Instruction Buffering Exploration for Low Energy VLIWs with Instruction Clusters

Tom Vander Aa, Murali Jayapala, Francisco Barat and Geert Deconinck
K.U.Leuven/ESAT
Kasteelpark Arenberg 10
Heverlee, Belgium-3001

Rudy Lauwereins and Francky Catthoor
IMEC vzw
Kapeldreef 75
Heverlee, Belgium-3001

Henk Corporaal
TU Eindhoven, Electrical Engineering
Den Dolech 2
5612 AZ Eindhoven, Netherlands

Abstract— For multimedia applications, loop buffering is an efficient mechanism to reduce the power in the instruction memory of embedded processors. In particular, software controlled clustered loop buffers are energy efficient. However current compilers for VLIW do not fully exploit the potentials offered by such a clustered organization. This paper presents an algorithm to explore what is the optimal loop buffer configuration and the optimal way to use this configuration for an application or a set of applications. Results for the MediaBench application suite show an additional 18% reduction (on average) in energy in the instruction memory hierarchy as compared to traditional non-clustered approaches to the loop buffer without compromising performance.

I. INTRODUCTION AND MOTIVATION

Low energy is one of the key design goals of the current embedded systems for multimedia applications. Typically the cores of such systems are programmable processors. VLIW ASIPs in particular are known to be very effective in achieving high performance with reasonable low power for our domain of interest [1]. Examples of such processors are the Trimedia [2] processor from Philips or the 'C6x processors from Texas Instruments [3]. However, power analysis of such processors indicates that a significant amount of power is consumed in the on-chip (instruction) memory hierarchy: 30% of the total power according to [4]. Our experimental analysis shows that, if the appropriate data memory hierarchy mapping techniques are applied [5, 6], and if all methods to reduce power in the datapath are applied [7], this number goes up to 50%. Hence, reducing this part of the budget is crucial in reducing the overall power consumption of the system.

Loop buffering is an effective scheme to reduce energy consumption in the instruction memory hierarchy [8]. In any typical multimedia application, significant amount of execution time is spent in small program segments. Energy can be reduced by storing them in a small loop buffer instead of the big instruction cache (IL1). But, as more instruction level parallelism is extracted from the application, wider datapaths and wider loop buffers are needed. In the datapath, the register files and the interconnection network to them is the main bottleneck of wide VLIWs. This issue is resolved in recent papers using datapath clustering (see e.g. [9]). Data path clustering will split up these register file into several partitions, and restricts access to this register file to certain FUs. This reduces the ports to the register file and the number of wires in the interconnection network.

To obtain low power, clustering also needs to be applied to the instruction memory hierarchy. For this we proposed a clustered loop buffer architecture [10].

To ensure that this clustered loop buffer architecture leads to low power solutions we need to use it effectively. This has two aspects: firstly, we need to find the optimal loop buffer architecture configuration. We have to decide on the number of instruction clusters and the size of the loop buffer for each of the clusters (clustering step). Secondly, we need to decide what parts of the applications should be executed from the loop buffer (mapping step). These two aspects are not unrelated: you cannot do a good mapping if you do not know the loop buffer configuration you are mapping to, and to optimize the loop buffer for a certain applications you need to know what loops are executed from the loop buffer. To solve this interdependency an iterative method, as shown in Figure 1, is proposed.

Given a program compiled for a VLIW processor, and an initial loop buffer configuration, we first try to find the optimal mapping of loops on the loop buffer. For this mapping of loops we find the optimal loop buffer architecture configuration. The mapping stage is then repeated for this architecture, and so on. This process is repeated until we converge. Results show an average reduction in instruction memory energy consumption for 18% after convergence, as compared to the initial mapping with a single loop buffer. The rest of this paper is organized as follows: A brief account of the related work is presented in Section II. In Section III the software controlled clustered loop buffer organization is described. In Section IV the energy model under consideration for our exploration tool is outlined. Section V describes the loop buffer exploration algorithm, which is the main contribution of this paper. Section VI presents the simulation results and Section VII presents the
conclusions.

II. RELATED WORK

Several loop buffering schemes have been proposed in the past [11, 12]. An overview of the options can be found in [13]. Initially only inner loops without any control constructs could be mapped to the loop buffer. In [14] support is added for control constructs such as if-then-else, subroutine calls and nested loops. Our loop buffer architecture also supports conditional constructs, as well as mapping a set of nested loops.

The main reason to use a software controlled loop buffer is to exploit knowledge about the program in the compiler to reduce power. But, as is discussed in [8], software controlled loop buffers also do not need the energy consuming tag memories. Furthermore, they do not suffer any cycle penalty.

The idea to add compiler support has already been proposed in [15]. In that paper, however, a regular cache is used (with tags) and no energy model is used directly in the exploration framework. Furthermore, to the best of our knowledge, none of the aforementioned schemes analyze a clustered loop buffer organization like ours [10]; implicitly they assume a single logical partition of the loop buffer and a single controller.

The basic notion of distributing the storage in accordance with the datapath units, can already be seen in some of the VLIW architectures with a clustered datapath [16, 17]. For a fixed clustered datapath, the goal is to optimally schedule (map) instructions on the different clusters. In [18, 19] architecture design space exploration is performed for VLIW with clustered datapath, but only for homogeneous clusters. In [20] this is extended to heterogeneous clusters. However, the criteria for an optimal datapath clustering are different than those used in instruction clustering.

III. CLUSTERED LOOP BUFFER ORGANIZATION

Figure 2a illustrates the essentials of the clustered loop buffer under consideration. A more detailed analysis of the architecture can be found in [10]. Instructions are fed to the processing unit from either the level 1 instruction cache (IL1), or from the loop buffer.

Initially the loop buffer is disabled and the program executes via IL1. When a loop buffer control instruction is encountered marking the beginning of the loop that has to be mapped to the loop buffer, the loop buffer will be turned on. The form of this special instruction is lb<n<startaddress>,<endaddress>. (Lb<n means loop buffer on). Startaddress is the address of the first instruction of the loop and endaddress that of the last one. These values are stored in the local controller (LC) of each cluster and will be used during the execution of the loop.

When the loop buffer is used, a local controller (see Figure 2b) will translate the program counter (PC) to an index in the loop buffer (called NEW_PC in the figure) by subtracting the stored startaddress from the PC value. This NEW_PC is used as an index in the local controller table. Only if the first column of this table indicates that an operation is stored in the loop buffer partition for that PC, the loop buffer partition is accessed. This is an important feature since it saves energy (by not accessing the loop buffer partition) and area (by not storing any operations if that cluster is not used in that cycle). No NOP compression is used in the loop buffer, so if only FU is active in a certain cluster, the other FUs will execute an explicitly coded NOP instruction. What entry in the loop buffer partition is accessed depends on the second column in the local controller table.

When the LC detects that the program counter is bigger than endaddress, the loop buffer will be turned off.

IV. ENERGY DISSIPATION MODEL

A tree like representation for the loops, as shown in Figure 3, is extracted from the source code of the application. From this representation we can identify different mappings of loops on the loop buffer. The figure shows two possible configurations for a given program, both leading to different loop buffer sizes and different energy consumptions. In the first case loops B and C are loaded each to a bigger size. In the second, B and C are loaded each time the loop starts, leading to more loads from IL1.

A loop buffer configuration is a set of clusters each having a loop buffer partition of a certain depth and such that each functional unit is assigned to one and only one cluster.

If the loop buffer configuration and the mapping of the application is known, we can calculate the energy E in the instruction memory hierarchy with the following formula (Equation 1):

\[
E = \sum_{l,m} N_{unmapped}(l, m) \times E_{access}(l, m) + \sum_{l} N_{mapped}(l, 1) \times E_{access}(l) + \sum_{l} N_{mapped}(l, LC) \times E_{access}(LC) + \sum_{l} N_{unmapped}(l, m) \times E_{access}(c, m)
\]

In this formula, \(N(l, m)\) is the number of accesses of loop \(l\) to memory \(m\) due to the loading or execution of the loop.
$E_{\text{access}}$ is the energy per access of the loop buffer partition, the local controller or the instruction level 1 cache. The value of $E_{\text{access}}(c)$ for a certain loop buffer partition depends on the required depth and width of each loop buffer partition, which can be calculated when you know what loops are currently mapped and if the loops are mapped together or reloaded each time the loops are invoked.

The four sums correspond to the four places in the instruction memory where energy is consumed:

- (a) Executing instructions from IL1. This also incorporates all the code of the application that is outside loops.
- (b) Loading the instructions into the loop buffer from IL1 in the first iteration of the loop. The access to the loop buffer to write the loop are taken into account in the fourth sum.
- (c) Access to the local controller, when executing instructions from the loop buffer. Each LC is activated every cycle the loop buffer is used. So $E_{\text{access}}(LC)$ is the energy per access summed over all LCs.
- (d) Access to each loop buffer partition, when executing instructions from the loop buffer. Energy is only consumed if in a certain cycle this partition is effectively used. This information is extracted from the scheduled code.

Loops that are mapped to the loop buffer contribute to the last three terms, loops that are not mapped (unmapped) to the first term.

The energy values we have used for the memories ($E_{\text{access}}$) are calculated using the Wattch [21] power models. For the IL1 we used a regular direct mapped cache of 8KB, for the loop buffer partitions we used a cache without tags. The local controller is also modeled as a tag-less cache. The technology level used was 18µ. Instruction level 2 and above was not modeled since it’s contribution to the total energy is not significant. Although the Wattch model has some known limitations and is not correct when absolute values are required, it is still suited for our purpose since we only need good relative energy values.

V. Design Space Exploration

To find the optimal loop buffer configuration and the optimal way to use of this configuration, two questions need to be answered, corresponding to the last two stages in Figure 1.

1. What is the optimal loop buffer configuration, i.e. what is the best clustering and what is the optimal depth for each loop buffer partition? If a certain partition is too small not enough loops fit and there will be too many access to the IL1 cache. If the partition is too big the energy per access will not be optimal.

2. What is the best way to map the loops? If we decide to use the loop buffer for a certain loop, we still have several options: Do we map the loop entirely or only parts of the loop? If we load two loops like B and C in Figure 3, do we put them in the overlapping addresses or in non-overlapping addresses? The former will save us space in the loop buffer, but we will have to load the loop each time it starts. The latter case needs a bigger loop buffer, but will save us accesses to IL1 to load the loops (as can be seen from the energy model from the previous section).

A. Size of the Design Space

Although we have already split the design space into two orthogonal axes and explore them independently, each of them is too big to be explored using the brute force approach.

Given a clustering, to find a good mapping by trying all possible mappings and evaluating the energy function for each of them, would lead to more than $10^{18}$ combinations for the MPEG2 Decoding application. This is because the complexity is $O(2^{n \times m})$, where $n$ is the depth, and $m$ the width of the loop nest (cf. Figure 3).

Given a mapping, the number of loop buffer configurations grows exponentially with the number of FUs ($O(n^m)$ with $n$ the number of FUs). For 16 FUs this number is around $10^{19}$.

It’s clear that we need an effective heuristic. In the next two subsections we will explain the exploration algorithm for both problems and the used heuristics.

B. Mapping Exploration Algorithm

Instead of the brute force approach, we use a recursive algorithm to explore our design space in a more intelligent way (see Algorithm 1). The input of the algorithm is a set of loops of a certain program. The loops of a program are represented as a tree as already presented in Figure 3. The output is a set of solutions, where for each solution it is stated (1) what loops are mapped to the loop buffer, (2) if they have to be mapped together or loaded each time they are executed, (3) the energy of this solution, and (4) the required depths of each of the partitions. The required depths are represented as a depth vector. Examples of such depth vectors are shown in Table I, column 4.

The procedure Find_Mappings will be called with the top loop of the tree as an argument. If the program has multiple top loops, a dummy loop with an iteration count of one can be assumed around the whole program.

Two basic solutions exist for a loop passed to Find_Mappings: mapping the loop completely or not mapping it at all. Both solutions are kept since they will have a different energy consumption and required loop buffer depths. If the loop has children, Find_Mappings is called recursively to find the solutions of the child loops. These solutions are then combined in the procedure Filter_Solution. Here two heuristics are used to prune the design space: firstly, we only allow loop buffer depths that are a power of two (because that is also the most energy-efficient for the address decoders in the memories), and secondly, if for a certain loop and loop buffer size several solutions exist, we only keep one solution for each possible depth vector, namely, the most energy efficient solution. Although we do not have a proof, we believe that only this solution may lead to the optimum for the top loop. For several small examples we did an exhaustive search, and for these our second assumption was indeed true.
Algorithm 1 Design space exploration to find the optimal mapping and loop buffer configuration for an application. Energy is calculated using Equation 1

**Procedure**: Find_Mappings
**Input**: Loop $l$
**Output**: Set of mapping solutions $S = \{(Z_1, E_1), (Z_2, E_2), \ldots, (Z_n, E_n)\}$ such that for each possible loop buffer depth vector $Z_i$, $E_i$ is the minimal energy for that configuration

Begin
/* 2 base solutions */
$S \leftarrow \{(\text{no loop buffer, Energy}(l, \text{unmapped})), \cup \text{ depth vector for the complete loop, Energy}(l, \text{mapped})\}$

/* Find the solutions of the children */
let $\{c_1, c_2, \ldots, c_n\} \leftarrow \text{children of } l$
for $i = 1 \text{ to } n$
do
| $Sol_i \leftarrow \text{Find_Mappings}(c_i)$ |
end for

$\text{Filter_Solutions}(S, Sol_1 \times Sol_2 \times \ldots \times Sol_n)$
End

**Procedure**: Filter_Solutions
**Input**: Set of Solutions $S$
**Input**: $Sol_1 \times Sol_2 \times \ldots \times Sol_n$
**Output**: Updated set of Solutions $S$

Begin
/* Combine the solutions of the children */
/* filtering out the unnecessary solutions */
for all $(sol_1, sol_2, \ldots, sol_n) \in Sol_1 \times Sol_2 \times \ldots \times Sol_n$
do
| let $(Z, \text{energy\_current})$ $\leftarrow$ New solution by combining $sol_1, sol_2, \ldots, sol_n$ and rounding to the next power of two for each buffer depth. |
if a solution exists in $S$ with depth vector $Z$ then
| let $(Z, \text{energy\_optimal})$ $\leftarrow$ Solution in $S$ with depth vector $Z$ |
if \text{energy\_current} < \text{energy\_optimal} then
| $S \leftarrow S \cup \{(Z, \text{energy\_optimal})\} \cup \{(Z, \text{energy\_current})\}$ |
| if $S$ $\neq S$ then
| $S \leftarrow S \cup \{(Z, \text{energy\_current})\}$ |
end if
end if
end for
End

**Procedure**: Find_Clustering
**Input**: Set of Loops Mapped to the Loop Buffer $L$
**Output**: Clustering solution $S = \{(f_{u_1}, f_{u_2}, \ldots, f_{u_n})\}$

Begin
/* Initial solution: each FU in separate cluster */
let $S_{\text{best}} = \{(f_{u_1}), (f_{u_2}), \ldots, (f_{u_n})\}$
let $E_{\text{best}} = \text{Energy}(S_{\text{best}})$

loop
for all $(\text{Cluster}_{u_1}, \text{Cluster}_{u_2}) \in S_{\text{best}} \times S_{\text{best}}$ do
| let $S_{\text{tmp}} \leftarrow S_{\text{best}} \setminus \{\text{Cluster}_{u_1}, \text{Cluster}_{u_2}\}$ |
| if $\text{Energy}(S_{\text{tmp}}) < E_{\text{best}}$ then
| $E_{\text{best}} = \text{Energy}(S_{\text{tmp}})$ |
| $S_{\text{current}} \leftarrow S_{\text{tmp}}$ |
end if
end for
if $S_{\text{current}}$ changed then
| $S_{\text{best}} \leftarrow S_{\text{current}}$ |
else
| exit /* No change. Fixed point. We’re done. */
end if
end loop
End

C. Clustering Exploration Algorithm

The $\text{Find_Clustering}$ function in Algorithm 1 finds, for a given mapping of loops, the best clustering (what functional units are in what cluster). Instead of generating all possible configurations and evaluating the energy with Equation 1, we use a best-fit approach.

Initially, each functional unit has its own cluster. Then, in every step of the algorithm we merge the two clusters that give the highest gain in energy, by trying all combinations of two clusters. We stop, when at a certain point, merging clusters will make the total energy go up.

This algorithm is based on the trade-off between Local Controller overhead and NOP overhead. The LC overhead is high when many small clusters are present, since every LC is accessed every cycle. When few but wider clusters are selected, the loop buffer partitions will contain a lot of useless NOP instructions. Even if only one FU is active in a certain cluster, the whole line needs to be fetched.

We have compared our best-fit approach with the exhaustive search of all configurations for VLIWs with a small number of FUs (8, 9 and 10). The average error is less than 2% in total energy consumption over all the benchmarks.

VI. RESULTS AND DISCUSSION

For our evaluation we have modeled in Wattch a realistic embedded VLIW processor from the TI C6x processor series [3], with 8 functional units (8 issue slots) and an instruction width of 256 bits with 32 bit operations. This processor has a clustered datapath with two identical clusters of 4 functional units. The datapath clustering and instruction clustering do not need to be the same. In general this is even a bad idea, as we will see from the results. Using the compiler of the Trimaran tool suite [22], applications were mapped onto this processor model and simulated to generate the statistics.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ADPCM decode</td>
<td>Audio</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>ADPCM encode</td>
<td>Audio</td>
<td>2x1x3x1x1</td>
<td>32x8x4x4x4</td>
</tr>
<tr>
<td>AES</td>
<td>Encryption</td>
<td>3x2x1x1x1</td>
<td>32x8x8x16x8</td>
</tr>
<tr>
<td>Blowfish encode</td>
<td>Encryption</td>
<td>1x3x3x1</td>
<td>64x32x16x16</td>
</tr>
<tr>
<td>JPEG decode</td>
<td>Image</td>
<td>16x1</td>
<td>64x64x32</td>
</tr>
<tr>
<td>EPIC</td>
<td>Image</td>
<td>2x2x3x1</td>
<td>16x8x4x16</td>
</tr>
<tr>
<td>g721 decode</td>
<td>Audio</td>
<td>2x2x3x1</td>
<td>64x32x16x64</td>
</tr>
<tr>
<td>g721 encode</td>
<td>Audio</td>
<td>2x2x3x1</td>
<td>64x32x16x64</td>
</tr>
<tr>
<td>gsm decode</td>
<td>Audio</td>
<td>3x2x3</td>
<td>32x8x16</td>
</tr>
<tr>
<td>gsm encode</td>
<td>Audio</td>
<td>3x2x3</td>
<td>32x8x16</td>
</tr>
<tr>
<td>H.263</td>
<td>Video</td>
<td>3x4x1</td>
<td>64x64x32</td>
</tr>
<tr>
<td>mesa</td>
<td>3D graphics</td>
<td>2x2x4</td>
<td>32x32x16</td>
</tr>
<tr>
<td>MPEG2 decode</td>
<td>Video</td>
<td>7x1</td>
<td>64x16</td>
</tr>
<tr>
<td>Rasta</td>
<td>Speech Recogn.</td>
<td>3x4x1</td>
<td>32x16x16</td>
</tr>
<tr>
<td>SHA</td>
<td>Encryption</td>
<td>4x4</td>
<td>8x8</td>
</tr>
<tr>
<td>Snake</td>
<td>3D graphics</td>
<td>2x2x2x1x1</td>
<td>64x32x64x4x32</td>
</tr>
</tbody>
</table>

Table I shows the benchmarks we used, and the optimal loop buffer configuration for each benchmark. The depth and width of the partitions are represented as vectors (e.g. the width and depth of the first partition for the MPEG2 decode benchmark is 7 and 64 respectively). The depth of the first partitions is generally larger because for low ILP loops, only the first partitions
energy per access of the IL1 stays the same. This allows the fit-2 step to map more loops on the loop buffer (because it is now less expensive per access), changing the relative contribution of $E_{il1}$ and $E_{clusters}$. This trend continues during the first and second iteration. The third iteration of the clustering stage (cluster-3) resulted in the same clustering as the second iteration. This means convergence is reached and we can stop iterating here.

The last bar in the chart is the optimal mapping on a configuration where datapath clusters and instruction clusters are the same. This is clearly not optimal. Instruction clusters are typically much smaller, as can be seen in Table I. This is because the overhead per cluster (of adding the local controller) is smaller.

**B. MediaBench**

Figure 5 shows the normalized energy consumption in the instruction memory for the applications of the MediaBench [23] benchmark suit. The naming of the steps on the X-axis is the same as in the previous subsection. The average energy reduction is 18%, with peaks of 45% as compared to the optimal mapping on a non-clustered architecture (i.e. fit-1). Most of this 18% comes from 2 steps in the iteration: on average 10% comes from finding the optimal clustering after the initial mapping (from fit-1 to cluster-1), and 7% from the optimal mapping on the clustered loop buffer (from cluster-1 to fit-2).

An additional gain of 65% can be obtained when comparing fit-1 to the very simple mapping of loops currently used in existing processors (only mapping most inner loops), but this has already been presented in [24].

We ran three iterations of the exploration. After running the clustering stage a third time, it did not come up with a clustering that was better than before for any of the applications, except for djpeg. Here convergence was reached after the fourth iterations and covered 2 iterations (i.e. the fourth iteration of the clustering stage gave the same results as the second).

We have also evaluated our approach with a VLIW processor with 16 functional units (not shown in the graph). Here the gain was even more: 36% on average.

**VII. CONCLUSIONS**

This paper presents a loop buffer exploration methodology based on detailed analytical energy models for software controlled clustered loop buffers. An algorithm is presented to find a good loop buffer configuration and an optimal mapping for an application on the loop buffer. The knowledge the compiler has about the application is exploited to make the right decisions. Different loop buffers can be evaluated allowing the user to make the trade-off between the size (area) of the loop buffer and the consumed energy.

The algorithm is demonstrated using MediaBench, giving an average of 18% reduction in energy consumption, with peaks of 45% for some applications.

After all these optimizations the original contribution of the instruction memory to the total processor power has gone down from 50% to 10%, for a TI C64x architecture modeled in
Wattch. This means a software controlled clustered loop buffer and compiler support for this, is something future low power embedded VLIWs cannot do without.

ACKNOWLEDGMENTS

This project is partially supported by the Fund for Scientific Research - Flanders (FWO) through projects G.0036.99 and G.0160.02, and by the IWT through MEDEA+ project A502 MESA.

REFERENCES


