ELSEVIER

Available online at www.sciencedirect.com

ScienceDirect

Microelectronics Journal 00 (2016) 000–000

Micro-electronics Journal

www.elsevier.com/journals/microelectronics

Energy-efficient data prefetch buffering for low-end embedded processors

Muhammad Yasir Qadri[1] Nadia N. Qadri[2] Martin Fleury[1]a* Klaus D. McDonald-Maier[1]

[1]*School of Computer Sci. and Electronic Syst., University of Essex, Colchester Co4 3SQ, U.K.*

[2]*Dept. of Electrical Engineering, COMSATS Institute of Information, Technology, Wah Campus, Pakistan*

Abstract

An energy-efficient architecture should jointly optimize energy consumption and throughput, as captured by the Energy-Delay-Square Product (ED^2P) metric. This paper introduces a prefetch data buffer micro-architecture, which achieves that goal with the aid of software-inserted control words to govern the prefetch process. The proposed architecture is aimed at low-end embedded processors, which, so as to reduce energy consumption, lack a cache-based memory hierarchy. By identifying after compilation which data should be prefetched and modifying the object code, the rate of prefetch misses is reduced. And by pre-computing memory addresses using auxiliary software after compilation and modifying the object code, address computation by hardware at run time is avoided, reducing pipeline stalls and, thus, improving throughput. Additionally in the case of branches, by prefetching two data items at any one time, alternative instruction outcomes are anticipated. The paper contains results from running a range of well-known and representative benchmarks on the proposed architecture. There was an improvement of 6%−20% compared to an unbuffered architecture in execution times when tested over those seven benchmarks. Furthermore, the average ED^2P for the buffered architecture when normalized against the same architecture without buffering was found to vary between 54% − 90% according to benchmarking, though there is a cost in code size increase. That is to say, for the benchmarks tested there was a net energy efficiency improvement of between 10% and 46% in comparison with the equivalent unbuffered architecture with a lower area overhead.

* Corresponding author. Tel.: +44 1206 872678; fax: +44 1206 872900.

Keywords: control words; data prefetch; embedded processor; micro-architecture

1. Introduction

The growing proliferation of embedded battery-powered devices, often performing complicated tasks, leads to the prioritization of energy-efficient design optimizations. Such design optimizations strive to strike a balance between energy usage and throughput in a system and do not simply attempt to reduce energy consumption. In some energy-efficient designs, for some applications, there might even be a negative impact on energy consumption. However, the throughput can to rise to compensate, as quantified in the Energy Delay Product (EDP) metric [1].

The proposed architecture gives preference to a data prefetch buffer rather than a data cache. Though cache-based memory hierarchies are the norm for general-purpose PC architectures, in the embedded world system, architectures may employ alternatives to caches. The energy usage and chip area take-up of caches these can both be considerable. In addition, if the cache is not carefully tuned the cache miss ratio will also increase, leading to processor idling. Thus it is [2] that caches are a problematic feature in battery-powered embedded systems. Prior research [3] [4] [5] confirms that caches may be responsible for as much as 50% of a low-end processor's energy budget. For the most part, not herein, a cache-like structure underpins software-prefetching schemes, i.e. software prefetching assists already present hardware cache memories. However, the proposed scheme does *not* require a cache-like structure to be present, which is why it is likely to be more effective. Instead, the proposed architecture with a data prefetch buffer replaces the typical cache memory, and, hence, the inherent disadvantages of such caches (i.e. compulsory, conflict, and capacity misses). Comparing to the typical cache, the proposed prefetch buffer requires: much smaller storage, is more area efficient, and less power consumption.

Compiler-controlled prefetching of data [6] is one way that a data prefetch buffer can take the place of a data cache in an energy-efficient manner. However, prefetching typically suffers from

E-mail address: fleum@essex.ac.uk.

an increased memory bandwidth. This increase is caused by unnecessary prefetches, owing to false predictions by the particular algorithm employed. (For previous research on prefetch buffering refer to Section 2.) This paper proposes a software-controlled prefetch buffering architecture that, through the mechanism of control-word insertion, removes such false predictions. The proposal also has a number of additional advantages, including a reduction in pipeline stalls arising from memory address calculations. We believe that the introduction of a varying instruction size, when control words are introduced is justified by the gains made. The Acorn RISC Machine (ARM) in its Thumb variant also includes 16-bit and 32-bit instructions.

Low-end embedded microprocessors and microcontrollers typically have on-chip memory. This is a way to reduce the number of additional components needed in an embedded application. Such an arrangement also results in single-cycle access to the memory, which this paper's proposal takes advantage of. In a further simplification, the instruction pipeline changes from the basic five-stages of a Reduced Instruction Set Processor (RISC) to just two stages. Both the 8-bit Atmel AVR [7] and the Peripheral Interface Controller (PIC) microcontroller families [8] support on-chip memory and two-stage pipelines, which the proposed software-based prefetching technique exploits. Both families also have a Harvard architecture, which allows instructions and data to be accessed simultaneously. As these microcontroller families are extensively deployed, the proposal in this paper is of wide generality and applicability to embedded applications. For example, by 2013 an AVR was present on every one of 700,000 official Arduino boards[b] and Microchip, PIC's manufacturer, output a 2013 press release[c] stating that it supplies one billion processors per year. According to an SAE article of 2014 entitled "Market for 8-bit chips remains strong", T. Costlow points out that such processors account for 24% of the automotive microcontroller market, which figure is expected only to decline to 22% by 2018. As the vast majority of applications for low-end processors are in embedded computing, not in general-purpose computing, our proposal is geared towards embedded computing applications. Though, as Section 2 describes, pioneering work has gone on in the past within the general field of pre-fetching, we believe there is still scope for improvements, even though these improvements will now be focussed on specific domains.

---

[b] According to Cuartielles in 2013 on the Arduino FAQ at http:// http://www.arduino.cc/en/Main/FAQ

Immediately after the compilation of application code, our software inserts control words, which cause a processor to prefetch data required by the next instruction in the two-stage instruction pipeline. In this way, the method avoids static pointer-based data references and associated address computations. More generally, the prefetch technique is implementable either by an additional software tool operating at compile time, the choice herein, or by an enhanced compiler directly, not used by us. Specifically, during the software creation phase or after program compilation, control words are placed at a location at least one instruction ahead. As a result, during execution the data required can be fetched without pipeline stalls. Therefore, this architecture provides greater energy efficiency when compared to an unbuffered architecture with lower area overhead. As with other software prefetching schemes, our proposal leads to what could be for some applications a significant increase in code size owing to the need to store some 32-bit rather than 16-bit instructions to accommodate control words. The significance will depend on the size of the application code, which might anyway fit within the existing on-chip memory.

Although the previously-mentioned EDP [1] is a widely adapted metric to evaluate energy and delay effects, Martin et al. [9] recommends a weighted approach, using another metric i.e. energy-delay-square-product (ED^2P), which in [9] is alternatively called energy-time-square (ET^2), as T is delay. This metric is very useful in evaluating trade-offs between the circuit-level power consumption and the overall energy efficiency of the system [10]. The ET^2 (or alternatively ED^2P) metric was first introduced by Martin et al. in [9] in order to evaluate the asynchronous MIPS3000 processor. The validity of the metric was later analysed by Martin in [11] [12]. In [13], ED^2P is defined as $ED^2P = EPI \cdot CPI^2 = EPC \cdot CPI^3$, where EPI is the energy per committed instruction, EPC is energy per cycle, and CPI is cycles per instruction. For a complete execution of benchmark i, ED^2P can be calculated as: $ED^2P_{\eta i}=\eta_i.EPC.CPI^3$, where η_i is the number of instructions executed. However, herein we use the notation ED^2P instead of $ED^2P_{\eta i}$ as a simplification.

In evaluating a design [13], ED^2P highlights performance more than EDP does. To first order, ED^2P is also independent of variations in voltage and frequency. A mathematical analysis of the advantages of using ET^2 over ET can be found in [11], where it is said that “The energy-delay

[c] Available at http://www.microchip.com/pagehandler/en-us/press-release/microchips-12-billionth-pic-mi.html

product E × t is often used to compare designs but is unfortunately not an acceptable metric". Indeed, some authors, for example [14], even suggest using the cube of delay to weight the delay more than the energy of a system. However, we adopt a more moderate approach. The focus of the architecture is to achieve greater throughput for an overall reduced active cycle for a processor. Thus, the authors of this paper consider ED^2P to be the most appropriate metric for the research presented in this paper. Motivated and guided in that way, in this paper we introduce a novel prefetch data buffering micro-architecture for low-end, embedded processors with on-chip memories, which provides increased energy efficiency.

Finally in this introduction, notice that a brief outline of some of our ideas has been filed as a U.S. patent application [15], though without relevant prior research papers, consideration of the context, or performance results and analysis, as now occurs in this paper. This paper also includes a longer description of the innovation and broadens the treatment. Our ideas are also applicable to instruction prefetching. In [16] we did exactly that, thus confirming the benefit of the ideas contained in this paper for data prefetching.

The rest of this paper is arranged as follows. Section 2 is a review of related work in this field before going on to describe the proposed architecture in Section 3. In Section 4, the area and power overheads of the software-controlled prefetching are compared against those of the original unbuffered architecture. A detailed analysis of energy consumption reduction and throughput improvement occurs using various benchmark applications. Finally, Section 5 rounds up the paper with some concluding remarks.

2. Related research

A significant trend in low-power cache design [17] is to include an additional small extra data buffer, which is accessed directly by the embedded system. Therefore, these designs require the buffer to be accessed first, preventing altogether direct access to the original caching structure. The intention of such designs is to save energy by achieving a high hit rate to the small intermediate buffer. Notice that a larger buffer does not result in the same energy savings. Investigation of these intermediate hardware buffers or caches is the inspiration behind substituting software control of data prefetch. Crucially, however, the current paper avoids a cache-based memory hierarchy. On the other hand, purely hardware-guided prefetching of data

into caches, e.g. [18], may be energy intensive [19] and is certainly not suitable for multicore platforms. (For some counter-examples of energy efficient hardware prefetching consult [20].)

Most recently, there has been interest in introducing machine learning into prefetching. Work in [21] considers the risk of aggressive prefetching saturating the memory bandwidth of a multicore processor, for example, with a 40% risk of hardware prefetching harming the performance of Intel's Sandyridge i7-2600K processor. Instead [21] considers dynamically combining hardware- and software-based prefetching in the Adaptive Resource Efficient Prefetching (AREP) framework. That framework examines a selection of prefetch configurations in order to choose the one with the least impact on performance. The work in [21] reports an 8% increase on average in performance from applying AREP. The automatic prefetching tuner (PATer) for the POWER8 processor [22] provides a way of tuning the prefetch configuration. The need arises because the POWER8 processor has a 25-bit hardware register in which the cache prefetch configuration can be set. Without the aid of linear discriminant analysis, manual tuning of the register faces a difficult task owing to the large number of possible configurations. However, to apply PATer requires an offline training phase with representative workloads. The authors of [22] report a 1.4 improvement in processing speed but do not consider energy consumption. Again, tuning is for a cache-based system, not the cache-less processors considered in this paper. Moving on, in [23], machine learning, specifically Phase-Residency and FFT fingerprints, is used to identify phases within a processor's workload for which it is preferable to either employ compiler-based prefetching or hardware-based prefetching. The work targeted level-2 cache prefetching for the Xeon Phi many-core processor and a 95% prediction precision was reported. Clearly though with between 57 and 61 cores, the Xeon Phi is a much more complex processor than the ones considered herein.

There are a number of historical patents that are partially relevant to the software-controlled data prefetch buffering mechanism proposed in this paper. Software supervision of cache coherence is proposed in [24], especially in respect to a multiprocessor of the type in which write-through to main memory is not used. It is suggested that the addition of cache-control instructions to a processor's instruction set can ensure main memory integrity in a situation when each processor has its own private cache taken from a shared memory. Turning to the innovation described in [25], this innovation is directly concerned with prefetch buffering, though for

instruction rather than data prefetch buffering. Two hardware pointers are maintained to indicate where slots in the prefetch buffer can be updated and where instructions can be taken from to be executed. This hardware control of the prefetch buffer is thought to be helpful in the case of short loops, in which the same instructions from the prefetch buffer might be executed. The prefetch buffer control can also be helpful when the buffer itself is a circular buffer because it can control overwriting of still-to-be-executed instructions in the buffer. Branches (approximately 15% − 25% of all instructions [26]) are another example of when instructions from a prefetch buffer might be re-executed.

The purpose of the innovation just described is complementary to our proposal, as it involves control of instructions about to be placed or already in a prefetch buffer, rather than the instruction or (in our case) data access process itself. Patent [27] does involve a prefetch data buffer and the addition of prefetch address decoding instructions (stored beforehand in the instruction cache). Thus, a Harvard architecture is assumed such that the purpose of the prefetch address decoding instructions is to bring data into the prefetch buffer before they are needed in the instruction cache. Therefore, the motivation of the patent is different to that of this current paper. Again in the context of caches, compiler control of instruction/data cache prefetching is proposed in [28]. Specialist prefetch instructions are included in the processor instruction set for use by the compiler. The main purpose of the innovation is to reduce thrashing occasioned by re-storage of the same information in a cache because of a mapping clash causing a cache line to be repeatedly overwritten while executing a program loop.

Off-line analysis of previous application execution traces is another technique that is utilized in [29] to improve prefetch prediction performance. For static embedded applications, such somewhat cumbersome techniques may be worthwhile. However, where dynamic adjustment is required then helper threads [30] may run ahead of execution to guide prefetching. Alternatively, idle hardware may be utilized [31] for a similar purpose. However, this is only appropriate if there is available processing power, as may occur on multicores. Clearly also, these latter techniques cannot also save energy.

Further examples of prefetch buffering receive detailed analysis in the authors' previous work. These examples are mostly intended to reduce the cache miss rate, reducing pipeline stalls, rather than reducing energy usage. One threat that such designs [32] face is irregular memory

references, which result in high miss rates. Either compiler analysis [33] or pre-execution by a thread [34] [35] [36] are ways that have been tried in the past in order to counter irregular memory references. However, the survey in [37] of prior data prefetching techniques for multicore processors indicates that random memory access remains a challenge to be addressed.

One principal weakness of prefetch proposals in the literature is that branch targets can be inaccurately predicted. Software control of branch target prefetching has the potential to significantly reduce this category of prediction error. At the same time, the energy expended by prefetch prediction hardware is saved. Software intervention also avoids the need for hardware computation of addresses, as they are calculated in advance. For these reasons, we were guided by the literature towards software intervention in prefetch buffering, as now discussed.

3. Software-controlled data prefetch buffering

This Section describes the prefetch buffering mechanism in the context of embedded processors.

3.1. Context

As mentioned in Section 1, energy efficiency is not simply the reduction of energy consumption but requires the joint optimization of energy consumption and throughput. A processor is usually in sleep mode [38], awakening for brief periods in time when required, resulting in a low duty cycle. The period of activity can be reduced by increasing the throughput during that time. The peak energy consumption and/or the sleep mode energy consumption can also be reduced. Thus, energy and throughput are inter-related and are jointly involved in overall power saving. Unfortunately for a cached architecture, whenever a processor in an active state encounters a cache miss, it must stall until the required block of data is fetched from lower down in the memory hierarchy and is made available to it. The amount of time a CPU spends in such a state, the cache miss penalty, lowers the throughput in cache-based memory organizations. The proposed design provides efficient prefetching by avoiding the large miss-prediction penalties that can arise in typical cache-based memory hierarchies.

In fact, the proposed buffer storage is different from a typical cache memory in many ways. A cache fetches data from memory on a block/line basis, which in turn can result in fetching some data that is not required. This causes memory transactions that are not desired and results in an

energy overhead. Also the undesired data occupies the cache storage and needs to be flushed to store other blocks of data as required. Therefore, the pipeline has to stall until the data are made available in the cache. All these steps lead to energy consumption, which the intention of this paper is to jointly reduce.

However, despite not using data cache memory, prefetch buffering can be compared (hypothetically) to a cache configuration in which: the prefetch locations are fully associative; and the replacement policy is simple, as with each new instruction the data in the buffers get replaced if required. The write policy is write-through, since the memory latency is usually one in low-end, embedded processors.

Tri-state data busses are assumed in the design; both the Atmel AVR and the PIC have these. In addition, the presence of a dual-ported memory structure is assumed. In fact, it is common to implement register files as dual- or multi-ported Static RAM and in [39] there is an example of an AVR communicating with a Field Programmable Gate Array (FPGA) via dual-ported RAM. The control words are inserted when required. To reduce energy consumption further, null control words are available to put the data bus into the high-impedance state of the tristate bus. Although, on the one hand, the insertion of control words in this software-assisted prefetch buffering increases the code memory size, on the other hand, throughput increases and prefetch accuracy is improved. The addition of a control word requires an extra 16-bits of data, along with the usual 16-bit program memory width. By increasing the bus width to accommodate control words, there is no impact on memory bandwidth.

To operate this scheme almost certainly requires a software tool or an augmented compiler to automatically generate control words and insert them into a program's code. Additionally software functionality is needed to keep track of available data buffer space. There now follows a discussion in which the data buffering architecture is further explained.

3.2. *Data prefetch buffering*

The proposed system comprises of a storage array (the data buffers). The storage array has several storage locations each with an associated tag (refer to Figure 1, where the additional storage locations are labelled A, B, and C). The software inserts control words into a program's code. These control words are subsequently employed whenever needed to direct prefetching of

data required by the instructions that follow. At a minimum, the control words are inserted one instruction ahead of the targeted instruction in the case of a two-stage pipeline but may need to be placed several instructions before the targeted instruction in the case of pipelines with more than two stages. Typically, the next instructions will be load instructions but store instructions can also benefit, as these also require address computations. Address computation usually requires multiple machine cycles and, thus, can cause pipeline stalls.

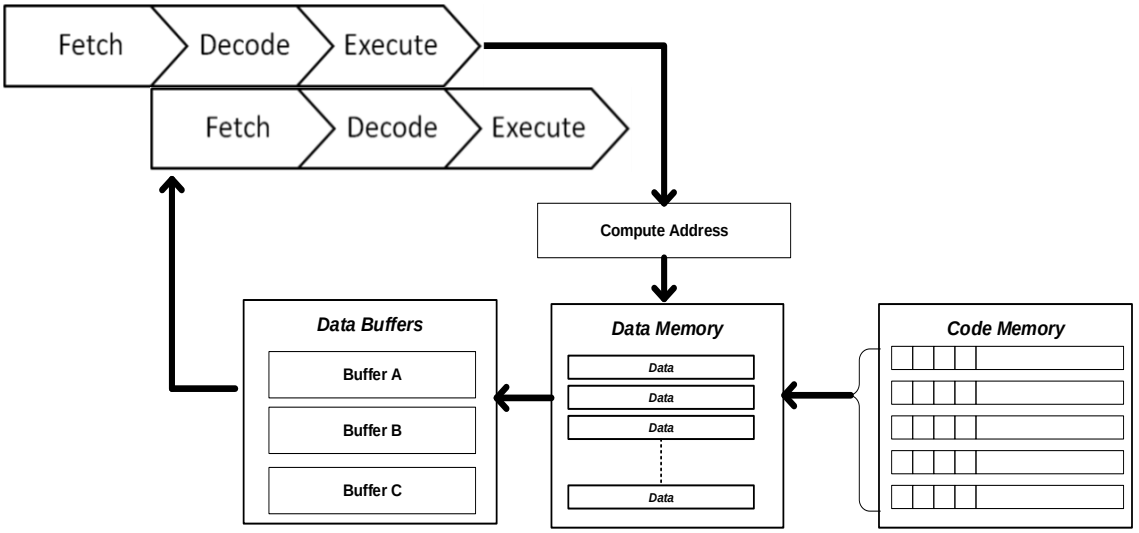

Fig. 1. Data buffering architecture

The authors have developed simple Visual Basic software to add these control words. In a first pass, once the application code is compiled and linked, the resulting .hex file is read by this software and all the instructions that are load/store/branch instructions are identified. This is achieved by matching the opcodes against the instruction-set table. Subsequently in a second pass, control words are inserted an instruction before the target instruction. As the control words will be fetched ahead of the target instruction and as they point to the location of the data, the pre-fetch process starts and data are made available in buffers A, B, and C beforehand. An AVR implementation of control word insertion is described in more detail in Section 4.1.

As mentioned in Section 1, the method is, therefore, particularly helpful in avoiding stalls whenever there are pointer-based operations including pre-increment, post-decrement, indirect load or store with fixed displacement, or combinations of these operations. In the case of

dynamic pointer-based operations, in which address calculation is performed dynamically during execution such as in pointer-chasing, prefetch buffering can supply the data needed by the next instruction to start execution in the pipeline. This is achieved by operand forwarding from the instruction already executing, the result of which instruction will determine the address for the next data to be fetched. The operand forwarding scheme that is commonly used to avoid pipeline stalls results in a reduction of an additional cycle that is otherwise required for address computation.

The data buffering space contains several data buffer locations and has address tags associated with each storage location. Taking advantage of dual-port memory of the data buffering, a pair of data can be fetched in every machine cycle based on the preceding control words. If a branch instruction occurs then the fact that two data items can be fetched simultaneously is very beneficial, as one data item is available for a true result and the other for a false result. The data buffering has to contain address tags of the data to be fetched. However, the buffering size can be much smaller than the typical cache because the data fetching is highly targeted and no other data other than that required are fetched. The data replacement is determined in advance under software control as per the requirements of the program executing and the data buffering space available.

As also mentioned in Section 1, we take the Atmel AVR as an example of the targeted low-end, embedded processors. Because the Atmel AVR is a RISC-based Load/Store architecture, all the ALU operations are carried out on the registers, i.e. data from the memory are fetched to registers and stored back from them to the main memory. In this architecture, the direct load and store instructions, (LDS and STS in the AVR instruction set), take one cycle to execute on the reduced core version. However, indexed or pointer-based load/store operations take two cycles to execute (on the reduced core version). The proposed data buffering architecture resolves the data dependencies of the subsequent instruction a cycle ahead and stores the required data in the data buffering beforehand. The AVR architecture has three index registers namely X, Y, and Z; hence, three data buffers, labelled A, B, C, are now introduced to hold the respective data associated with them. The load and store instructions for the pointer-based operations also support pre-increment, post-decrement, and indirect load and store with fixed displacement, as previously described.

Thus, the prefetched data are stored in buffers A, B, and C. In the VHSIC Hardware Description Language (VHDL) source code, the instruction fetch unit is modified. We have added registers to store control words and logic to fetch data from addresses pointed to in those registers. Figure 2 is a timing diagram for illustrative code before the modifications, whereas Figure 3 is a timing diagram for after the modifications. Notice that in Figure 2, the unmodified architecture takes two cycles to perform a load operation, whereas in the proposed buffered architecture of Figure 3, such an instruction executes in one cycle only.

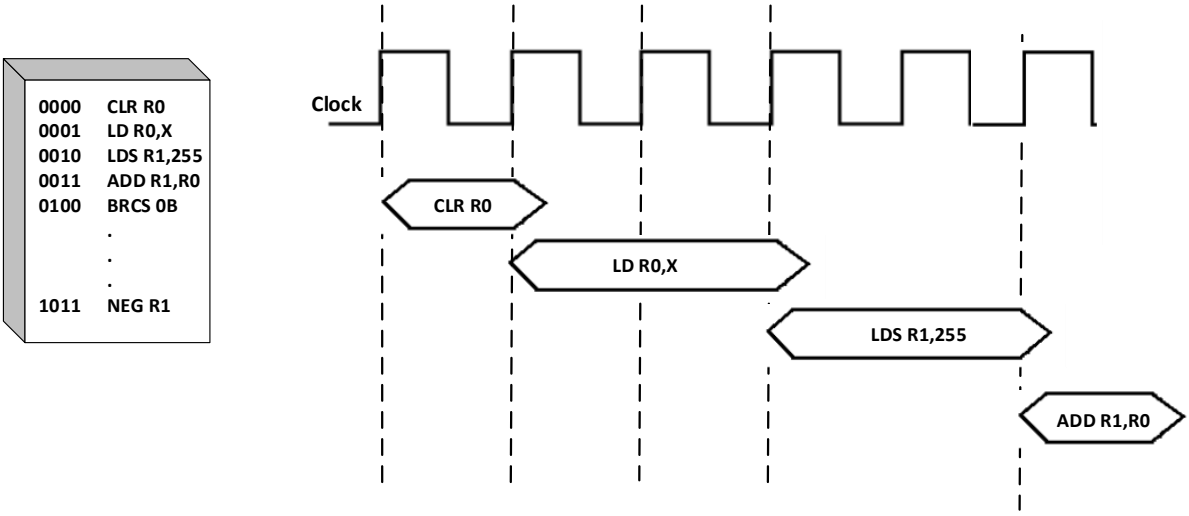

Fig. 2. Instruction timing diagram in the original architecture

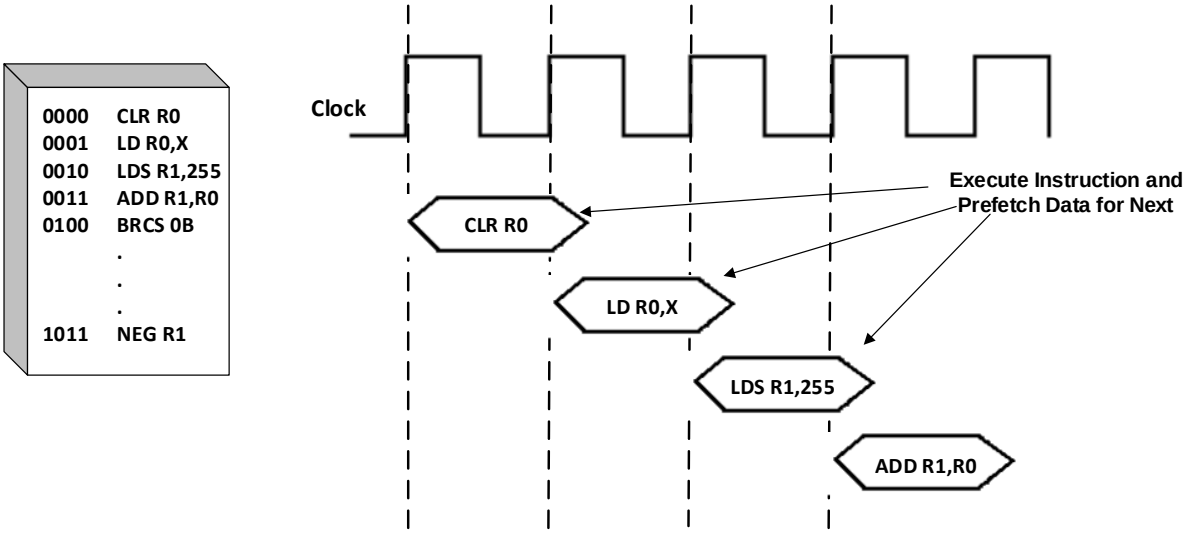

Fig. 3. Instruction timing diagram in the modified architecture

All of these operations use an address computation module to resolve the target address. The control words tell the address computation module about the forthcoming operation in the pipeline so that, upon arrival, the respective instruction can find the required data in the related data buffers and, thus, take just one cycle to execute. In Figure 2, in the case of a load instruction, the pipeline has to stall for two reasons 1) address computation and 2) the data fetch operation. In the proposed architecture, address computation at runtime is avoided as the data required in the next instruction are prefetched and made available in the buffers beforehand. Hence, during the execution of the load instruction, the proposal avoids address computation and allows data to be directly accessed from the local buffers without stalling the pipeline.

As an alternative to control word insertion, compiler instruction reordering cannot achieve equivalent performance because the time a load/store instruction requires will not be changed in that case. The pipeline has to stall for one more cycle until the data fetch is complete, see Figure 2. In the proposed architecture, the data are fetched and stored in the buffers before the instruction commit.

4. Architecture Implementation, and Evaluation

This Section describes an FPGA implementation of the proposed architecture together with energy-delay-square-product (ED^2P) results relative to an architecture without the addition of the prefetch control and buffering.

4.1. Data prefetch buffering

The control word format is illustrated in Figure 4. After compilation control words are introduced into the code by auxiliary software. Our software passes over a binary version of the code in order to identify those instructions that require data prefetches, including load/store operations that have pointer-based data references. The software then inserts control words before those instructions identified, so that during execution, the data prefetch unit can fetch the data beforehand based on one or more pre-calculated addresses. In Figure 5, our software identifies that the address 0010b contains a load direct from data space instruction. Consequently, a control word is placed at 0010b. Within the control word is the address of the data that is FFh or 11111111b, according to the prefetch instruction format of Figure 4. The

software continues its pass through the compiled binary until all the control words have been introduced ready for data prefetching.

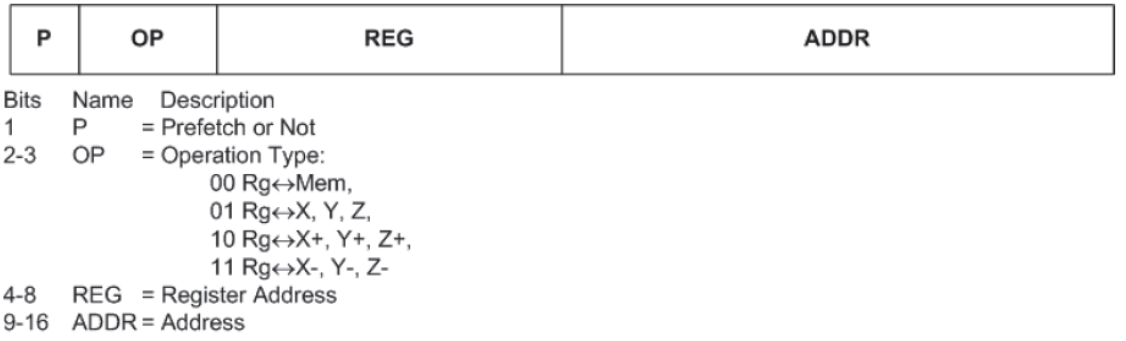

Fig. 4. Control Word format

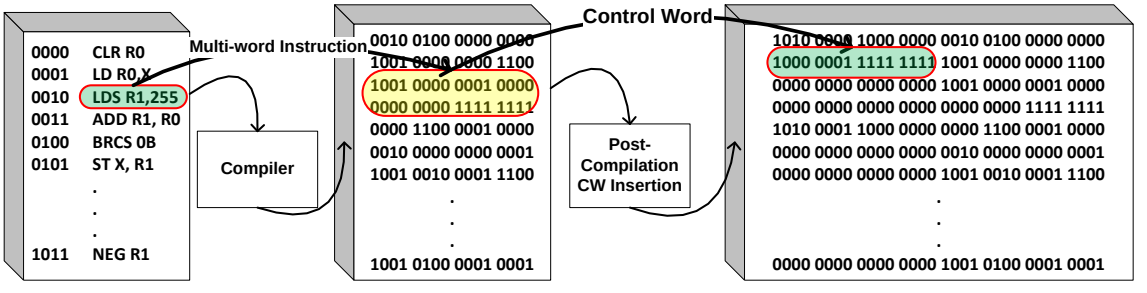

Fig. 5. Control Word insertion during compile time for data prefetch

Thus, control words are treated separately from instructions in the existing Instruction Set Architecture (ISA). Therefore, no modification is made to the ISA. When an augmented instruction is fetched, Figure 5, the lower 16-bits are passed to the instruction decoder; whereas the upper 16-bits are decoded as described in Figure 4. The first bit shows if a prefetch operation is required or not; bits 2-3 define the type of operation i.e. register to memory; or indexed operation. Bits 4-8 define the address of the register; and the rest of the bits show the address from where the data has to be fetched. The source location, as shown in the control word, is accessed and stored in the buffers shown in Figure 1.

4.2. *Hardware implementation*

In order to introduce the additional buffer, the implementation took advantage of an existing

VHDL implementation of an AVR core[d]. The existing open-source, cycle-accurate VHDL hereby is referred to as the Original architecture. The version with the additional buffer is called the Buffered architecture in the following set of comparisons aimed at highlighting the gain from introducing the prefetch buffer mechanism. To clarify further, the Original and Buffered architectures are both built upon an ATMEL AVR core, the VHDL Atmega103(L) model [40]. The instruction throughput of the core approaches 1 MIPS per MHz, due to single machine cycle rate of execution. The 32 general-purpose registers are all connected directly to the ALU. As a result, two independent registers can be accessed in one machine cycle. There are separate SRAM instruction and data memories. The data memory size is 4k×8 bits.

The proposed architecture was implemented with Xilinx ISE Design Suite ver. 13.1 (available from http://www.xilinx.com) verified by simulation by means of Mentor Graphics Modelsim ver. 6.1 (available from http://www.mentor.com/products/fv/modelsim/). The implementation took place on a Xilinx XC3S500E FPGA. For cache-based memory structures, the CACTI tool [41] is commonly used. However, as a prefetch buffer rather than cache is employed, it was found to be preferable to utilize the Xilinx XPower Estimator (XPE) tool (available from http://www.xilinx.com). XPE estimates the worst-case power consumption by means of a spreadsheet. As is normal practice for FPGA designs so as to discover whether the energy consumption would be within the limits set for an embedded application, the XPE was applied at an early stage in the design process. As on-chip Dual Ported (DP)-RAM was to be added to the core, the interest was in discovering its impact. Notice though that the XPE takes account of all components of the design in its estimate.

While the Original unbuffered architecture consumed 2940 Look-up-Tables (LUTs) and 797 Logic slices, the proposed Buffered architecture consumed an additional 2984 LUTs and 87 Logic Slices. All the same, both architectures could each fit on the XC3S500E FPGA and each could operate up to a maximum 33 MHz clock frequency. The addition of control words, when present, requires an increase in the memory size for their storage from 16-bit to 32-bit in width (i.e. the code size doubles), consequently the area of the processor core has increased. However, area optimization is not the aim of this paper. Section 4.3 confirms the gains from the additional

[d] See R. Lepetenok. The VHDL implementation of AVR Open Core. Aug. 2014, available from

area, which still does not take up the resources of a quite small FPGA.

In general, a user's application code could be instrumented with control words directly by the compiler or immediately after compilation into binary object code. As previously outlined, the experimental procedure was to employ our post-compilation software to identify locations and insert appropriate control words.

4.3. Findings

To test the proposed design, we selected routines from the MiBench [42] benchmark suite. Though MiBench offers 38 benchmarks, a number of these are inappropriate to the anticipated workload. For example, we did not select routines from the office sub-set of benchmarks, which includes the ghostscript postscript renderer, the ispell spelling checker, and the rsynch text-to-speech synthesizer. Instead routines that could reasonably be run on a low-end processor were chosen. Overall the per-cycle energy consumption increases somewhat by 13 mW compared to the 100 mWs of the Original architecture. However, this would be to miss the point, as energy efficiency improves as a result of employing the Buffered architecture because the throughput increases. That is the number of cycles required to execute a routine decreases.

The chosen routines are (1) Basic Math, (2) Quick Sort, (3) CRC-32, (4) FFT, (5) Dijkstra, (6) Matrix Multiplication, and (7) FIR Filter. The benchmark results show normalized energy consumption to be over 5.85% (i.e. a negative sign shows energy consumption on an energy savings plot) for FFT, and the most savings i.e. 11.5% for Matrix Multiplication application (see Figure 6(a)). This variation of energy consumption from a positive energy savings to negative for a range of benchmarks is due to the fact that the applications comprise of varying amount of instructions that take advantage of the proposed architecture. For an application with minimal amount of Jumps and Branches the proposed architecture will consume more energy and vice versa.

For timing performance Matrix Multiplication shows savings in access of 21%, whereas the least amount of time savings i.e. 7.3% occurs for the Basic Math benchmark (see Figure 6(b)). An overall effect of energy and timing takes place according to ED^2P in Figure 6(c), in which Matrix Multiplication achieves normalized ED^2P of 0.54 and the highest one is 0.9 for the Basic Math application. Rather than relative percentages, for completeness Table 1 records the actual

---

http://opencores.org/

figures recorded in the experiments of Figure 6. Due to the large dynamic range of the results, with the figures for the Basic Math benchmark dominating the others, the tabular format is preferable.

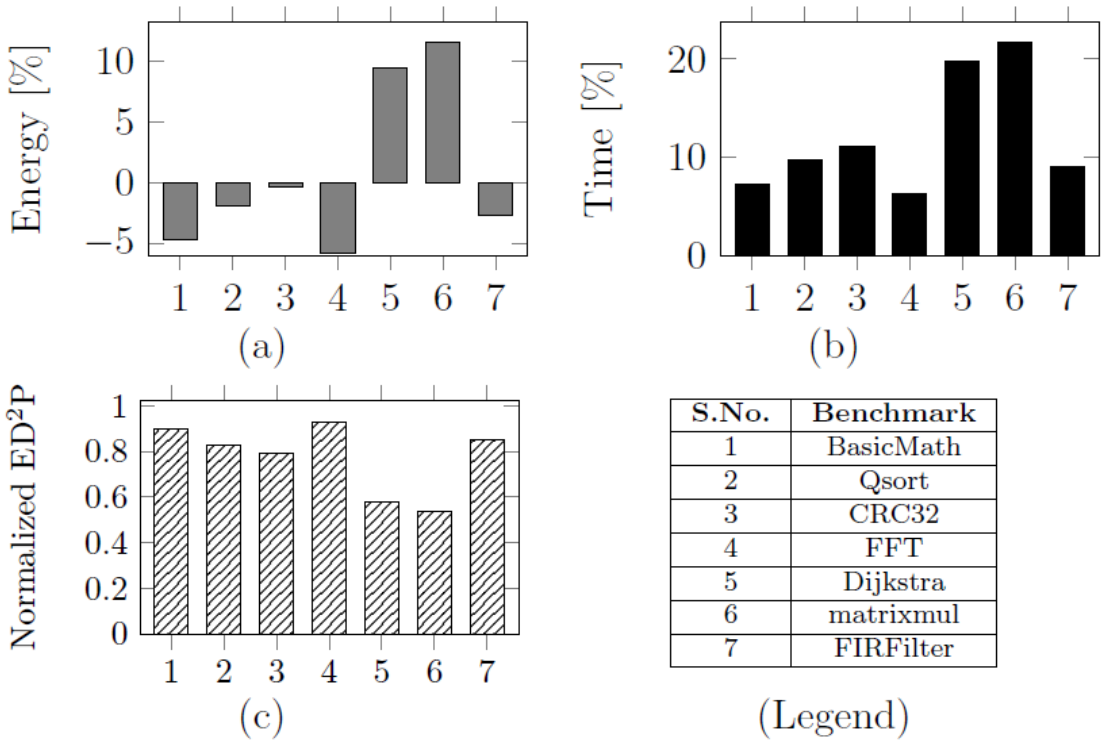

Fig. 6: Buffered Architecture (a) Energy, (b) Time Savings, and (c) Normalized ED^2P

Table 1: Energy and Time Comparison of Original and Buffered Architectures

	Energy [j]		*Time [s]*	
Benchmark	*Original Architecture*	*Buffered Architecture*	*Original Architecture*	*Buffered Architecture*
BasicMath	2.80E+00	2.93E+00	5.78E+01	5.36E+01
Qsort	2.15E-04	2.19E-04	4.44E-03	4.00E-03
CRC32	2.87E-05	2.88E-05	5.91E-04	5.25E-04
FFT	7.30E-04	7.73E-04	1.51E-02	1.41E-02
dijkstra	3.82E-03	3.46E-03	7.87E-02	6.31E-02
matrixmul	9.04E-06	8.01E-06	1.87E-04	1.46E-04
FIR Filter	2.90E-03	2.97E-03	5.97E-02	5.43E-02

5. Conclusion

This paper introduces a software-controlled prefetch data buffering design, targeted at low-cost embedded microprocessors. Such processors as the Atmel AVR and Microchip PIC have single cycle latency. Consequently, increased throughput results in significant gains in energy efficiency. Thus, the proposed design demonstrates greater net energy efficiency according to the ED^2P metric, with efficiency being 54% to 90% of that of an equivalent unbuffered architecture, i.e. 10% to 46% more efficient As this class of processor does not possess cache memory, tuning the caches in some way is not available as a means of improving energy efficiency, while, in contrast, prefetch buffering can lead to gains. As these processors are now more likely to be embedded in battery-operated devices, the proposal in this paper is timely. In fact, the architecture shows a 6%−20% improvement in execution times when tested over seven widely deployed bench-marks for this class of low-end processor. Basic algorithm such as the CRC and FFT gain from this proposal, which implies that there will be many embedded applications that benefit from this innovation. Further research will investigate, for a set of applications, the trade-off between area increase from the use of a device with additional on-chip memory and the performance gains shown in this paper.

References

[1] J. Laros III, K. Pedretti, S. Kelly, W. Shu, K. Ferreira, J. Vandyke, C. Vaughan, Energy delay product, in: Energy-Efficient High Performance Computing, Berlin, Germany: Springer Verlag, 2013, pp. 51–55.

[2] J.M. Rabaey, M. Pedram, Low power design methodologies, Boston, MA: Kluwer Academic Publ., 1996.

[3] C. Zhang, F. Vahid, W. Najjar, A highly configurable cache architecture for embedded systems, in: Proc. IEEE 30th Ann. Int. Symp. Computer Architecture, 2003, pp. 136–146.

[4] A. Malik, B. Moyer, D. Cermak, A low power unified cache architecture providing power and performance flexibility, in: Proc. IEEE Int. Symp. Low Power Electronics and Design, 2000, pp. 241–243.

[5] S. Segars, Low power design techniques for microprocessors, in: Proc. IEEE Int. Solid-State Circuits Conf. Tutorial, 2001.

[6] T.C. Mowry, M.S. Lam, A. Gupta, Design and evaluation of a compiler algorithm for prefetching, ACM Sigplan Notices 27, (1992) 62–73.

[7] J. Turley, Atmel AVR brings RISC to 8-bit world, Microprocessor Report 11, (1997) 1–4.

[8] T. Wilmshurst, Designing embedded systems with PIC microcontrollers: Principles and applications (2nd ed.), Burlington, MA: Newnes Publishers, 2009.

[9] A. J. Martin, A. Lines, R. Manohar, M. Nystrom, P. Penzes, R. Southworth, U. Cummings, and T. K. Lee. The design of an asynchronous MIPS R3000 microprocessor, in: Proc. of the IEEE 17th Conf. on Advanced Research in VLSI, 1997, pp. 164-181.

[10] R. Gonzalez, B.M. Gordon, M.A. Horowitz, Supply and threshold voltage scaling for low power CMOS, IEEE Journal of Solid-State Circuits 32, (1997) 1210–1216.

[11] A. J. Martin. Towards an energy complexity of computation. Info. Processing Letters 77, (2001) 181-187.

[12] A. J. Martin, M. Nyström, and P. I. Pénzes. ET2: a metric for time and energy efficiency of computation, in: Power Aware Computing, R. Graybill and R. Melhem (Eds.), Norwell, MA, USA: Kluwer Academic Publishers, 2002, pp. 293-315.

[13] S. Eyerman, L. Eeckhout, and K. D. Bosschere. The shape of the processor design space and its implications for early stage explorations, in Proc. 7th WSEAS Int. Conf. on Automatic Control, Modeling and Simulation. 2005.

[14] D.M. Brooks, P. Bose, S.E. Schuster, H. Jacobson, P.N. Kudva, A. Buyuktosunoglu, J.D. Wellman, V. Zyuban, M. Gupta, P.W. Cook, Power-aware microarchitecture: Design and modelling challenges for next-generation microprocessors, IEEE Micro 20, (2000) 26–44.

[15] M. Qadri, N. Qadri, K. McDonald-Maier, Software controlled data prefetch buffering, Patent application no. 13/918,407, United States Patent and Trademark Office, June 2013.

[16] M.Y. Qadri, N.N. Qadri, M. Fleury, K.D. McDonald-Maier, Software controlled instruction prefetch buffering for low-end processors, Journal of Circuits, Syst., and Computers 24, (2015).

[17] J. Henkel, S. Parameswaran, Designing embedded processors: A low power perspective, Berlin, Germany: Springer Verlag, 2007.

[18] J. Xie, J. Mao, A novel hardware prefetching scheme exploiting 2-D spatial locality in multimedia applications, in: Proc. of IEEE 9th Int. Conf. on ASICs, 2011, pp. 192–195.

[19] Y. Guo, S. Chheda, I. Koren, C. Krishna, A. Moritz, Energy characterization of hardware-based data prefetching, in: Proc. of IEEE Int. Conf. on Computer Design, 2004, pp. 518–523.

[20] J. Tang, S. Liu, S. Gu, C. Liu, J.L. Gaudiot, Prefetching in embedded mobile systems can be energy-efficient, IEEE Computer Architecture Letters 10, (2014) 8–11.

[21] M. Kahn, M.A. Laurenzano, J. Mars, E. Hagersten, D. Black-Schaffer, AREP: Adaptive Resource Efficient Prefetching for maximising multicore performance, in: Proc. of IEEE Int. Conf. on Parallel Architecture and Compilation, 2015, pp. 367-378.

[22] M. Li, G. Chen, Q. Wang, Y. Lin, P. Hofstee, P. Stenstrom, D. Zhou, PATer: A hardware prefetching automatic tuner on IBM POWER8 processor, IEEE Computer Architecture Letters 11, (2015) 1-4.

[23] D. Guttman, M.T. Kandemir, M. Arunachalam, R. Khanna, Machine learning techniques for improved data prefetching, in: Proc. of IEEE Int. Conf. on Energy Aware Computing Systems & Apps., 2015, pp. 1-4.

[24] W. Worley Jr., W. Bryg, Cache memory consistency control with explicit software instructions, Patent no. 4,713,755, United States Patent and Trademark Office, June 1985.

[25] V. Oklobdzija, T. Ling, Instruction prefetch buffer control, Patent no. 4,714,994, United States Patent and Trademark Office, December 1985.

[26] A.K. Ray, K.M. Bhurchandi, Advanced microprocessors and peripherals: architecture, programming and interfacing, Noida, India: Tata McGraw-Hill Publishing Company, 2000.

[27] C.H. Chi, Data prefetching under the control of instruction cache, Patent no. 5,784, 711, United States Patent and Trademark Office Patent, June 1998.

[28] D. Emberson, Tunable software control of Harvard architecture cache memories using prefetch instructions, Patent no. 5,838,945, United States Patent and Trademark Office, November 1998.

[29] J. Kim, K.V. Palem, W.F. Wong, A framework for data prefetching using off-line training of Markovian predictors, in: Proc. of IEEE Int. Conf. on Computer Design, 2002, pp. 340–347.

[30] Y. Chen, S. Byna, X.-H. Sun, R. Thakur, W. Gropp, Hiding I/O latency with pre-execution prefetching for parallel applications, in: Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis, 2008, Article No.40.

[31] I. Ganusov, M. Burtscher, Future execution: A hardware prefetching technique for chip multiprocessors, In: Proc. Int. Conf. on Parallel Architectures and Compilation Techniques, 2005, pp.350–360.

[32] 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.

[33] T. Chen, T. Zhang, Z. Sura, M.G. Tallada, Prefetching irregular references for software cache on cell, In: Proc. 6th Ann. IEEE/ACM Int. Symp. Code Generation and Optimization, 2008, pp. 155–164.

[34] Y. Solihin, J. Lee, J. Torrellas, Using a user-level memory thread for correlation prefetching, in: Proc. 29th Ann. Int. Symp. Computer Architecture, 2002, pp. 171–182.

[35] M. Annavaram, J.M. Patel, E.S. Davidson. Data prefetching by dependence graph precomputation, in: ACM Ann. Int. Symp. on Computer Architecture, 2001, pp. 52–61.

[36] C.-K. Luk, Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors, in: ACM Ann. Int. Symp. on Computer Architecture, 2001, pp. 40–51.

[37] S. Byna, Y. Chen, X.-H. Sun, Taxonomy of data prefetching for multicore processors, Journal of Computer Science and Technology 24, (2009) 405–417.

[38] A.M. Holberg, A. Saetre, Innovative techniques for extremely low power consumption with 8-bit microcontrollers, White Paper: 7903A- AVR-2006/02, 2006.

[39] M. Iliopoulos, T. Antonakopoulos, Reconfigurable network processors based on field programmable system level integrated circuits, in: Field-Programmable Logic and Applications, 2000, pp. 39-47.

[40] T. van Leuken, A. de Graaf, H. Lincklaen Arriens, A high-level design and implementation platform for IP prototyping on FPGA, in: Proc. ProRISC IEEE 15th Ann. Workshop on Circuits, Syst. and Signal Process., 2004, pp. 68–71.

[41] G. Reinman, N. Jouppi, CACTI 2.0: An integrated cache timing and power model, Tech. report, Compaq Computer Corporation Western Research Laboratory, 2000.

[42] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, R.B. Brown, MiBench: A free, commercially representative embedded benchmark suite, in: Proc. IEEE Int. Workshop on Workload Characterization, 2001, pp. 3–14.