. *Significance compression* achieves higher energy savings (more than 20%) despite the use of 7 extra bits per word. The best *size compression* scheme (around 10% energy savings) is the one that compresses values to 8, 16, 40 and 64 bits. The fact that the scheme includes the memory addresses (typically 5 bytes long) allows it to perform better than the other *size compression* mechanisms. The *zero compression* mechanism achieves a maximum of 11% overall energy reduction.

In the following figures, we analyze behavior of value compression schemes for several structures (instruction-cache, data-cache, register file and ALU). Fig 7a) shows the energy benefits in the data cache (both addresses sent to the cache and the data stored/loaded). The distribution of the energy savings in the data cache is similar to that of the whole processor. In this case, the *significance compression* energy savings are close to 14% and the version of *significance compression* that compresses all the bytes (not just up to the $5^{th}$ byte) performs better than the other configurations of *significance compression*.

Fig 7b shows the reduction in activity in the instruction cache. Since the instruction word is 32-bit wide (in the Alpha ISA used in this study) just three mechanisms are evaluated. The first (labeled *size*) compresses the data to 8, 16, 24 or 32-bits in the same way as *size compression* presented earlier. The second method (labeled *significance*) compresses the instructions using significance compression to 8, 16, 24, and 32-bits. Finally, the third column (labeled *zero*) compresses the instructions using *zero compression* where each byte of the 32-bit word can be tagged as being zero. All the schemes perform very well and they achieve a 30% energy reduction minimum in the instruction cache indicating that Alpha instructions are compressible in a way that the schemes are able to find and exploit.

Fig 7c shows the percentage of reduction of the energy consumed by the ALU. The difference between *significance compression* and the other schemes is larger in this case (almost 50% vs 25%). Fig 7d shows the energy savings for the register file. The savings scale up to 50% for *significance compression* while *size compression* reaches a 33% reduction in energy and *zero compression* is a little bit behind.

Finally, Fig 7e shows the energy reduction of the branch predictor (just the BTB). In this case the savings are smaller since the compressibility of addresses shows to be minimal.

### 4.3 Peak Power Reduction

Peak power is an important metric because it determines the maximum possible burst of power that a processor might consume. This translates directly to hot spots and to the temperature-thermal limits of the processor. Although one may think that compressing the data may not have a direct impact on peak power because there may be cycles where every computation will need 64 bits, our experiments show that peak power is significantly reduced with the proposed compression mechanisms. The peak power shown in Fig 8 corresponds to the execution of the SpecInt2000 suite.

As in the case of the energy consumption, the significance compression mechanism achieves an 18% peak power reduction. It is interesting to see that the configuration of significance compression that achieves the highest energy reduction (see Fig 6) is not the best in terms of peak power reduction (see Fig 8) where the scheme that compresses all the bytes (significance 8,16,24,32,40,56,64) performs a little bit better. The fact that it can compress bytes within large words makes it perform better in terms of

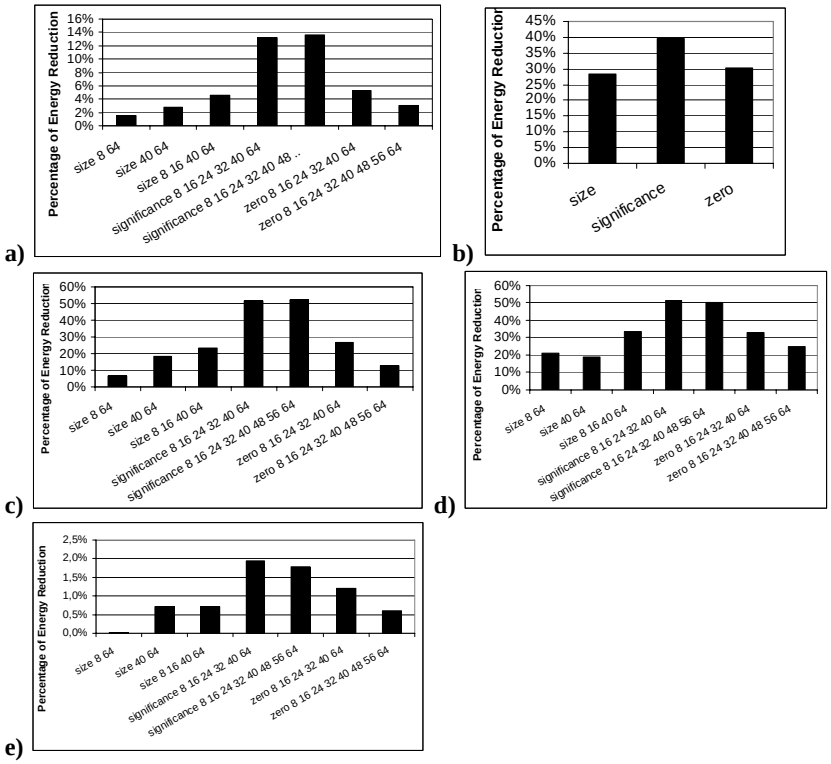


**Fig 7.** Energy Savings for: (a) Data Cache (b) Instruction Cache (c) ALU (d) Register File (e) Branch Predictor

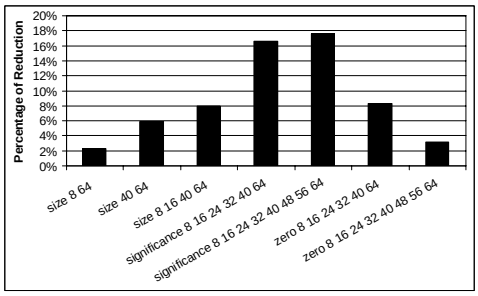


**Fig 8.** Peak power reduction.

peak power. The size compression mechanism achieves, in its best configuration, an 8% peak power reduction while the zero compression mechanism stays above the 8% line.

Benchmarks aside, one can conceive of (or contrive) a program with uncompressible data. In this case, the peak power would not be reduced. In fact, the extra bits needed by the data compression could even increase the worst case peak power. Nevertheless, we argue that the small complexity of the required hardware mechanisms does not add a significant overhead in this worst case peak power because there are more power hungry units such as the clock network and the caches.

## 5 Conclusions

We have focused on the value compression paradigm and the proposals around this topic. The compression of data values for different microarchitecture components has been shown to be an effective way of reducing the overall power consumption of processors. By reducing the activity levels, value compression achieves a significant reduction in dynamic energy consumption. At the same time, value compression can be used to make the different components of the pipeline simpler (or smaller) and thus further reducing the energy –in this case, the static energy consumption. Furthermore, we have shown that value compression can reduce the run-time peak power consumption and thus it can be a good approach for temperature-aware computing. Several studies have used different kinds of value compression mechanisms to achieve these goals. In this work, we have extended, analyzed and compared them.

## References


1. D. Brooks and M. Martonosi, "Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance", in Proc. of 5th. International Symposium on High-Performance Computer Architecture (HPCA-5), 1999.
2. D. Brooks, V. Tiwari and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimization", in Proc. of the 27th Annual International Symposium on Computer Architecture, June 2000.
3. G. Cai and C.H. Lim, "Architectural Level Power/Performance Optimization and Dynamic Power Estimation", Cool Chips tutorial of the 32nd Int. Symp. On Microarchitecture 1999.
4. R. Canal, A. González and J.E. Smith, "Very Low Power Pipelines using Significance Compression", in Proc. of the 33rd Int. Symposium on Microarchitecture, Dec. 2000.
5. R. Canal, A. González and J.E. Smith, "Software-Controlled Operand Gating", in Proc. of 2nd International Symposium on Code Generation and Optimization, March 2004
6. J. Choi, J. Jeon and K. Choi, "Power Minimization of Functional Units by Partially Guarded Computation", in Proc. of the 2000 International Symposium On Low Power Electronics and Design (ISLPED'00), pp. 131-136, Rapallo (Italy), August 2002.
7. R. Gonzalez and M. Horowitz, "Energy dissipation in general purpose processors", IEEE Journal of Solid State Circuits, v. 31, n. 9, pp. 1277-1284, September 1996.
8. G. Loh, "Exploiting Data-Width Locality to Increase Superscalar Execution Bandwidth", in Proc. of the 35th International Symposium on Microarchitecture (MICRO-35), pp. 395-405, Istanbul (Turkey) November 2002.
9. T. Nakra, B. Childers, and M.L.Soffa, "Width Sensitive Scheduling for Resource Contained VLIW processors", FDDO Workshop (MICRO33), Dec. 2001.
10. T. Okuma, Y. Cao, M. Muroyama and H. Yasuura, "Reducing Access Energy of On-Chip Data Memory Considering Active Data Width", in Proc. of the 2002 Int. Symp. On Low Power Electronics and Design, pp. 88-91, Monterey (CA-USA), August 2002.
11. T.Sato and I. Arita, "Table Size Reduction for Data Value Predictors by Exploiting Narrow Width Values", in Proc. of the 2000 Int. Conf. on Supercomputing, May 2000, pp.196-205.
12. J. Turley, "PowerPC Adopts Code Compression", Microprocessor Report, October 1998.
13. R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. E. Wazlowski, and P. M. Bland, "IBM Memory Expansion Technology (MXT)", IBM Journal of Research and Development, Volume 45, Number 2, 2001, pp. 271-286.
14. L. Villa, M. Zhang, and K. Asanovic, "Dynamic Zero Compression for Cache Energy Reduction", in Proc. of the 33rd International Symposium on Microarchitecture, Dec.2000.
15. J. Yang and R. Gupta, "Energy Efficient Frequent Value Data Cache Design", in Proc. of the 35th International Symposium on Microarchitecture (MICRO-35), pp. 197-207, Istanbul (Turkey), November 2002.