Instruction Buffer with Limited Control Flow and Loop Nest Support

Vladimír Guzma, Teemu Pitkänen, Jarmo Takala
Department of Computer Systems
Tampere University of Technology
FI-33720 Tampere, Finland
Email: name.surname@tut.fi

***Abstract*—In this work, we present a minimalistic, energy efficient implementation of instruction buffer. We use loop detection and execution trace analysis to find most commonly executed loops in already scheduled application and tailor instruction buffer size to the size of most commonly executed loop(s). In addition to our previous work, we allow buffering of loops with limited control flow (early exit from the loop or early return to the beginning of the loop). We also show how analysis of loop nests can decrease the number of times loop body is copied from memory into the buffer. Our results show that in case of favorable loop nest, we can execute all but initial loop iterations from the instruction buffer, keeping instruction memory in the deselect mode.**

I. Introduction

Repeat buffers and instruction caches are known methods to improve performance of computing systems, avoiding penalties introduced by memory hierarchies. In the area of embedded systems, buffers and caches can also contribute to reduction of overall energy requirements of the memories. In particular, many DSP applications exhibits large amount of instruction level parallelism with relatively simple control structures. This, when compiled for wide parallel processor (VLIW [1] or TTA [2] for example) often results in relatively small loops executed for most of the algorithm execution.

Different methods for extending memory hierarchies, leading to reduction in energy requirements were surveyed in [3]. In particular, with growing size of the memory energy requirements increase. Therefore, storing bodies of loops in smaller memories reduces energy requirements of the hierarchy. Instruction buffers, as present in AT&T DSP16X series, are an example of such method. In order to use such an instruction buffer with size fixed in architecture, compiler needs to be aware of the presence of instruction buffer. Also, loop transformations may be required to fit the loops into the buffer. Natural extension of this mechanism is storing already decoded instructions in the buffer [4]. This saves cost of repeated decoding of the instructions in loop body.

In [5], the energy consumption of processor is analyzed, when using addressed, compiler controlled, fixed size instruction register file. Another approach to the problem is the use of loop caches, such as [6]. Used as a L0 cache, they can store typical small loops. Typical use of caches does not require assistance from the compiler or the designer. Although in case of embedded system, customizing loop cache size for particular application leads to better results. As with the instruction buffers, loop caches can also be extended to store cached instructions already decoded, such as *Decode Filter Cache* described in [7], and *Decoded Instruction Buffer* described in [4].

Another popular method is use of instruction scratch pad memories [8]. Their use, unlike with caches, require mapping of program parts by the compiler or the user.

Difference between instruction repeat buffer and simple cache is in how the instruction to be executed from cache or buffer are selected. Cache entries hold their instruction addresses, and during execution the address is compared with program counter and instruction is executed from cache, if address matches. This process happens automatically, unless the cache is locked via special instruction. In case of buffer, the addresses of individual instructions are not stored. Therefore, all the content of buffer is executed, unless control is informed to leave the buffer and continue from specific instruction memory address. This result in lower energy consumption of instruction buffer implementation.

In our previous *proof of the concept* work [9], we considered only the simplest loops without control flow and without giving consideration to the loop nests.

In this paper, we extend the instruction buffer control to allow for early exit of the buffer, or early return to the beginning of the buffer. This addition allows for certain loop structures to be executed from the buffer (as described in Section II), increasing percentage of application instructions executed from the instruction buffer.

We also address another recurring problem of our early implementation. In case of nested loops, the body of the most nested loop was stored in the buffer. However, when the program execution finally left the loop body, returned to the outer loop and execution eventually returned to the most nested loop, copying of the loop body from memory into the buffer was performed again, wasting energy. In this work we show situations when this waste of energy can be avoided, and loop content invalidated only after exiting outer loop(s).

The rest of this paper is organized as follows: in Section II we show types of loops we find favorable for storing in the instruction buffer. Section III outlines our implementation of the control of the instruction buffer. Section IV describes our approach for detecting loops to be stored in buffer. Section V

provides information of our experimental setup and Section VI shows our results. Finally, Section VII provides final words and outlines possibilities for future work.

II. Characteristic Loop Types

The optimizing compilers provide wide range of code transformations to increase performance of resulting code. We use *Low Level Virtual Machine* (LLVM) [10] as the compiler front-end. The resulting binaries, when reverse engineered using Control Dependence Graph (CDG) [11], shows presence of several typical loop types. By constructing the CDG of the already scheduled binary, we get advantage of knowledge of loop positions in the generated binary.

In the Control Dependence Graph (static representation of the code, unlike Control Flow Graph that is dynamic representation), there are several types of nodes. The most basic type contains single basic block, without a control flow. In the Figures 1, 2 and 3, the nodes containing the range of numbers indicate a range of instruction addresses that the node representing the basic block contains.

Another type of node, denoted as *Predicate* with range of instruction numbers, indicate the basic block ending with a predicate. Based on the evaluation of the predicate, either *True* or *False* branch will be taken, if the execution of some part of the code depends on the predicate.

Remaining types of nodes do not have direct correspondence to the instructions in the code. Nodes denoted as *Region* (and specific type of Region node denoted as *Start*) are placeholders, grouping together subtrees with the same control dependence.

Their meaning is as follows: If execution reaches the Region node, all of the child nodes (and subtrees rooted at child nodes) will be executed, in no predefined order. Therefore, in the Figure 1 for example, control starts at node Start, and nodes *0-6*, *107-115* and *Entry 1* will be executed.

Another of the helper nodes is denoted as *Entry*. This node is specific type of Region node. The subtree rooted at node Entry forms a loop. The last of the node types presented in our graphs is denoted as *Close*. This is virtual placeholder for the loop repeat. In presence of the loop with several *continue* statements, all the predicates that leads to repeat of the loop has edge towards the Close node, which then has loop close edge back to the respective Entry node.

In our earlier work [9] we explored benefits of introducing instruction buffer sized specifically for the most often used simple loop. This loop type, visible in Figure 1 as starting from node *Entry 3*, contained only single basic block, ending with the loop predicate test. When test evaluated to be *False*, the loop was repeated. After exiting the loop, we invalidated the content of the buffer, so the next candidate loop can be buffered.

This approach has some disadvantages. One, clearly visible from the Figure 1 is that in presence of the loop nest, with single loop at the deepest level, the content of the loop will be repeatedly copied, after leaving the loop starting at *Entry 3*, the control will return to the outer loop starting with *Entry 2* and the loop at *Entry 3* will be re-entered. This provides unnecessary lost of energy, as each time the loop is entered, the first iteration will be executed from the memory while content of the loop will be copied to the buffer. Similar situation can be seen also on the Figure 2, with loop starting with *Entry 3* inside the loop starting with *Entry 2*, without any sibling loops.

Therefore, this particular type of loop nests needs to be detected and handled differently. Once the simplest loop is found, the parent node of the loop Entry node is tested. If the node is Entry as well we check whether or not our simple loop has sibling node denoted as Entry too. This would indicate two loops side by side inside the outer loop. If this is not the case, we can avoid unnecessary copying, the buffer invalidate flag can be moved further away from the loop. In case pictured in the Figure 1, there are three possible locations for invalidate flag.

1) The invalidate flag can be placed right after the end of the most nested loop, virtually in the basic block succeeding execution of the loop (Entry 3) present in the outer loop (Entry 2).
2) The invalidate flag can be placed outside the middle loop. In this case, the flag can be placed in the body of loop denoted with Entry 1, after exiting loop starting with Entry 3.
3) The invalidate flag can be placed outside the outer loop, effectively in the wind up code in the main function of the program.

In case pictured in the Figure 2, only first two choices are available. With placing the invalidate marker in the body of loop starting with Entry 2, or in the basic block succeeding the loop with Entry 2.

Another, often generated loop type, contains limited control flow. In practical terms, the loop is often written as to be executed infinitely, but it contains one or more tests which provide early exit of the loop, or it is executed for limited number of iterations with early *continue* statements. Example of such a loop can be seen on Figure 3, with the loop starting at node *Entry 9*, containing two basic blocks ending with predicate.

This type of loop can also be buffered, pending buffer control ability to evaluate if the branch inside the loop buffer points to the instruction at the beginning of the buffer or other address. Our implementation of this mechanism is presented in Section III on Figure 4.

The case pictured on Figure 3 show two additional loops. Starting with *Entry 7* and *Entry 8*. Both those loops can be buffered as was presented in our earlier work [9]. Adding loop starting with *Entry 9* therefore increases the number of instruction executed from the buffer.

III. Instruction Buffer Control

The Figure 4 outlines our implementation of state machine controlling instruction buffer. For controlling of instruction buffer, we choose to add two bits to the instruction word.

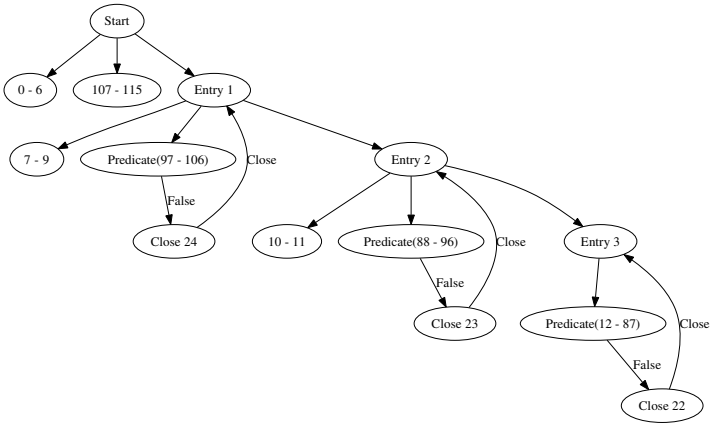

Fig. 1. Simple loop nest in DCT 8x8.

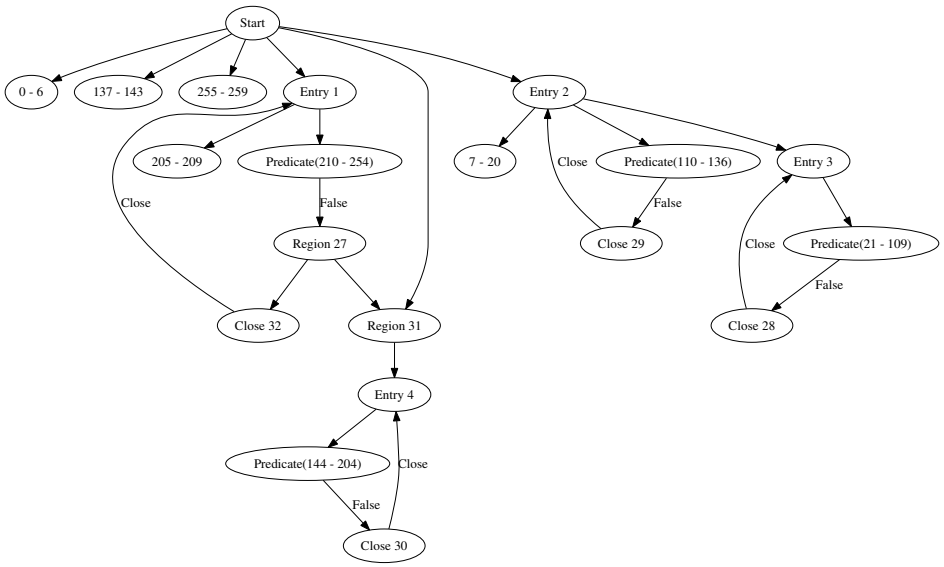

Fig. 2. Structure of Viterbi.

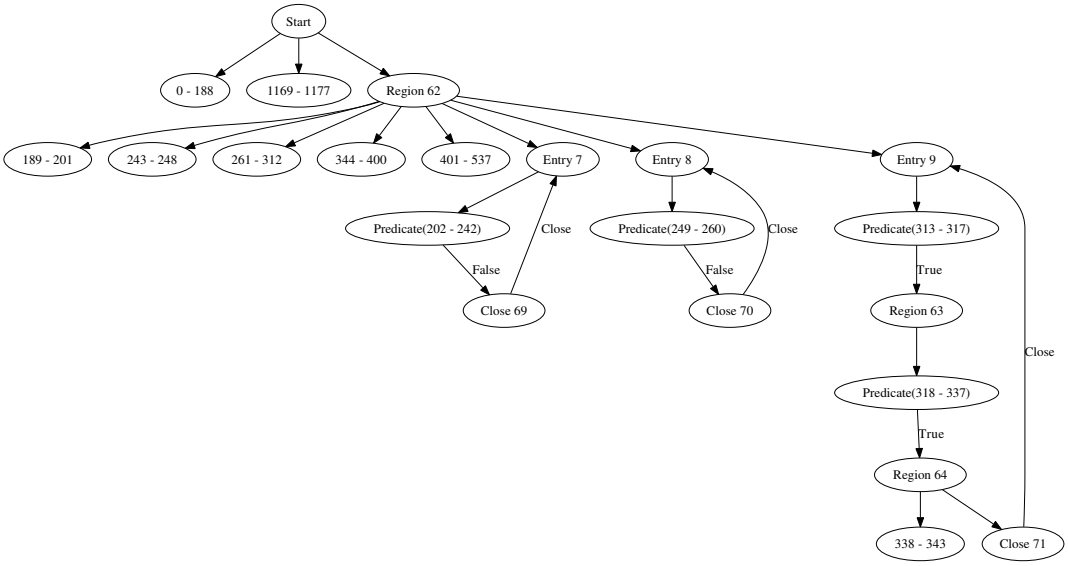

Fig. 3. Part of structure of ADPCM.

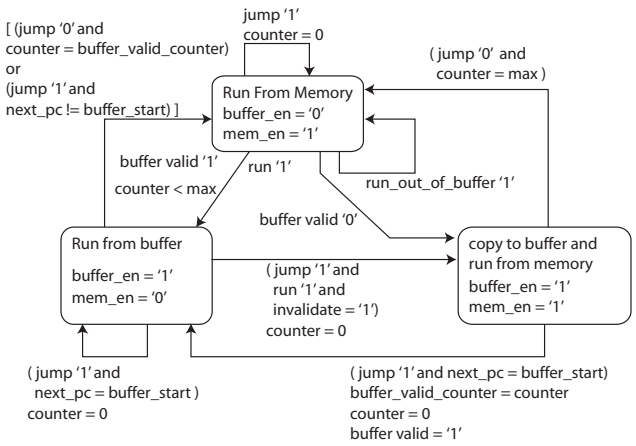

Fig. 4. State machine controlling instruction buffer.

The default option is to *execute from memory*. The instructions that should be stored in the instruction buffer are marked with different flag, *execute and copy*. If the content of the buffer is invalid, executing such an instruction from memory triggers simultaneous copying of the instruction into the instruction buffer. Once buffer is full, all further execution comes from the buffer, until it is exited. Another flag, *invalidate*, is to indicate that the content of the buffer becomes invalid. On the next occasion when the *execute and copy* marked instruction is fetched from the memory, it will be stored in the buffer again. Finally, the *execute and invalidate* flag will force buffer to invalidate the content if exiting the buffer. This allows for successive loop to be stored without any instructions in between.

This approach together with our state machine allows for several possible scenarios.

- Simple loop ending with conditional jump to the beginning of the loop is fully stored in the buffer
- Simple loop ending with conditional jump to the beginning of the loop is larger then the buffer and only part of it is stored in buffer
- Loop contains several conditional jumps, either causing loop to repeat before reaching end of the loop or exit the loop

In order to allow for conditional jumps to the beginning of the loop, or exiting loop, the starting address of the loop is stored when the buffer is entered. Upon detecting the jump, the destination address of jump is compared with the address of the beginning of the loop.

Possible alternative for controlling the buffer is to add buffer control register to the global control unit. Visible to the compiler and programmer, control of the buffer could be implementing by simply writing control commands to this register as simple moves within the desired instruction. However, in the presence of tightly scheduled loops, there may not be free move slot to fill in with dedicated command in required instructions. In particular, the beginning instruction of the loop is often fully packed. In such case, recompilation of the application would be necessary, possibly leading to altered schedule, with different loop size and need for repeated analysis and profiling. This approach may be worth considering in case of compressed instructions, where the addition of two bits is significant compared to compressed instruction width.

IV. Detection of Loops and Profiling

In order to find loops most likely to benefit from the use of instruction buffer, we combine analysis with trace information. We start with application binary, compiled for the selected architecture. By analyzing it, we obtain first Control Flow Graph [12], from which in turn we construct Control Dependence Graph (CDG) [11]. We favor CDG as it provides easy way how to detect loops, by recursive application of strongly connected components detection, and simple way how to detect favorable loops.

The simplest kind of loop that can be stored in instruction buffer contains only three nodes. Node *Entry*, which is virtual node without actual equivalent in the application code, denotes the location of the loop. Another member of the loop is the *Predicate* node. In our implementation, the basic block and predicate are combined. Therefore, the Predicate node contains the body of the loop and ends with computation of loop predicate and the jump. Third node we find in the simplest loop is the *Close* node. Which again is the virtual node.

In order for loop to be without any additional control, there can only be one outgoing edge from the Entry to the Predicate. And upon evaluation of Predicate, there is one edge from Predicate to the Close node and back edge from the Close to the Entry. Cases with more than single outgoing edge from Predicate node indicate if/else statement inside the loop body, not the early exit or return to the beginning of the loop we look for. Examples of such a loops are visible on Figure 1 with Entry 3, Figure 2 with Entry 3 and Entry 4, as well as on Figure 3 with Entry 7 and Entry 8.

More complex loop type we look for, containing early exits from the loop or early returns to the loop beginning, have slightly more complex structure. Again, only one edge out of the Entry node is allowed, leading to a first Predicate. From the

Predicate however, there can be only edge to the Region node with child nodes with body of basic block and Close node, or another Predicate with only one outgoing edge. Example of such a loop is shown on Figure 3, starting with node Entry 9. Detection of such a loops allows for storing them in the instruction buffer.

Another feature we look for in the generated CDG is the presence of favorable loop nests. In case of loop nest, the Entry node of the most nested loop is child node of the Entry node of the outer loop. In order for most nested loop to be stored in the instruction buffer, and the buffer invalidate marker to be present outside the outer loop, the outer loop must be without any additional control. In practice, this transfer to outer loop's Entry node to have only one child node with the Entry (only one loop nested in the outer loop). Presence of more than single Predicate as the child of the Entry node of the outer loop does not create a problem. This kind of loops are present in Figure 1, marked with Entry 2, and even Entry 1, as well as in Figure 2 marked with Entry 2.

Once we have detected the favored loops, most deeply nested as well as ones containing limited control flow, we can find loop starting instruction address, as well as instructions in which the conditional jumps are present.

All loops, however, are of no equal importance to our effort to decrease energy consumption of the memory. Only loops with sufficient high repeat count are good candidates for storing in the instruction buffer since the process of storing the instructions into the instruction buffer while reading them from memory and executing, as it needs to be done in the first iteration of the loop, brings additional energy cost of writing into the buffer.

Therefore, we use instruction trace obtained by running the application in the simulator to collect information about how often the jump instruction detected in the loop body is followed by the starting instruction of the loop - loop iteration, and vice versa, how many times the jump instruction is followed by the instruction outside the loop body - loop exit.

Collecting such information for all of the detected loops, allows us to select the loops with higher percentage of executed instructions as candidates for storing in the buffer, and informs us of the average number of times loop is executed before it is exited.

The accuracy of the collected information of course depends on the quality of the input provided when collected the execution trace. It is, however, possible to select a range of possible inputs and collect a range of separate execution traces, to improve accuracy of the collected loop iteration statistics.

V. Experimental Setup

To demonstrate our approach for use of the instruction buffer with limited control flow as well as advantage of invalidating buffer after the exit of the outer loop, we picked three example applications with favorable structures of the CDG, as show in the Figure 1, 2 and 3. In case of *DCT8x8* and *Viterbi* we show the impact of delaying the buffer invalidation, reducing number of copying into the buffer, and in the case of *adpcm* we insert into the loop buffer loop with two early exits.

TABLE I
Collected statistics for DCT 8x8. Number of executed instruction in whole application as well as number of instructions executed inside loop body and number of instruction copied into the buffer.

Total cycles	Loop iteration	Loop iteration (%)	Loop copying	Loop copying (%)
20033	17024	84.9 %	2432	12.1 %

TABLE II
Energy results for DCT 8x8, with memory of 128 instructions 270 bits wide and instruction buffer of 76 instructions.

Buffer status	Buffer Control (mW)	Buffer (mW)	Memory (mW)	All (mW)
No buffer	0	0	65.23	65.23
Simple buffer	3.19	10.5	15.69	29.38
2 loops buffer	3.19	10.9	11.33	25.42
3 loops buffer	3.19	11.3	7.24	21.73

We used publicly available TCE toolset [13], which allows for processor designs based on Transport Triggered Architecture template [2]. This allowed us to easily modify control of the previously designed TTA processor, adding the instruction buffer of required size for particular application, as well as state machine to implement control of the buffer. We also added two bits to the instruction word binary for controlling the buffer, as outlined in Section III.

To collect the actual power data, we synthesized the processor with Synopsys Design Compiler and ran gate level simulation with Mentor Modelsim, from which the gate activity was acquired for the Synopsys Power compiler. We used 130nm technology library.

VI. Results

We used the same baseline processor architecture in all of our tests. The number of resources, in particular interconnection network, required instruction width of 268 bits, 270 with additional buffer control bits. In Table I we present the results of the collected statistics for the DCT8x8 application, as outlined in Section IV. The fact that 84.9 % of the executed instructions during a test run are from within the most deeply nested loop gives us indication that the use of instruction buffer can bring significant energy savings.

In Table II, we show collected results from the gate level simulation. The line denoted *No buffer* indicate our baseline case, where no instruction buffer is in use. In this case, only the power consumed by the memories is shown.

The line denoted as *Simple buffer* show results for our most simple control mechanism of the instruction buffer, with size of 76 instructions. In this case, loop body is stored in the buffer and once the loop is exited, content of the buffer is invalidated. Therefore, if the application enters loop again, later in the execution, the content of the loop will need to be copied into the buffer again.

TABLE III
COLLECTED STATISTICS FOR VITERBI. NUMBER OF EXECUTED INSTRUCTION IN WHOLE APPLICATION AS WELL AS NUMBER OF INSTRUCTIONS EXECUTED INSIDE LOOP BODY AND NUMBER OF INSTRUCTION COPIED INTO THE BUFFER.

Total cycles	Loop iteration	Loop iteration (%)	Loop copying	Loop copying (%)
1536148	1456752	94.8 %	46992	3 %

TABLE IV
ENERGY RESULTS FOR VITERBI, WITH MEMORY OF 2048 INSTRUCTIONS 270 BITS WIDE AND INSTRUCTION BUFFER OF 89 INSTRUCTIONS.

Buffer status	Buffer Control (mW)	Buffer (mW)	Memory (mW)	All (mW)
No buffer	0	0	77.26	77.26
Simple buffer	4.27	10.34	8.73	23.34
2 loops buffer	4.31	10.5	6.55	21.36

TABLE V
COLLECTED STATISTICS FOR ADPCM. NUMBER OF EXECUTED INSTRUCTION IN WHOLE APPLICATION AS WELL AS NUMBER OF INSTRUCTIONS EXECUTED INSIDE LOOP BODY AND NUMBER OF INSTRUCTION COPIED INTO THE BUFFER.

Total cycles	Loop iteration	Loop iteration (%)	Loop copying	Loop copying (%)
84108	13082	15.5 %	1600	2 %

TABLE VI
ENERGY RESULTS FOR ADPCM WITH MEMORY OF 2048 INSTRUCTIONS 270 BITS WIDE AND INSTRUCTION BUFFER WITH 32 INSTRUCTIONS.

Buffer status	Buffer Control (mW)	Buffer (mW)	Memory (mW)	All (mW)
No buffer	0	0	77.26	77.26
Early exit buffer	2.26	1.25	70.00	73.51

We can already see from our results that the addition of the buffer requires additional power for the control logic of the buffer, as well as for buffer itself. However, the power required by the memory dropped dramatically, from 65.23 mW to 15.69 mW. As a result, the overall power required dropped from the 65.23 mW to the 29.38 mW, some 45 % of power without the buffer.

We utilize advantage of favorable loop structure in this case and provide two more test cases. In line denoted with *2 loops buffer*, we store again most nested loop into the same buffer of 76 instructions, but we invalidate the buffer content only after the outer loop is exited. Eventually reducing number of times the content of the loop is copied into the buffer. Our results show that in this case, the power required by buffer control remains same as previously, while buffer power increases slightly since the buffer content is read more often. Memory power requirement, however, drops compared to the simple buffer case to 11.33 mW, resulting in whole power of 25.42 mW, some 38 % of our baseline.

Since our approach of issuing invalidate buffer marker outside the outer loop can be pushed up through loop nest, as long as the structure of loop nest allows it, we provide another case. In the line denoted *3 loops buffer* we move buffer invalidate marker outside the outer loop. This is essentially situation that mimics single loop in the main program. The content of the most nested loop will be copied into the buffer just once and invalidated after exit of the outer most loop. This case brings highest reduction in the memory energy with 7.24 mW, compared to 65.23 mW when no buffer is used at all. Overall power required drops to 21.73 mW, some 33 % of the baseline.

In the Table III we present collected information about the most often executed loop in the *Viterbi* case. The Table IV shows the results of a gate level simulation. In the line marked as *No buffer* we show results of architecture without the buffer implemented, providing for baseline case. In the line marked as *Simple buffer*, only single most nested loop is stored in the buffer of 89 instructions, and its content is invalidated once the loop is exited. Due to fact that the loop is executed for some 94.8 % of all instruction, this case already provides significant savings. Memory power drops from 77.26 mW to some 8.73 mW, some 11 % of former. With added cost of buffer control and buffer, the sum of power required grows to 23.34 mW, some 30 % of the baseline case.

As in case of DCT8x8, Viterbi also contains nested loop of favorable structure. Therefore, we provide results for another test, line marked with *2 loops buffer*, where the buffer invalidate marker is pushed outside the outer loop. This reduces number of copies into the buffer. Results show slight improvement compared to previous case, with power of memory dropping to 6.55 mW compared to Simple buffer's 8.73 mW. This result in overall power of 21.36 mW, some 27 % of the baseline case.

Above two cases show how our approach to invalidate buffer content further away from the buffered inner loop contributed to lowering of the power required by the memory. In particular, the DCT8x8 case allows for all, except first, executions of the inner loop of the loop nest from the buffer. The Tables V and VI provides example of another approach presented in this paper. First, Table V shows collected statistics about the loop containing two early exits. This loop, however, provides relative small coverage; only 15.5 % of program execution is from within this loop. The Table VI show results of our gate level simulation of this case. The *No buffer* shows baseline case, with a memory power of 77.26 mW. The line *Early exit buffer* shows the power of the memory dropping to some 70 mW. Buffer size in this case was 32 instructions. This relative small drop is worsened by additional power required for a buffer control and a buffer itself. Total power therefore is 73.51 mW, some 95 % of the baseline case.

While this case shows that approach of storing the loop with specific type of control flow in the buffer, it also shows the borderline case. The small coverage of buffered loop, just some 15 %, provide some drop in the memory power, but the increase of the power for the buffer control and the buffer

makes overall saving minimal.

VII. Conclusion

In this work, we show two methods how to improve utilization of the instruction buffer and reduce the memory power. When favorable loop structures are detected, the trace information can be used to collect information about how often the particular loops are executed. This allows us to select the loops with most coverage, and create instruction buffer specifically sized for the given loops. While storing loops with limited control flow structures, early exit or early return to the beginning of the loop requires slightly more complicated buffer control in order to compare the address of a jump with the address of first instruction in the buffer. Second approach of careful positioning of loop invalidate marker do not require storing of the loop start address in the buffer control.

Our results show, that for loops with favorable structure, large portion of copying the body of the loop into the buffer can be eliminated, reducing power required by the memory to minimum.

Furthermore, the instruction buffer may store instructions in the decoded form, saving instruction decoding step when executing from the buffer and consequently, saving power. Another possibility is tied with instruction compression. Instructions can be decompressed before storing in the buffer, saving additional power required for repeated decompression of the loop body.

References

[1] J. A. Fisher, "Very long instruction word architectures and the ELI-512," in *ISCA '83: Proc. 10th int. symp. on Computer architecture*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1983, pp. 140–150. [Online]. Available: http://portal.acm.org/citation.cfm?id=801649

[2] H. Corporaal, *Microprocessor Architectures: From VLIW to TTA*. John Wiley & Sons, 1997.

[3] L. Benini, A. Macii, and M. Poncino, "Energy-aware design of embedded memories: A survey of technologies, architectures, and optimization techniques," *ACM Trans. Embed. Comput. Syst.*, vol. 2, no. 1, pp. 5–32, 2003.

[4] R. S. Bajwa, M. Hiraki, H. Kojima, D. J. Gorny, K. Nitta, A. Shridhar, K. Seki, and K. Sasaki, "Instruction buffering to reduce power in processors for signal processing," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 5, no. 4, pp. 417–424, 1997.

[5] W. J. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R. C. Harting, V. Parikh, J. Park, and D. Sheffield, "Efficient embedded computing," *Computer*, vol. 41, pp. 27–32, 2008.

[6] J. Kin, M. Gupta, and W. H. Mangione-Smith, "The filter cache: an energy efficient memory structure," in *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 1997, pp. 184–193.

[7] W. Tang, R. Gupta, and A. Nicolau, "Power savings in embedded processors through decode filer cache," in *DATE '02: Proceedings of the conference on Design, automation and test in Europe*. Washington, DC, USA: IEEE Computer Society, 2002, p. 443.

[8] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: design alternative for cache on-chip memory in embedded systems," in *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*. New York, NY, USA: ACM, 2002, pp. 73–78.

[9] V. Guzma, T. Pitkänen, and J. Takala, "Reducing instruction memory energy consumption by using instruction buffer and after scheduling analysis," in *Proc. Int. Symp. System-on-Chip*, Tampere, Finland, Sep. 29–30 2010, pp. 99–102.

[10] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Generation Optimization*, Palo Alto, CA, March 20–24 2004, p. 75.

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, 1991.

[12] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006. [Online]. Available: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20\&path=ASIN/0321486811

[13] P. Jääskeläinen, V. Guzma, A. Cilio, and J. Takala, "Codesign toolset for application-specific instruction-set processors," in *Proc. SPIE Multimedia on Mobile Devices*, San Jose, CA, Jan. 29–30 2007, pp. 65 070X–1 – 65 070X–11.