Low Energy Clustered Instruction Fetch and Split Loop Cache Architecture for Long instruction Word Processors

Murali Jayapala, Francisco Barat, Pieter Op de Beek, Francky Catthoor and Rudy Lauwereins *

ESAT/ACCA, K.U.Leuven, Belgium
{mjayapal, fbaratqu, pieter}@esat.kuleuven.ac.be

IMEC, Leuven, Belgium
{catthoor, lauwerei}@imec.be

Abstract

In current embedded systems for multimedia-oriented applications, low power is one of the key constraints. The core of such systems are typically application-specific programmable processors (ASIPs). Power analysis of such processors, especially the long instruction word processors, indicate that a significant amount of power is consumed in instruction memory hierarchy and associated control. Past proposals have addressed this problem by inserting a loop cache close to the data path resulting in a clear power saving. In this paper we propose a software-triggered split loop cache placed close to the data path combined with a clustered fetch mechanism, to ameliorate the situation. Further, the fetch and issue mechanisms are clustered in accordance with the application-specific data path clustering. Also, within each cluster a software triggered instruction buffer is employed. Initial experimental results performed on several representative media application benchmarks show the effectiveness of this architecture in achieving low power.

1 Introduction and Motivation

In current embedded systems for multimedia oriented applications, low power is one of the key constraints. The core of such systems are typically application specific programmable processors (ASIPs). Recent trends have shown that for multimedia oriented applications like video compression and decompression, wireless communications, speech and image processing, VLIW processors and alike are particularly effective in achieving high performance [10]. However, power analysis of such processors indicate that up to 20 to 25% of total processor power is consumed in instruction memory hierarchy and associated control [2][17]. In our proposal, we address this problem of reducing power consumption in instruction memory hierarchy and associated control, by means of a clustered instruction fetch and split loop caches. Also, we provide a simple framework for analysis of energy requirements in different loop cache schemes depending on the application.

Some of the motivations for our approach have been derived from the observations made in the VLIW architectures and in several representative media benchmarks. These observations are explained in sections 1.1 and 1.2.

1.1 Architecture

The instruction fetch/decode and the instruction cache hierarchy in most of the current VLIW processors are as illustrated in Figure 1. In every instruction cycle, large number of function units in the execution core are fed with instructions by a centrally located instruction fetch/decode logic. This logic in turn fetches a long instruction word from a wide (512 bits wide cache line in TMS320C6211 [18] and TriMedia [19]) instruction cache. Also, the interconnect between cache and fetch/decode logic is wide and the interconnect between fetch/decode and the execution core is not only wide but also long. This kind of architecture does not scale well in terms of power and performance with increasing number of function units. In fact, the power consumption of some of the hardware structures in this hierarchy, like the caches, are known to grow exponentially [16]. Also, these are the main power consuming elements in the instruction memory hierarchy. Every instruction cycle there is switching in almost all the structures shown in Figure 1. Even when some of the function units are not computing anything, they are issued with NOPs, leading to significant and unnecessary power consumption.

1.2 Program Behavior

It is observed from some of our media benchmarks that significant amount of time is spent in small and tight loop nests. In Table 1, $'a'$ shows the percentage of exe-

*This work is supported in part by MESA under the MEDEA program.

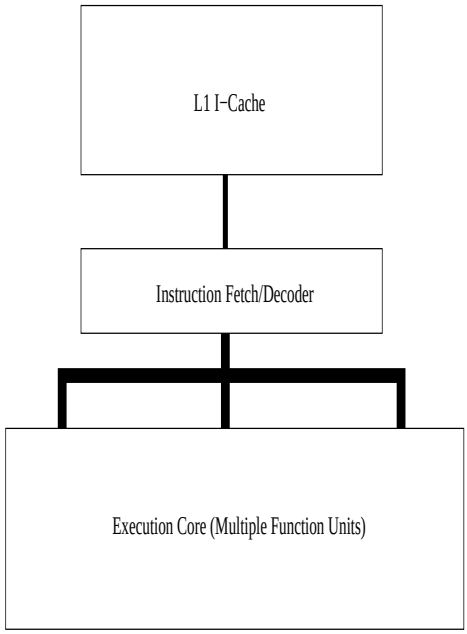

Figure 1: Centralized architecture in current instruction memory hierarchy

Application	Nature	$'a'$
Cavity Detection	Image Processing	70%
GSM	Wireless Communication	67%
Epic	Image Compression	75%
Mpeg2 Decoder	Video Compression	50%

Table 1: % of execution time of loop nests in some representative media applications

cution time spent in various loop nests within an application.

A second observation is that the type of computation in the loop nests vary from one loop to another. For instance.

- *Arithmetic and logic computations*: Computations performed over data elements of certain data structures like, variables and arrays.

- *Address Calculations*: Computations of the complex index expressions of certain data structures, specifically data arrays [14].

Within the loops that are engaged in initializing or loading the data arrays or transferring the data from one array to another, only address calculations are performed and no computations are done on the data themselves. However, within loops that perform computations on the data elements, there are more arithmetic computations than address calculations. Even if these loops have address calculations these two kinds of computations could be partially decoupled with some code transformations.

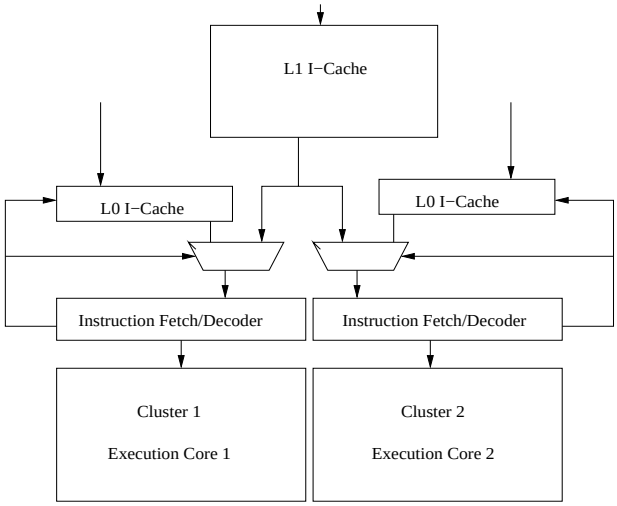

Figure 2: Decentralized architecture for instruction memory

Based on these observations, our proposal of clustered instruction fetch and split loop caches are illustrated in Figures 2 and 5. Also, a simple framework is provided for analysis of energy requirements in different loop cache schemes depending on the application.

The rest of the paper is organized as follows. In section 2, a brief description of work available in the literature related to low power optimizations is provided, followed by explanation of different loop cache schemes along with the clustered architecture are in sections 3 and 4. Finally, results of initial simulations performed over the loop cache schemes and a conclusion are provided in sections 5 and 6.

2 Related Work

In regard to the problem of reducing power in instruction memory hierarchy, many researchers have attempted to solve it with various amounts of success. The low power techniques proposed by several authors vary from logic and circuit level optimizations to architectural optimizations to pure software optimizations. An overview of the former is presented in [1][15], and few examples of the latter can be found in [5][13]. However, our approach is orthogonal to these optimizations and hence the two could be combined together to yield better results.

Based on similar observations on multimedia applications, as mentioned in the previous section, several authors have proposed loop cache architectures [3][4][9][11]. However, a combined energy requirement analysis of all these loop cache architectures has not been done.

In regard to clustered (decentralized) architectures, there are many proposals for data path clustering by various authors. A classic reference in this regard can

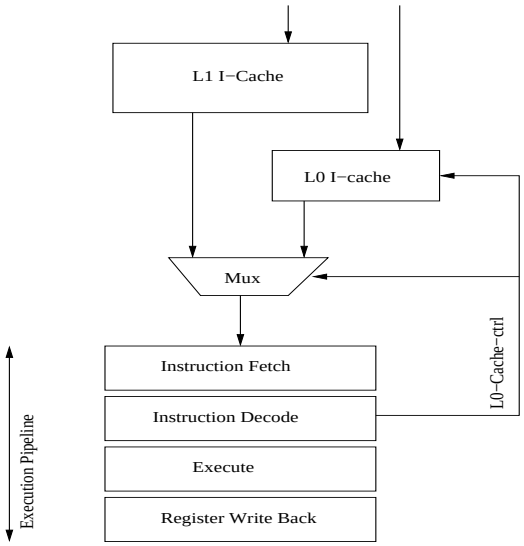

Figure 3: Loop cache architecture (L0-I-Cache placed before instruction decode)

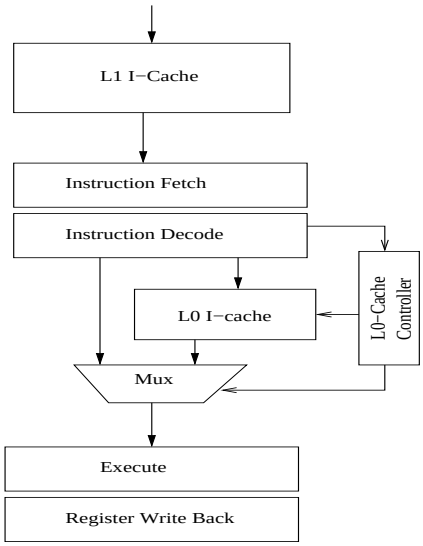

Figure 4: Loop Cache Architecture (L0-I-Cache placed after instruction decode)

be found in [8]. However, very few proposals address clustering in instruction control. One such proposal can be found in [20], where a clustering scheme in the context of superscalar processors is proposed. However, the clustering is at the instruction control and does not extend the same to the instruction caches or any of the memories.

3 Loop Cache Architecture

In our proposal a small cache/buffer is placed within each cluster and close to the data path or the execution core. This is a special cache to be used for programs with high temporal and spatial locality, more specifically for the loops. The cache could be placed either before or after the instruction decode stage, as illustrated in Figures 3 and 4. The complexity of the cache controller and the support for the control constructs depends on the loop cache location. In deciding where to place the loop cache, trade offs are involved. Hence, energy consumption in each case has to analysed and compared. This is a clear motivation for a energy requirement analysis of different loop cache schemes within a common framework.

3.1 Case (a): Loop cache placed before instruction decoder

As mentioned before, the operation of the loop buffer is triggered by software. An explicit way of triggering is by having a special instruction, ***lcon*** (loop cache ON). When this instruction is fetched and decoded, a signal *L0-cache-ctrl* is enabled, so that all further instructions are fetched from the loop buffer. The signal basically selects the appropriate inputs of the multiplexer and enables the loop cache controller.

Once the loop cache is initiated all further instructions are fetched from loop cache. The operation is very similar to that of a L1 instruction cache. In case of a hit, the requested instruction is passed onto the instruction decoder. In case of a miss, the controller passes on the request to the level above and then updates the loop buffer and simultaneously passes the instruction to the instruction decoder. The loop buffer is direct mapped, so the cache controller is very simple without any hardware overhead. The miss handler passes the request to L2 cache, instead of L1 in order to avoid unnecessary duplication of instructions in L1 cache. Also, no additional penalty should paid in the performance which could be incurred by transferring the instruction from L2 to L1 and then to the loop buffer.

The termination of the access to loop cache is done explicitly using another instruction, ***lcoff*** (loop cache OFF). When this instruction is encountered, the instruction decoder disables the *L0-Cache-ctrl*, hence disabling the access to loop buffer. Now, all further instruction requests are fetched from L1 instruction cache. This kind of operation of the loop cache is similar to the schemes proposed in [4][11].

Some advantages of this scheme are, the support for all the control constructs within the loop. No special loop detection mechanism is need, so nested loops can also be supported. The loop cache controller is quite simple. Specific parts of the code can be explicitly placed in this loop cache, and if the loop nest can be fitted into the buffer, the misses can be restricted to only compulsory misses. Due to high temporal and spatial locality of the loops, the rest of the instruction memory hierarchy can

be powered down.

However, there are still a few shortcomings in this basic scheme. The buffer needs tag memory to identify the misses. Also, during every instruction cycle the instruction decoder is active.

3.1.1 Energy analysis

Some insights on how energy is reduced can be obtained by analysing the equation representing the energy consumption in the instruction memory hierarchy.

$E_{inst,NL0} = C' P_{L1} + C' P_D$

$E_{inst,L0,B} = aCP_{L0} + (1-a)CP_{L1} + CP_D$

where,

$E_{inst,NL0}$	Energy consumption of instruction window without loop cache
$E_{inst,L0,B}$	Energy consumption of instruction window with loop cache (loop cache, placed *before* Instruction Decoder)
P_{L1}	Power consumption of L1 I-Cache
P_{L0}	Power consumption of loop cache
C'	# of cycles taken to execute a program P (without loop cache)
C	# of cycles taken to execute a program P (with loop cache)
a	Fraction of time during which loop cache is activated (0 - 1) $a = \frac{\#\ of\ cycles\ when\ loop\ cache\ is\ ON}{Total\ \#\ of\ cycles}$
P_D	Power consumption of the instruction decoder.

Now, it could be argued that, $C' \simeq C$. The reasoning is that, instructions to loop cache are mapped such that there is no conflict or capacity misses, but only compulsory misses. When there is a compulsory miss, the instruction is fetched from the L2 Cache or from another level above. The miss latency of loop cache is slightly larger than L1 I-Cache (but, has lower hit latency), and total number of misses (which are compulsory) are small compared to total number of hits. So, the increase in execution cycles is very small. Hence for practical purposes, $C' \simeq C$ (alternatively this implies that, there is no loss in performance).

Effectively,

(1) $E_{inst,NL0} = C\{P_{L1} + P_D\}$

(2) $E_{inst,L0,B} = C\{P_{L1} - (P_{L1} - P_{L0})a\} + CP_D$

Also, $P_{L0} < P_{L1}$, because a loop cache is much smaller than a L1 I-Cache. So, the reduction in energy is determined by a and the difference between P_{L1} and P_{L0} .

The upper bound for $E_{inst,L0,B}$ represents that energy consumption can be no more than $E_{inst,NL0}$. i.e, when $a = 0$ and the loop buffer is not used at all. The lower bound represents the lowest possible energy consumption for $E_{inst,L0}$. This basically indicates that, the whole program contains nothing but a loop i.e, when $a = 1$ and the whole program is completely mapped onto the loop cache, so that the L1 I-Cache is never used.

Factors, P_{L0} and a, in the above equations are not independent. If P_{L0} is made larger (i.e, larger loop cache), then some of the loops which were initially not mapped onto the loop cache can now be mapped making a larger (since they can fit into the loop cache without any capacity misses). But it is interesting to note that the *maximum* value of a for a given program, is a characteristic of that program itself. It represents the amount of loop nests inherent in that program. If the whole program contains just loops, then a could be as high as 1, achieving maximum energy reduction.

However, for a given P_{L0} (fixed loop cache size) and a given program, it is possible to conceive of software transformations, to increase the value of a and try to reach maximum value for that program. These transformations should aim at the loop nests which do not fit in the loop cache, and transform those loop nests so that they can be mapped into it. Some interesting software transformations has been proposed in [4][12].

So in conclusion, if there are any loop nests in the program, and if those loop nests are mapped onto a smaller loop cache, then there is always a reduction in energy. And the *amount of reduction is determined by 'a' and by the difference between P_{L1} and P_{L0}.*

It is also interesting to note that a similar argument holds for energy consumption of the interconnect between the L1 I-Cache and the instruction decoder (program memory bus).

3.2 Case (b): Loop cache placed after instruction decode

This loop cache organization is as shown in Figure 4. Again the loop buffer operation is triggered by a special instruction ***lcon*** in the software, as described in the previous section. When this instruction is encountered, the buffer operation enters a FILL phase. Here, the decoded instructions are stored simultaneously in the loop buffer and also fed to the execution core. During this phase, the counter in the buffer controller is loaded with the exact number of iterations the loop will execute.

Once all the decoded instructions of the loop are stored in the buffer, all further instructions are fed by the buffer controller to the execution core; this is the RUN phase. This phase terminates and returns to IDLE phase,when the counter in the controller reaches zero. In the IDLE phase, the loop buffer is not used, and all the instructions are fed directly from the instruction de-

code stage. This part of the scheme is very similar to the scheme presented in [2][3][9]. However, additional considerations have been made in our proposal, namely: local controller and several loop nest handling (support for control constructs).

A significant advantage of this scheme is that the energy consumption per instruction can be very low during the RUN phase. This is because, only the execution core and the loop cache are active, while the rest of the instruction memory hierarchy including the instruction decoder can be powered down. Also, the buffer does not need any tag memories since the operation is well orchestrated and there will be no cache misses.

However, the reduction comes with certain trade-off in the hardware complexity of the buffer controller. With no support to any of the control constructs the controller can be kept simple. For partial support for control constructs the complexity of the controller is quite high.

Since the buffered instructions are decoded, the buffer size is larger compared to the case where buffer is placed before instruction decoder. For certain sizes of the buffers, the energy consumed in case (a) could be lower than case (b), in spite of the tag array.

3.2.1 Energy analysis

Besides the reductions mentioned in the previous section, there are additional reductions by powering down the instruction decoder. To evaluate the energy reduction, the power consumed by instruction decoder should also be included in the equations representing energy.

(3) $E_{inst,L0,A} = C\{aP_{L0} + (1-a)P_{L1} + (1-a)P_D\}$

$E_{inst,L0,A}$ Energy Consumption of instruction window with a loop cache (loop cache, placed *after* Instruction Decoder)

As it can be seen from this equation, further reduction in energy can be obtained than in case (a), Eq (2). The reduction however depends on how small P_{L0} is compared to P_{L1} and how large a is. It is important to note that P_{L0} represents not only the power consumption in the loop cache but also includes the power consumption of the associated controller. In order to obtain significant energy reduction, the power overhead due to the controller should also be kept as low as possible.

4 Clustered Fetch and Split Loop Cache Architecture

The architecture of a decentralized instruction memory hierarchy, which is the main point of our proposal is illustrated in Figures 2 and 5, for two data path clusters.

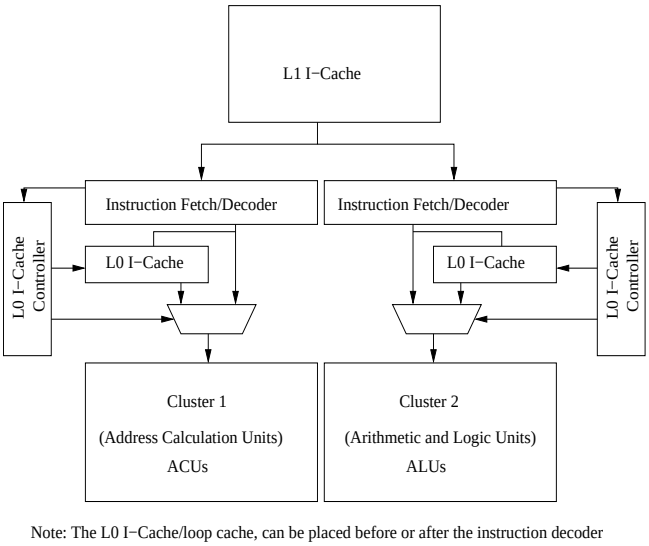

Figure 5: Decentralized architecture for instruction memory

Application-specific units with similar functionality are grouped together to form clusters. In the figure illustrated, one of the cluster could be arithmetic calculation unit and another cluster could be an address calculation unit [14].

Each data path cluster has an instruction decoder and a loop cache. The buffer could be placed either before or after the instruction decoder, based on the trade offs in the controller complexity and the amount of control construct support needed. This scheme can be extended to multiple clusters as well.

When the loop caches are not in use, the instruction fetch mechanism of the clusters work synchronously, and are tightly coupled with each other. It works as if there was only one fetch mechanism. This is in fact the case with current VLIW architectures with instruction level parallelism. The instruction to a function unit is fetched and issued in parallel with instructions for the other function units. However, the significant difference in operation is when loop caches are enabled. The triggering of the loop cache operation is done explicitly for each cluster, as explained in the section 3. However, there are two different cases for the combined buffer operation.

When the loop body contains instructions for both clusters

Here, the instructions within the loop body specific to a cluster are stored in the corresponding loop buffers. Depending on which of the earlier schemes case (a) or case (b) is employed, the operation within each cluster will be as described in section 3. While these two caches are in operation, rest of the instruction memory hierarchy can be powered down.

When the loop body contains instructions for one of the clusters

Here, the instructions within the loop body are specific to a cluster. So, only that cluster's loop buffer is put into operation. While this loop cache is in operation, rest of the instruction memory hierarchy can be powered down, including the other cluster's loop cache. The data path of the other cluster need not be issued with NOPs, instead it could be powered down as well. This is one of the major advantages. Also, the loop caches among the clusters need not be of same size, they could be further optimized to make them as small as possible.

4.1 Energy analysis

By combining the energy equations for each cluster the energy consumption of the clustered loop buffers are,

(4) $ClustE_{inst,L0,B} = C\{\sum_i (a_i P_{L0i} + P_{Di}) + (1-a)P_{L1}\}$ for case (a) in section 3

(5) $ClustE_{inst,L0,A} = C\{\sum_i (a_i P_{L0i} + (1-a)P_{Di}) + (1-a)P_{L1}\}$ for case (b) in section 3.

where,

$ClustE_{inst,L0,B}$	Energy consumption of instruction window with clustered loop cache for case (a)
$ClustE_{inst,L0,A}$	Energy consumption of instruction window with clustered loop cache for case (b)
P_{L1}	Power consumption of L1 I-Cache
P_{L0i}	Power consumption of loop cache in the i^{th} cluster
a_i	Fraction of time during which loop cache is activated (0 - 1) in the i^{th} cluster $a_i = \frac{\#\ of\ cycles\ when\ i^{th}\ loop\ cache\ is\ ON}{Total\ \#\ of\ cycles}$
a	Fraction of time during which any of the loop cache is activated
P_{Di}	Power consumption of the i^{th} instruction decoder

The energy represented by the equations, namely (4) and (5), are to be compared with the energy equations for the non-clustered loop buffer architecture, namely equations (2) and (3), respectively.

In the case where loop body with instructions for both clusters (*upper bound* for energy consumption), the energy consumption of small instruction buffers (low P_{L0i}), combined with significant $a_i = a$, and smaller interconnects, (not in the energy equations) lead to lower $ClustE_{inst,L0,B}$.

In the case where loop body with instructions for one of the clusters (*lower bound* for energy consumption), the energy consumption is much smaller than (4).

Application	a	a_{ACU}	a_{ALU}
Cavity Detection	0.35	0.21	0.15
GSM	0.60	0.44	0.27
Epic	0.73	0.34	0.45
Mpeg2 Decoder	0.47	0.42	0.10

Table 2: a and a_i for each application

This is because, when one of the loop cache is in operation the rest of the instruction memory hierarchy is switched off including the loop cache and instruction decoder of the other cluster, which in turn means lower $ClustE_{inst,L0,A}$.

The main idea of the split loop caches is that smaller storage elements combined together have lower energy consumption compared to one large storage element. In employing this scheme, the long active interconnects which are a potential bottleneck for both power and performance can be avoided.

5 Experimental Results

All our initial simulations have been carried out using the Simplescalar tool suite [7] and the Wattch power estimator [6], which is integrated into the performance simulator of the former. The tool suite was modified to identify the new instructions namely, *lcon* and *lcoff*. Also, the cache behavior in the performance simulator was extended to incorporate the loop cache behavior as described in section 3. The loop cache was modeled as a small cache of 64 words (256 bits wide for centralized and 128 bits for clustered) for the cases where it was placed before the instruction decoder. In cases where loop cache was placed after the instruction decoder, it was modeled as a simple array (no tag memories) of 64 words. The benchmarks were compiled and simulated using this tool suite. So, from our simulation runs it was possible to obtain C, a, P_D, P_{L0}, P_{L1}, a_i, P_{L0i} and P_{Di}.

Within the benchmarks the loops were identified manually and were hand mapped into the loop cache. With such a mapping and for the above mentioned cache size, $'a'$ that we could achieve is as shown in Table 2. For the clustered cases, the loops were hand mapped to each clusters, and $'a'_i$, that we achieved are as shown in Table 2. The normalized energy estimates from the simulations are as shown in the Figure 6.

The different loop cache schemes that are compared here are

1. No loop cache {as represented in eqn (1)}
2. Non-clustered fetch and single loop cache, placed before instruction decoder {as represented in eqn (2)}

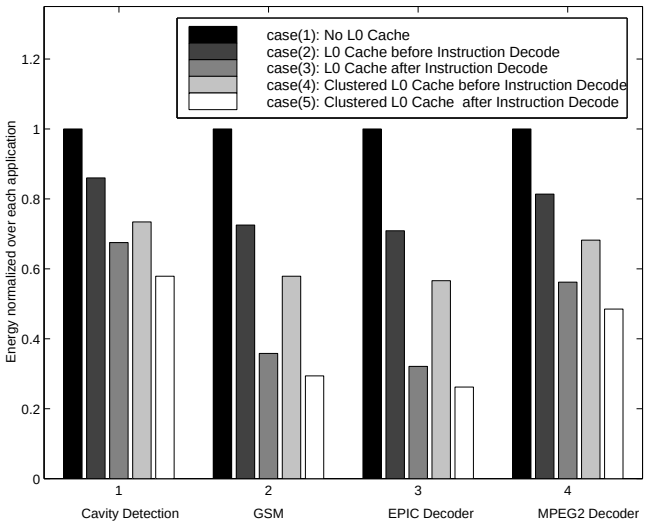

Figure 6: Comparison of energy consumptions in various loop cache schemes

3. Non-clustered fetch and single loop cache, placed after instruction decoder {as represented in eqn (3)}

4. Clustered fetch and split loop cache, placed before instruction decoder {as represented in eqn (4)}

5. Clustered fetch and split loop cache, placed after instruction decoder {as represented in eqn (5)}

The energy reductions from the figure indicate that, up to 70% (and 60% on the average) of the energy consumed in the instruction memory hierarchy can be reduced, by employing the loop cache schemes. The results on non-clustered loop cache schemes are close to the results shown in reference [2].

The energy reductions in cases (3), (5) over cases (2), (4) shows that, placing a loop cache after the instruction decoder could reduce energy much more than placing the loop cache before instruction decoder. However, the reductions shown in the figure were obtained by assuming that the loop nests were very regular and there was no nested loops or other control constructs. Also, the controller overhead which is present in cases (3) and (5) was not modeled. If these were not the case, then the trade offs in energy depending on the loop cache placement would be visible.

The energy reductions in cases (4), (5) over cases (2), (3) shows that, additional energy reduction of about 10% on the average can be achieved by clustering. This shows that clustered architecture is indeed more power efficient than the non-clustered architecture. However, it is important to note that the interconnect (Loop cache to Instruction decoder and instruction decoder to function units) energy was not modeled. If those figures were included, the reduction in energy could have been much more significant.

6 Conclusions and Future work

This paper presented a clustered fetch and split loop cache architecture along with an energy framework to analyze the energy requirements of different loop cache schemes for long instruction word processors based on the application behavior. This proposal aims at reducing the energy consumption in instruction memory hierarchy and associated control. The architecture is not only low energy inherently, but it is also scalable. Our initial experiments show that up to 70% reduction in energy could be achieved in instruction memory hierarchy. The loop cache techniques are orthogonal to the standard circuit or gate level techniques that are traditionally used by designers to reduce energy and can therefore be used to further reduce energy consumption without impairing performance.

Our future work is to extend a VLIW compiler to support the clustered fetch and split loop cache architecture. Here, instead of hand mapping the loops into the loop cache, the responsibility is handed over to the compiler. The granularity of the code which are placed in the loop cache can in principle be reduced to basic blocks as well and the current compiler can readily support this. Second aspect is to exploit the energy framework in the context of automatic loop cache architecture exploration for multimedia applications.

References

[1] B. Ackland and C. Nicol "High Performance DSPs-What's Hot and What's Not?", ISLPED 1998.

[2] T. Anderson and S. Agarwala, "Effective Hardware-Based Two-Way Loop Cache for High Performance Low Power Processors", International Conference on Computer Design 2000.

[3] R. S. Bajwa, M. Hiraki, et al, "Instruction Buffering to Reduce Powr in Processors for Signal Processing", IEEE Transactions on VLSI systems, vol 5, no 4, Dec 1997.

[4] N. Bellas, I. Hajj, C. Polychronopoulos and G. Stamoulis, "Architecture and Compiler Support for Energy Reduction in the Memory Hierarchy of High Performance Microprocessors", ISLPED 1998.

[5] L. Benini, et al, "Selective Instruction Compression for Memory Energy Reduction in Embedded Systems", ISLPED 1999.

[6] D. Brooks, V. Tiwari and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations", 27th International Symposium on Computer Architecture, June 2000.

[7] D. Burger and T. Austin, “The Simplescalar toolset, version 2.0 Technical Report”, University of Wisconsin Madison, 1997.

[8] R. Colwell, et al, “A VLIW architecture for a trace scheduling compiler”, IEEE Transactions on Computers”, 37(8):967-979, Aug 1988.

[9] M. Hiraki, R. S. Bajwa, et al, “Stage-Skip Pipeline: A low Power Processor Architecture Using a Decoded Instruction Buffer”, ISLPED 1996.

[10] M. F. Jacome, G. de Veciana, “Design Challenges for New Application-Specific Processors”, IEEE Design & Test of Computers, April-June 2000.

[11] L. H. Lee, W. Moyer, J. Arends, “Instruction Fetch Energy Reduction Using Loop Caches For Applications with Small Tight Loops”, ISLPED Aug 1999.

[12] N. Liveris, N. D. Zervas, C. E. Goutis, "A Code Transformation-based Methodology for Improving I-Cache Performance", submitted to ICECS, Malta, September 2001

[13] M. Mehendale, et al “Extensions to Programmable DSP architectures for Reduced Power Dissipation”, IEEE VLSI design Conference, 1997.

[14] M. Miranda, F. Catthoor, M. Janssen and H. De Man, “High-level Address Optimisation and Synthesis Techniques for Data-transfer Intensive Applications”, IEEE Trans. on VLSI Systems, Vol.6 No.4, pp. 667-676, December 1998.

[15] L. Nachtergaele, V. Tiwari and N. Dutt, “System and Architecture-Level Power Reduction of Microprocessor-based Communication and Multimedia Applications”, ICCAD 2000.

[16] W.T. Shiue and Chaitali Chakrabarti, "Memory exploration for low power, embedded systems," DAC, pp. 140-145, June 1999.

[17] Texas Instruments Inc, Technical report “TMS320C6000 Power Consumption Summary“, http://www.ti.com

[18] Texas Instruments Inc, Technical report,“TMS320C6211 Cache Analysis”, http://www.ti.com

[19] TriMedia Technologies Inc, “Trimedia32 CPU Handbook “, http://www.trimedia.com

[20] V. Zyuban and P. Kogge, “Optimization of High-Performance Superscalar Architecture for Energy Efficiency”, ISLPED’ 2000.