SOFTWARE-CONTROLLED INSTRUCTION PREFETCH BUFFERING FOR LOW-END PROCESSORS

MUHAMMAD YASIR QADRI[†]
School of CSEE, University of Essex, Wivenhoe Park,
Colchester, United Kingdom CO4 3SQ
[†] *yasirqadri@acm.org*

NADIA N. QADRI[§]
COMSATS Institute of Information Technology, Wah Campus,
Pakistan
[§] *nadianqadri@googlemail.com*

MARTIN FLEURY[††]
School of CSEE, University of Essex, Wivenhoe Park,
Colchester, United Kingdom CO4 3SQ
[††] *fleum@essex.ac.uk*

KLAUS D. McDONALD-MAIER[§§]
School of CSEE, University of Essex, Wivenhoe Park,
Colchester, United Kingdom CO4 3SQ
[§§] *kdm@essex.ac.uk*

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

This paper proposes a method of buffering instructions by software-based prefetching, which, with a minimum of logic and power overhead, can be employed in low-end processors to improve their instruction throughput. Low-end embedded processors do not employ cache for mainly two reasons i.e. the overhead of cache implementation in terms of energy and area is considerable, and, as the cache performance primarily depends on the number of hits, increasing misses could cause the processor to remain in stall mode for a longer duration, which in turn makes a cache become more of a liability than an advantage. In contrast, the benchmarked results for the proposed software-based prefetch buffering shows a 5–10% improvement in execution time, along with at least a 4% reduction in energy-delay-square-product (ED^{2}P) and a maximum reduction of 40%. The results show that the performance and efficiency of the proposed architecture scales with the number of multicycle instruction. The benchmarked routines tested are widely deployed components of embedded applications.

Keywords: Compiler control words; embedded processors; instruction prefetch buffering

1. Introduction

Our paper presents an instruction prefetching architecture that strives to reduce overall instruction latency and at the same time achieving stall-free pipeline operation using control words that are added to instructions at compile time by a compiler or by some other suitable software tool when code is generated. The architecture also seeks to reduce energy consumption in a way that is compatible with the previous two aims. The control words, which specify the addresses of instructions to be fetched, are placed in code memory one instruction ahead (assuming a two-stage pipeline), so that during execution the instruction or instructions required in the next cycle can be fetched seamlessly. Key to stall-free operation during conditional branches is intervention at the micro-architectural level through provision of a dual instruction buffer. This buffer is also of assistance when an instruction is encountered that takes-up two words, one of which is a target address.

Pre-fetch instruction can be contrasted with cache memories. Cache memories were primarily introduced to bridge the gap between processor and memory performance. With processors operating at remarkable speeds and DRAM operating at a fraction of those speeds, multi-level cache hierarchies provided a viable solution to maintain the trend that follows Dave House's revision of Moore's law, which predicts the doubling of computer performance every 18 months [1]. Cache memories have been widely used in microprocessors for faster data transfer between the processor and main memory. However, energy-constrained processor architectures typically do not employ a cache for mainly two reasons: 1) the overhead of cache implementation in terms of energy consumption and area is considerable; and 2) as the cache performance primarily depends on the number of hits, increasing misses can cause the processor to remain in a stall mode for longer periods, which can make the cache turn into a liability rather than an advantage [2]. In fact, research [3] [4] has shown that caches may consume up to 50% of a microprocessor's total energy. Therefore, in this paper, all the designs do not employ a cache, as is common for low-end embedded processors. As an alternative to the typical cache structure, software-controlled prefetching has been investigated over the years [5] in various ways explored in Section 2.

The proposed prefetch instruction buffering architecture takes advantage of the single-cycle memory latency supported by many low-end processors due to on-chip code and data memories. As an example, one such processor is the 8-bit Atmel AVR [6] microcontroller family, with a two-stage instruction pipeline in which many of the instructions take a single cycle to execute. However, instructions such as conditional and unconditional branches, function calls and returns, indirect loads and stores (using pointers) require more than one cycle to execute. The extra cycles include time for address calculation, while, in our scheme, software-calculated control words are placed in advance to reduce the number of cycles of such multi-cycle instructions. The proposed prefetch control-words not only reduce the number of cycles for such multi-cycle instructions by half (see Section 5) but also help to minimize the energy-delay-square product (ED^{2}P) i.e. an appropriate metric for energy performance trade-off analysis [7] [8]. The software-generated control words are inserted where required and a null control word can put the bus into a high-impedance state for minimal energy consumption. Use of control words implies that additional code memory storage is required, as some instructions consist of the original instruction code and an additional control code. Although on the one hand this type of software assisted prefetch buffering increases the code memory size, on the other hand it also results in an overall increase of the instruction throughput and greater prefetch accuracy.

An outline only of some of these ideas has been filed as a U.S. patent application [9], though without relevant prior research papers, consideration of the context, or performance results and analysis, as now occurs in this paper, which includes a longer description of the innovation and broadens the treatment. The dual buffer techniques should be applicable to low-end microprocessors and microcontrollers. What is required is that the processor cycle time should be greater than or equal to that of the associated data memory (i.e. the time to perform a memory read or memory write). The instruction memory read cycle time should be less than or equal to that of the processor. These conditions as restrictive because, for example, they are met by the AVR processor, which has on-chip code and data memory, allowing single cycle access. The AVR processor is widely deployed. For example, the AVR processor is embedded within the Arduino board, with 700,000 official Arduino boards in circulation in 2013 [10]. In the proposed modification, the instruction memory should also be capable of providing access to at least two locations in one cycle. Many processors also implement register files as dual-ported or multi-ported SRAM and, of course, (Video) VRAM is normally dual-ported DRAM.

The remainder of this paper is organized as follows. Section 2 considers related work on the topic of software guided prefetching. This work has augmented the normal prefetch instruction buffer and Section 3 considers the context in which additional buffering has been introduced in the past. Section 4 is a more detailed description of the proposed architecture and its principal features. The architecture has been implemented in reconfigurable hardware, which was then employed to benchmark the performance, as recorded in Section 5. Finally, Section 6 considers some of the implications of the new compiler-controlled architecture.

2. Related Work

A large body of the research on prefetch buffering focuses on complementing the multi-level cache hierarchy with prefetch buffers in order to reduce miss rates. The IBM Single Source Research Compiler for the Cell processor (the SSC Research Compiler) uses a software cache and buffers to prefetch the data [11]. However, for irregular references, as in the cases of indexed arrays, hash tables, and multi-level pointers, this type of prefetching also suffers from high miss rates. Chen et al. [12] proposed a scheme using run-time libraries, through which the compiler can identify the irregular references to target, and distribute each of the loops containing them into an address collection loop and a computation loop. In this way, DMA operations for prefetching are overlapped and the overhead of context switching is reduced for the cache miss handler. Another such hybrid or hardware/software prefetch scheme was proposed by Solihin et al. [13], in which, by exploiting a multi-threaded architecture, a thread analyzes the history of data accesses to predict future access with the help of an intelligent-memory processor residing in system main memory. However, for all of the above techniques, random memory accesses may result in increased miss rates [14].

Another approach that handles the issue of random access is to pre-execute the code by a thread and prefetch the data as per the requirements predicted in the pre-execution. One such scheme is proposed by Annavaram et al. [15] in which a Dependency Graph Pre-computation scheme (DGP) was used. When a load/store instruction that is likely to cause a cache miss enters the Instruction Fetch Queue (IFQ), a Dependency Graph Generator (DGG) follows the dependency pointers, which are already fetched from the previous instruction, to generate the dependence graph of those instructions yet to be executed. The dependency graphs generated by the DGG are then fed to a Pre-computation Engine (PE) that executes them to generate the load/store addresses early enough for timely prefetching. Another example of using helper threads for pre-execution can be found in a work by Luk [16].

A sophisticated approach was proposed by Zilles et al. [17], in which, to avoid cache misses and branch mispredictions associated with problem instructions, a code fragment called a speculative slice [18] is constructed that mimics the computation leading up to and including the problem instruction. However, Zilles et al. [18] generated these speculative slices manually. Therefore, adopting this technique for real systems would require some software-assisted method to accurately generate them. Although all of the above pre-execution techniques result in lower miss rates, in the case of random memory accesses, they require careful synchronization between the main thread and pre-execution thread. As instruction throughput has been the only metric generally employed to judge the performance of such methods, extra energy overhead by pre-execution threads has not been accounted for in most cases.

Branch Target Buffering is another approach that is closely related to our work. Branch Target Buffering combines branch prediction with a prefetch of target instructions into a Branch Target Buffer (BTB). However, branch prediction for modern architectures can consume up to 10% of the power of a processor [19]. Therefore, research has been undertaken to reduce this power. Kahn et al. [20] proposed two mechanisms, i.e. serial-BTB and filter BTB, through which power can be reduced. Both the techniques aim to buffer only necessary data; however, these methods introduce an extra delay. Therefore, for a 51% reduction in dynamic power consumption using both techniques, instruction throughput is decreased by 1.2%.

Gu et al. [21] proposed an instruction-cache architecture called Reduced One-

Bit Tag Instruction Cache (ROBTIC). In the said architecture, the cache size is reduced and the tag field only contains the least significant bit of the full tag. The reduction in tag field size affects the cache mapping to only a segment of the memory. This architecture takes advantage of a dynamic cache mapping scheme to map any memory location. The result is lower power consumption due to a smaller cache size but on average an instruction throughput degradation of up to 1.4% can be observed.

3. Context

A considerable number of low-power cache designs [22] are centered on the principle of adding an extra cache or buffer, usually small in size, and designing the system to fetch data directly from this buffer. This technique thereby prevents altogether a direct access to the original cache. Because the buffer is relatively small, it is possible to achieve significant power savings if one can ensure a high hit-rate to the buffer [22]. Another term that is used in the same context is 'filter cache' [23]. It has been observed [24] that instruction fetch and decode can consume up to 50% of processor power. Experimental results across a wide range of embedded applications showed that the filter cache of [23] results in improved memory system energy efficiency. An instruction filter cache or level-zero cache in [25] can be placed between the CPU core and the instruction cache to service the instruction stream. The filter cache can be efficiently optimized [23] by predicting the subsequent fetch addresses at run-time to identify whether the next fetch address is in the filter cache. In case a miss is predicted, the miss penalty is reduced by accessing the instruction cache directly. Consideration of these filter caches shows in hardware some of the features that we now delegate to software, though without using a cache-based memory hierarchy.

One disadvantage of such architectures and those of Section 2 is inaccurate prefetching due to misprediction of branch targets. By software intervention through compiler-controlled prefetch buffering, misprediction can be reduced. At the same time by placing more of the processing logic in software, energy-consuming prediction hardware is bypassed. Equally, address calculation is performed in advance in software, also reducing energy consumption.

The hardware architecture targeted embodies tri-state data busses and a dual-ported memory structure. In Figure 1, the code memory is divided into control words (shown in grey) and the current instruction itself (shown in black). Control words containing the pre-fetch addresses for the next instruction are inserted for multi-cycle instructions. As outlined in Section 1, control words are calculated by a compiler in advance and placed with a prior instruction to enable pre-fetching of instructions or data without the need for address calculation. In so doing pipeline stalls are reduced, as the number of cycles taken up by a multi-cycle instruction is reduced. In the special case of conditional branches, the need for prediction hardware is avoided by pre-calculating the instruction addresses of both the true and false branch targets. This allows two instructions to be prefetched into instruction buffers.

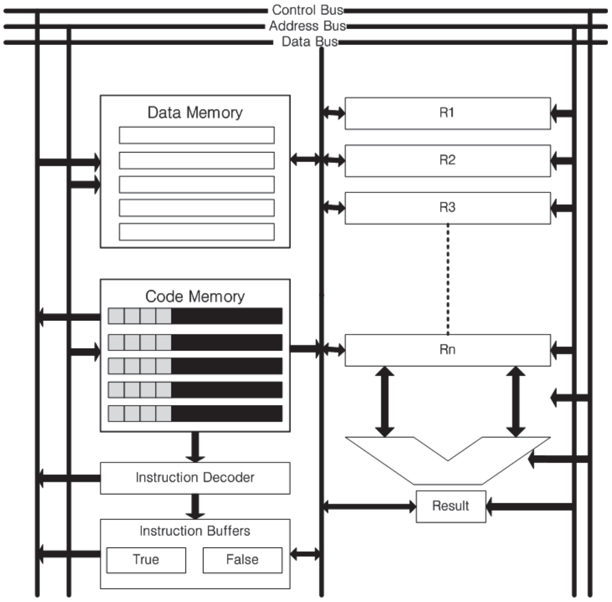


Fig. 1: Processor architecture with prefetch instruction buffering

One buffer is always available as a default location if a branch is taken (called True) and the other if the branch is not taken (called False). (Refer to Figure 1 in which these instruction buffers are shown after the instruction decoder.) Importantly, for normal operation, i.e. not a conditional branch or an instruction with associated target address, instructions are only placed in one buffer, the default True buffer. Because more than one instruction at a time will need to be prefetched when a conditional branch occurs, the code memory should be dual-ported, as shown in Figure 2.

4. Compiler Controlled Instruction Prefetch Buffering

Figure 3 illustrates the logical operation of the instruction buffer. The control-words control prefetching of instructions into the dual instruction buffers. Not shown in the diagram is instruction decoding that takes place before instructions are placed in the buffer. Alternatively, instructions could be placed in the prefetch buffer without decoding and after fetching be decoded as a third-stage prior to execution. Figure 4 is a timing diagram for operation of the two-stage instruction pipeline during a conditional branch. During the first clock cycle a first instruction fetch (IF) occurs. While this instruction is executed (EX), the second instruction is fetched and two possible target instructions are prefetched. Conditional on the outcome of the sec-

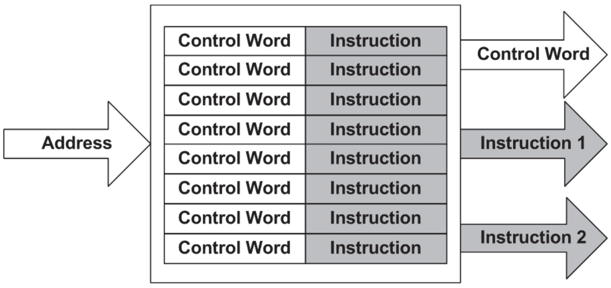


Fig. 2: Dual-ported memory organization to fetch Instruction(s) and Control words in one cycle

ond instruction (true or false) one of the two prefetched instructions is subsequently fetched.

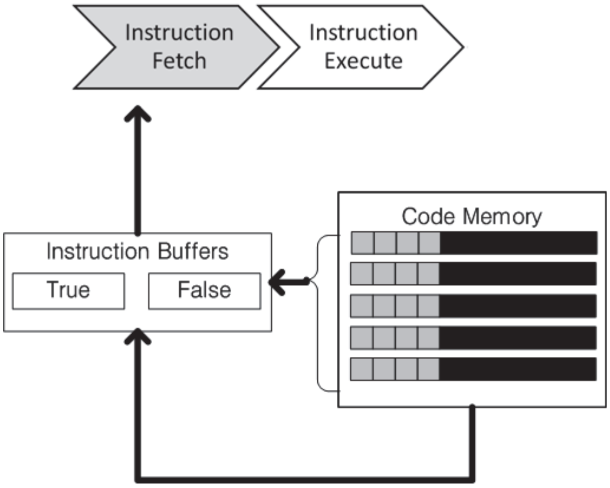


Fig. 3: Logical view of instruction buffer architecture

For most instructions except a conditional branch, the buffer operates in the same manner as that of a typical two-stage pipelined processor i.e. in the first cycle only one instruction fetch is performed, and in the following cycle the first instruction is executed and the next instruction is fetched (see Figure 5). As the single-cycle operation without a branch does not require any control word, it would be carried out uninterruptedly until a branch occurs. For multicycle instructions, the control words seek to reduce the number of cycles by placing pre-calculated addresses as code words with the instruction. If the subsequent instruction is a conditional branch, then two instructions are prefetched at a time for true and false

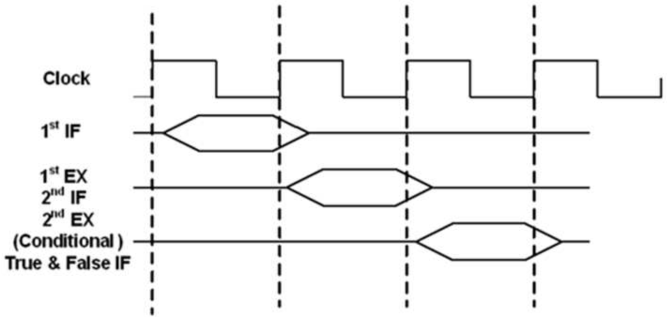


Fig. 4: Timing diagram of Instruction Buffer operation

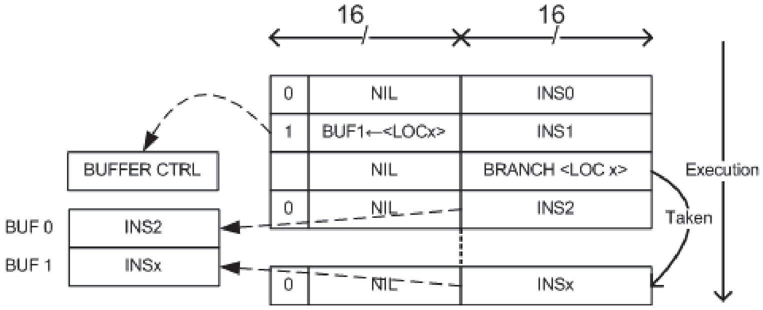


Fig. 5: Control Word fetch during buffer operation

operations. The address of these two instructions is indicated beforehand by the preceding control words. These two instructions will then be held at fixed locations in the dual instruction buffers i.e. there is one fixed location for true and another for false. If the branch is taken then the instruction from the True buffer location is executed; otherwise the instruction in the False buffer is executed. The non-branch instructions are stored and executed from the default buffer which can be either of the two. This type of buffer accelerates the instruction throughput of the processor which otherwise has to stall until the branch is resolved. Similarly, for unconditional branches or function calls/returns two-word long instructions that also indicate the target address are prefetched. The proposed buffer architecture allows prefetching both the words in a cycle and, thus, reducing the execution time by one cycle.

To summarize across the types of instructions catered for by the architecture. For simple single-cycle instructions, prefetch occurs as normal but null code words can be used to place a tri-state bus in a high-impedance state. For conditional branches, two alternative target instructions are prefetched and placed in a dual buffer structure. For multicycle instructions control words are used to specify pre-calculated addresses in order to reduce the number of cycles. Finally, for two-word instructions both words of the instruction are pre-fetched at the same time, thereby taking advantage of the dual buffer structure.

5. Architecture Implementation and Findings

5.1. *Implementation details*

5.1.1. *Software Implementation*

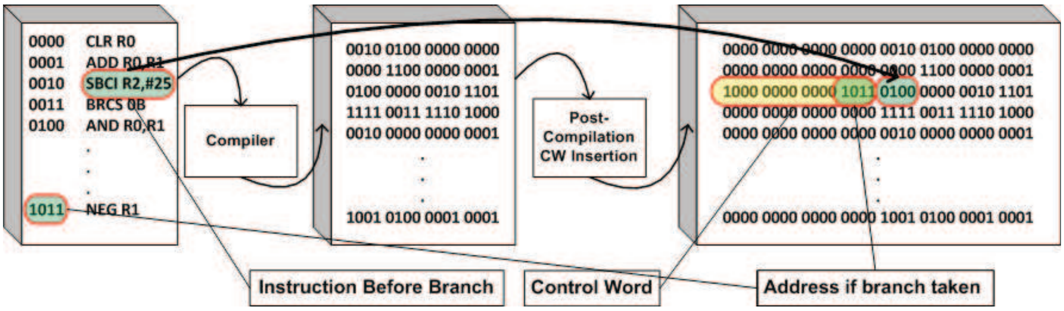


Fig. 6: Control Word insertion during compile time

The control words are inserted at compile time of the code. The software scans the compiled binary for target instructions i.e. branches, jumps, function calls etc. The software then inserts control words an instruction before the target instruction, so that at run-time, the prefetch unit gets the address beforehand when branch will be taken. In case the branch is not taken then prefetch buffers already knows the succeeding address is the one to buffer at second location. For example, in Figure 6, the software detects the address 0010b is the instruction before a branch, therefore a control word is inserted at this location. The control word contains the target address in case the branch is taken, that is 0Bh or 1011b. Similarly, all the compiled code is scanned and control words are inserted where required.

5.1.2. *Hardware Implementation*

The proposed architecture was implemented on an open-source, cycle accurate, VHDL implementation of an AVR core [26], hereby referred to as Original architecture and the one with the proposed instruction buffer architecture is referred to as Buffered architecture. Thus, both the Original and Buffered architectures are based on an ATMEL AVR core based on the VHDL Atmega103(L) model [27], which does not has any cache memory. The on-chip program code memory in both architectures is 64k×16bits in size, which again is the same as that of the Atmega103 model

[26]. In the Original architecture, instructions are transferred directly to an instruction register prior to instruction decoding in a two-stage pipeline. The Buffered architecture is similar but naturally includes a pre-fetch buffer.

The Atmega103(L) core, based on the ATMEL AVR, is an 8-bit enhanced RISC processor, which, because it executes most of its instructions in a single machine clock cycle, achieves instruction throughput approaching 1 MIPS per MHz. As shown in Fig. 7, there are 32 general-purpose registers, all of which are connected directly to the ALU. This arrangement allows two independent registers to be accessed in one machine cycle. A Harvard memory architecture is employed. The data memory size is 4k×8 bits, which is implemented as SRAM in the core. The Atmega AVR core also contains four level and four edge-triggered interrupts, together with a 16-bit timer and a UART.

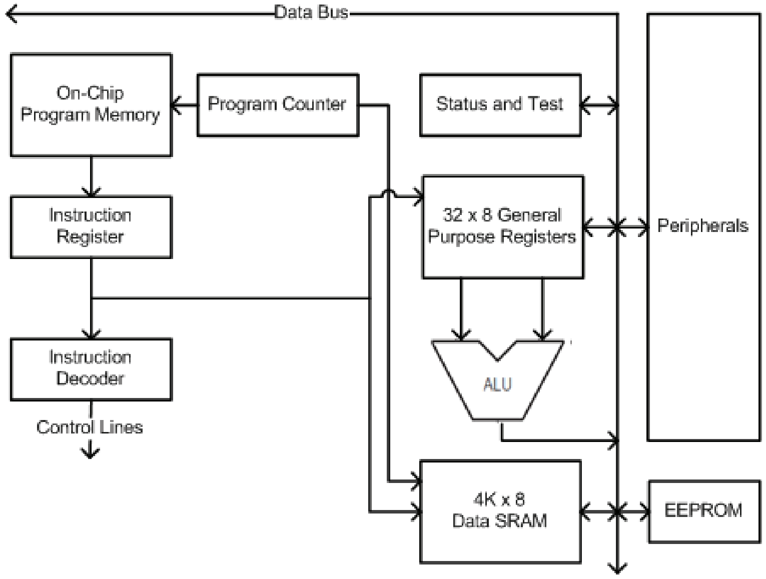


Fig. 7: Architecture of the Atmega103(L) core

The proposed architecture was implemented using Xilinx ISE 13.1 [28], verified by simulation using Mentor Graphics Modelsim 6.1 [29] and mapped to a Xilinx XC3S500E Field Programmable Gate Array (FPGA). The power consumption estimation was carried out using the Xilinx XPower tool [30]. The Xilinx Xpower tool is a spreadsheet-based tool that helps estimate the worst-case power consumption. In general, it can be used at any stage of the design cycle but in an FPGA's case it is normally employed at an early stage to determine whether power consumption will be within the bounds set for an application. In particular, the Xpower tool takes into account all the components of the design including on-chip Dual Ported

(DP)-RAM when used in the proposed design. The well-known HP lab's CACTI tool [31] was not used, as CACTI is specialized to cache power estimation, not to the buffer used in the proposal.

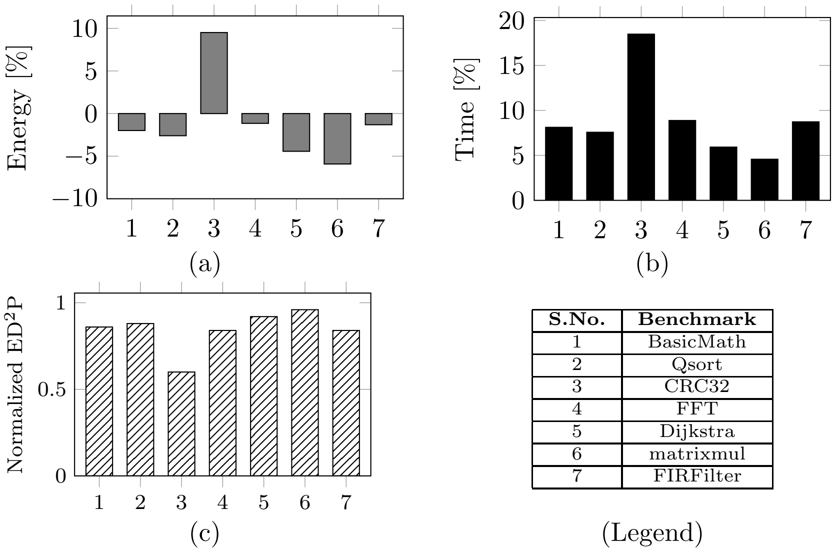
The proposed buffered architecture consumes 5721 LUTs and 866 Logic Slices compared to 2940 LUTs and 797 LogiSc Slices for the original un-buffered architecture. The proposed design was still able to fit into the target FPGA platform, with a maximum operating clock frequency of 33 MHz, which is the same clock frequency range as the Original architecture.

5.2. *Findings*

The per-cycle power consumption of the buffered architecture was found to be 111 mW compared to 100 mW of the original architecture. However the proposed architecture achieves greater energy efficiency by reducing the number of cycles required to execute an application. Notice that the results presented in this paper are based upon the FPGA implementation of both the Original and Buffered Architectures. Therefore, commercial viability of the proposed embodiment can only be verified using a chip design tool chain. The reader is reminded that we do not make a comparison with use of an instruction cache, as the low-end embedded processors considered in this paper do not use caches. The introduction of the control words into the user application can be performed during compilation, by the compiler or can be done after compilation. The experimental setup used post-compilation software to identify and place control words with the appropriate instruction.

The proposed architecture was tested using seven applications from the MiBench [32] benchmark suite i.e. (1) Basic Math, (2) Quick Sort, (3) CRC-32, (4) FFT, (5) Dijkstra, (6) Matrix Multiplication, and (7) FIR Filter. All the benchmark applications except CRC-32 showed an overall energy overhead of up to 6% and in case of CRC-32 energy savings of around 10% are observed(see Figure 8(a)). The general trend of execution time savings (see Figure 8(b)) is for a 6–10% saving for most of the applications except for the CRC-32 application that experienced around 18% saving in execution time. The Matrix Multiplication application was slightly below the execution time saving trend, with a 5% saving.

It was due to the nature of benchmarks that different amounts of execution time savings occurred. Figure 9 shows profiles of all the benchmark applications used to evaluate the proposed architecture. It can be observed that almost all benchmarks have around 10% of instructions that the proposed buffered architecture addresses i.e. Jump, Branch, Program Memory Load, and Call. The CRC-32 application is exceptional, as it has around 30% of its instructions buffered as a result of control words inserted at compile time. Since the proposed architecture is most suitable for workloads where multicycle instructions are greater, therefore CRC-32 shows an overall energy savings of up to 10%. This observation is further supported by the fact that FFT application has a mere 1.15% energy overhead (see Figure 8(a)), and has around 20% multicycle instructions (see Figure 9(d)).



(a)

(b)

(c)

S.No.	Benchmark
1	BasicMath
2	Qsort
3	CRC32
4	FFT
5	Dijkstra
6	matrixmul
7	FIRFilter

(Legend)

Fig. 8: Buffered Architecture (a) Energy, (b) Time Savings, and (c) Normalized ED^{2}P

A more comprehensive metric to observe the impact of execution time and energy consumption simultaneously is the Energy Delay Square Product (ED^{2}P) [7] [8], shown for each benchmark in Figure 8(c). The average ED^{2}P for the buffered architecture when normalized against the original architecture was found to be around 90% and the least amount of ED^{2}P reduction was observed for Matrix Multiplication i.e. around 4% of the original one that is due to the least amount of multicycle instructions i.e. around 12% (see Figure 9(f)). For the CRC-32 benchmark 40% ED^{2}P reduction is observed, as compared to the original non-buffered architecture. It may be noted that although the per-cycle power consumption of the buffered architecture is around 11% more than the original one; a significant amount of savings when expressed as energy and time combined through the ED^{2}P metric could be observed for applications where multi-cycle instructions account for more than 20% of the total workload.

6. Conclusion

In this paper a software-controlled instruction buffering architecture for low-end processors was introduced. The proposed architecture was implemented using VHDL over an available open-source microcontroller core. The architecture in general showed 6–10% improvement in execution times for a good number of benchmarks, with matching ED^{2}P reduction of 4–40%. The latter maximum occurred for CRC-32, which made frequent use of the prefetch buffer. For widely deployed low-end processors such as the AVR there are potentially many embedded applications

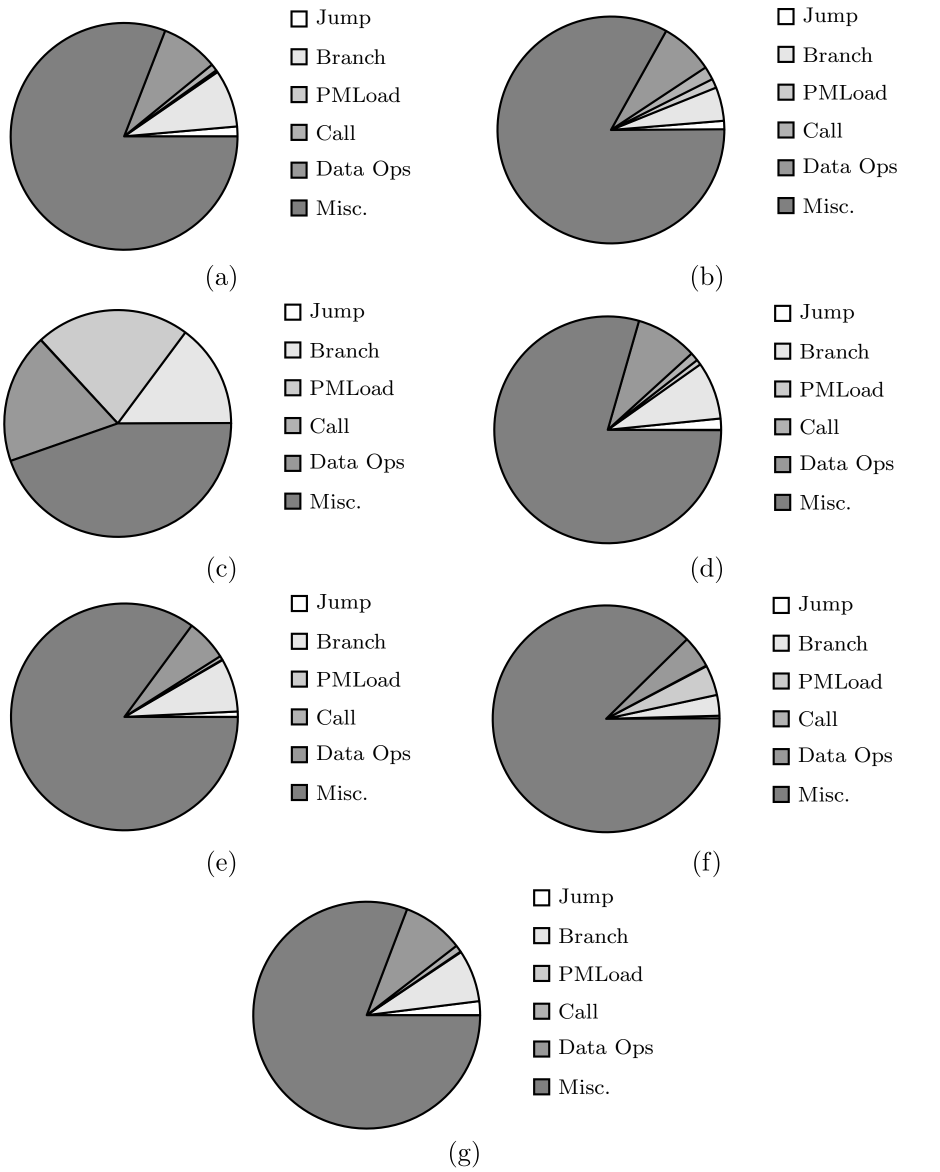


Fig. 9: Benchmark Profiles for (a) Basic Math, (b) Qsort, (c) CRC32, (d) FFT, (e) Dijkstra, (f) matrixmul, and (g) FIR Filter

that will gain from this reduction in execution time, as we have sought to show by benchmarking algorithmic staples such as the FFT and CRC. The design exploits the single-cycle latency of typical embedded processors to achieve these startling results. The proposed architecture is applicable when low-power operation is desirable. It is particularly appropriate as an alternative way to achieve energy-efficient operation for battery-powered, embedded processors where multi-cycle instructions account for more than 20% of the total workload.

References

1. G. Moore, Excerpts from a conversation with Gordon Moore: Moore's law,*Video Transcript, Intel* (2005).
2. J. M. Rabaey and M. Pedram, *Low power design methodologies* (Kluwer Academic Publishers, 1996).
3. C. Zhang, F. Vahid and W. Najjar, A highly configurable cache architecture for embedded systems, *Proc. of 30th Ann. Int. Symposium on Computer Architecture*, 2003, pp. 136–146.
4. A. Malik, B. Moyer and D. Cermak, A low power unified cache architecture providing power and performance flexibility, *Proc. of the Int. Symposium on Low Power Electronics and Design*, (2000), pp. 241–243.
5. T. C. Mowry, M. S. Lam and A. Gupta, Design and evaluation of a compiler algorithm for prefetching, *ACM SIGPLAN Notices* **27** (1992) 62–73.
6. J. Turley, Atmel AVR brings RISC to 8-bit world,*Microprocessor Report* **11** (1997).
7. P. Bose, M. Martonosi and D. Brooks, Modeling and analyzing CPU power and performance: Metrics, methods, and abstractions, *Tutorial, ACM SIGMETRICS* (2001).
8. Y. Liu and H. Zhu, A survey of the research on power management techniques for high-performance systems, *Software: Practice and Experience* **40** (2010) 943–964.
9. M.Y. Qadri, N.N. Qadri, K.D. McDonald-Maier, *Software controlled instruction prefetch buffering*, United States Patent and Trademark Office Patent Application Number: 13/918,431, filed (June, 2013).
10. D. Cuartielles, Arduino FAQ from: Malmö University (2013).
11. A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu et al., Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture, *IBM Systems Journal* **45**, (2006) 59–84.
12. T. Chen, T. Zhang, Z. Sura and M. G. Tallada, Prefetching irregular references for software cache on cell, *Proc. of the 6th Ann. IEEE/ACM Int. Symposium on Code Generation and Optimization*, 2008, pp. 155–164.
13. Y. Solihin, J. Lee and J. Torrellas, Using a user-level memory thread for correlation prefetching, *Proc. of the 29th Ann. Int. Symposium on Computer Architecture*, 2002, pp. 171–182.
14. S. Byna, Y. Chen and X.-H. Sun, Taxonomy of data prefetching for multicore processors, *Journal of Computer Science and Technology* **24** (2009) 405–417.
15. M. Annavaram, J. M. Patel and E. S. Davidson, Data prefetching by dependence graph precomputation *ACM SIGARCH Computer Architecture News* **29** (2001) 52–61.
16. C.-K. Luk, Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors *ACM SIGARCH Computer Architecture News* **29** (2001) 40–51.
17. C. Zilles and G. Sohi, Execution-based prediction using speculative slices *ACM SIGARCH Computer Architecture News* **29** (2001) 2–13.
18. C. B. Zilles and G. S. Sohi, Understanding the backward slices of performance degrading instructions *ACM SIGARCH Computer Architecture News* **28** (2000) 172–181.
19. D. Parikh, K. Skadron, Y. Zhang, M. Barcella and M. R. Stan, Power issues related to branch prediction, *Proc. of Eighth Int. Symposium on High-Performance Computer Architecture*, 2000, pp. 233–244.

20. R. Kahn and S. Weiss, Thrifty BTB: A comprehensive solution for dynamic power reduction in branch target buffers, *Microprocessors and Microsystems* **32** (2008) 425–436.
21. J. Gu, H. Guo and P. Li, An on-chip instruction cache design with one-bit tag for low-power embedded systems *Microprocessors and Microsystems* **35** (2011) 382–391.
22. J. Henkel and S. Parameswaran, *Designing embedded processors: a low power perspective* (Springer Verlag, 2007).
23. J. Kin, M. Gupta and W. H. Mangione-Smith, The filter cache: an energy efficient memory structure, *Proc. of the 30th Ann. ACM/IEEE Int. Symposium on Microarchitecture*, 1997, pp. 184–193.
24. W. Tang, R. Gupta and A. Nicolau, Power savings in embedded processors through decode filter cache, *Proc. of Design, Automation and Test in Europe Conf. and Exhib.*, 2002, pp. 443–448.
25. N. Bellas, I. Hajj and C. Polychronopoulos, Using dynamic cache management techniques to reduce energy in a high-performance processor, *Proc. of the Int. Symposium on Low Power Electronics and Design*, 1999, pp. 64–69.
26. R. Lepetenok, The VHDL implementation of AVR Open Core, (2014) *URL: http://opencores.org/*.
27. T. van Leuken, A. de Graaf and H. Lincklaen Arriens, A high-level design and implementation platform for IP prototyping on FPGA, *Proc. of the ProRISC IEEE 15th Ann. Workshop on Circ., Systems and Signal Processing*, 2004, pp. 68–71.
28. Xilinx ISE Design Suite ver 13.1 by: Xilinx Inc. *URL: http://www.xilinx.com*
29. Modelsim ver 6.1 by: Mentor Graphics Inc. *URL: www.mentor.com/products/fv/modelsim/*
30. Xilinx XPower Estimator tool by: Xilinx Inc. *URL: http://www.xilinx.com*
31. G. Reinman and N. Jouppi, *CACTI 2.0: An integrated cache timing and power model*, tech. rep., Compaq Computer Corporation Western Research Laboratory (2000).
32. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge and R. B. Brown, MiBench: A free, commercially representative embedded benchmark suite, *Proc. of the IEEE Int. Workshop on Workload Characterization,*, 2001, pp. 3–14.