An Energy Efficient Instruction Prefetching Scheme for Embedded Processors

Ji Gu Hui Guo

University of New South Wales, Australia
{jigu,huig}@cse.unsw.edu.au

Technical Report
UNSW-CSE-TR-1006
February 2010

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

Existing instruction prefetching schemes improve performance with significant energy sacrifice, making them unsuitable for embedded and ubiquitous systems where high performance and low energy consumption are all demanded. In this paper, we reduce energy overhead in instruction prefetching by using a simple prefetching hardware/software design and an efficient prefetching operation scheme. Two approaches are investigated: one, Decoded Loop Instruction Cache based Prefetching (DLICP) that is most effective for loop intensive applications; two, enhanced DLICP with the popular existing Next Line Prefetching (NLP) for applications of a moderate number of loops. Our experimental results show that both DLICP and enhanced DLICP deliver improved performance at greatly reduced energy overhead. Up to 21% performance can be improved by the enhanced DLICP at about 3.5% energy overhead, as in comparison to the maximal 11% performance improvement and 49% energy overhead from NLP.

1 Introduction

On-chip cache has been widely used in modern microprocessors to bridge the speed gap between the processor and main memory. Cache exploits the spatial and temporal locality of memory reference to avoid the long latency of memory access from the processor. A high cache hit ratio plays a vital role in the overall system performance. This is especially essential for the instruction cache (I-cache) due to frequent instruction fetch operations; An instruction cache miss will cause the processor stall, hence slowing down the system.

Plenty of techniques have been proposed to reduce I-cache misses. Among them is the instruction prefetching [1][2]- fetching instructions from memory into the cache before they are used so that cache misses can be avoided. However, existing instruction prefetching schemes mainly focus on improving cache performance, often sufferring significant energy losses due to a large amount of wasteful over-prefetching operations and/or complicated prefetching hardware components. Nevertheless, low energy consumption is one of the most important design constraints for embedded and ubiquitous systems, especially in the application domain of mobile and ubiquitous computing.

In this paper, we aim to reduce energy overhead in instruction prefetching by using a simple prefetching hardware/software design and an efficient prefetching operation scheme. We investigate two approaches: the decoded loop instruction cache based prefetching (DLICP) and the enhanced DLICP.

The decoded loop instruction cache (DLIC) originates from the decoded instruction buffer (DIB) proposed in [3]. It is a small *tag-less* cache residing between the instruction decoder and the execution unit in the microprocessor to store decoded loop instructions so that fetching and decoding the same set of instructions for the following loop iterations can be avoided, hence reducing energy dissipation in the processor.

We extend this energy-saving technique to instruction prefetching by overlapping the execution of decoded loops with fetching instructions to the cache from memory so that most instructions are available in the cache when they are executed. This approach is effective for loop intensive applications. For applications with a small amount of loops, we enhance the design with the existing Next Line prefetching (NLP) scheme, which has been proved efficient in cache miss reduction for applications with a dominant sequential instruction execution flow [4].

The rest of the paper is organized as follows. Section 2 reviews some existing instruction prefetching methods for cache performance optimization. The structure and working principle of our DLICP scheme is given in Section 3, where the hardware/software codesign and the prefetching operation scheme are detailed. Section 4 presents the experimental setup, results and related discussions. The paper is concluded in Section 5.

2 Related Work

A variety of prefetching techniques have been proposed to improve the traditional instruction cache miss for performance improvement. These can be classified as software based prefetching and hardware based prefetching.

Software prefetching schemes [5][6][7] rely on the compiler to insert prefetch instructions into the program code before the application is executed, which requires a known memory access behavior and a dedicated compiler.

Hardware prefetching is transparent to the software and exploits the status of the program execution to dynamically prefetch instructions for future use. It is more flexible than the software based approach but incurs hardware overhead and increases the complexity of the processor architecture. The hardware based approaches mainly include sequential prefetching and non-sequential prefetching.

Next-Line prefetching [4] utilizes the spatial locality of the program execution is one of the sequential prefetch approaches. On an instruction cache miss, it fetches the current cache miss line and sequentially prefetches the next lines to reduce possible cache misses. The adaptive sequential prefetch method [8] prefetches varying number of cache lines based on different program execution behaviors.

The stream buffer prefetch [9] is another sequential prefetch approach designed specifically for the direct-mapped cache (where conflict cache misses may become a key problem). This approach places the prefetched cache line into a stream buffer and only writes it to the cache when it is actually referenced by the processor to reduce possible conflict cache misses. A downside of this approach is that, in case a referenced data item is missing in both the cache and buffer, the buffer will be flushed by the next cache line. This wastes many prefetched cache lines and makes the prefetch scheme ineffective.

Sequential prefetching is efficient for programs with sequential execution. To handle applications with a large amount of branches, Pierce et al [10] proposed a non-sequential Wrong-Path scheme that prefetches instructions for all branch directions. Stride-directed prefetching [11][12] is another non-sequential approach, which is based on the observation that if a memory address is accessed, the memory location some stride away from the address is likely to be accessed soon. This method examines the memory access behavior for such a potential stride. If the stride is found, cache lines to be prefetched are offset by such a distance. Shadow directory prefetching [13] associates each cache line with a shadow address that points to the next possible cache line. When a cache line is accessed and hit, its shadow-address pointed cache line will be prefetched.

Some non-sequential schemes utilize cache miss prediction for instruction prefetch. Joseph and Grunwald [14] proposed a prediction based prefetching technique, where a Markov model is used to correlate a stream of instruction misses. The predicted miss addresses are stored in the prediction table indexed by the related miss address. The instruction prefetch is triggered when a cache miss occurs. The tag-correlating prefetching [15] is a similar technique. However, they only store the tag bits of the missing instruction addresses to reduce its size. The fetch di-

rected instruction prefetching scheme [16] uses a branch predictor to predict the program execution stream. The branch predictor generates a queue of prefetching targets and the prefetched cache lines are initially stored in an additional prefetch buffer. The prefetched instructions are only written into the cache when they are referenced. A cache miss due to the misprediction has to flush the prefetching target queue and the prefetched instruction buffer.

In branch history guided prefetching [17], Srinivasan et al. propose to correlate the instruction cache misses with branch instructions based on their execution history. They store the correlations in the prefetch table indexed by the address of branch instruction. The prefetches are triggered by the branch instructions when the same correlations are found later during the program execution. Zhang et al. propose the execution history guided prefetching [18] where they correlate the cache misses with every instruction according to the execution history. This scheme has finer granularity than [17]; any instruction (not only the branch instructions) can potentially be the prefetching trigger to allow more prefetching opportunities and effectiveness.

The above hardware-based prefetching schemes are popular techniques for cache performance improvement of the general purpose computer architecture. Most schemes are impractical in mobile or ubiquitous embedded systems, where low energy consumption is of ultimate importance.

In this paper, we aim to reduce energy consumption in instruction prefetching and further improve the prefetching efficiency by effectively paralleling instruction prefetching with the processor execution. We compared our approach with the Next Line Prefetching, which is the most power effective approach among existing prefetching schemes. Experimental results show that our design is more efficient in both power reduction and performance improvement.

3 DLIC-based Instruction Prefetching

The system architecture with our proposed DLIC-based prefetching scheme is illustrated in Fig. 3.1. It contains a five-stage pipeline processor, with the decoded loop instruction cache (DLIC) sitting between the instruction decode (ID) stage and execution (EXE) stage; a two-level memory hierarchy, with the separate on-chip instruction cache and data cache, and off-chip main memory, and a memory controller. The memory controller controls memory access in two fashions each operated by *Fetcher* and *Prefetcher*, respectively. For the normal processor execution, *Fetcher* retrieves instructions from cache, or from memory if there is a cache miss; During execution of a decoded instruction loop, *Prefetcher* fetches instructions that are to be executed after the loop but are not yet in the cache.

It is worth noting that the DLIC structure implemented in this paper has a more ability than the normal decoded instruction buffer. It allows to cache loops of indeterminate loop counts, rather than only cache loops of a known number of iterations in the traditional decoded instruction buffer design. The hardware/software design

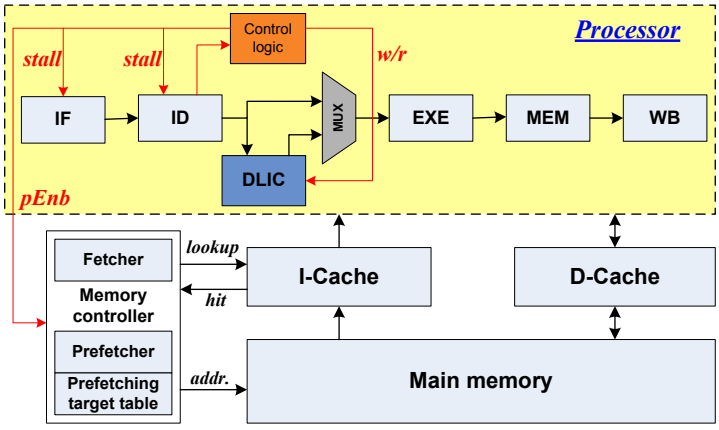

Figure 3.1: Architecture for DLIC-based instruction prefetching.

for the decoded loop instruction cache and an efficient strategy for prefetching operations are elaborated in the following subsections.

3.1 Hardware/Software Design of Decoded Loop Instruction Cache

For a given application, its loop execution behavior can be easily extracted, which makes it possible to use software approach to storing decoded loop instructions so that associated hardware component can be simplified, hence reducing hardware area cost and energy consumption. We aim at basic loops that are frequent and small in size, which is often the case in most embedded and ubiquitous systems [19]. For such loops, we define two special instructions:

$$\begin{cases} slp\ f,\ \#iterations \\ elp\ f,\ rs,\ rt \end{cases} \tag{3.1}$$

Instruction *slp* will be inserted at the top of loops, and instruction *elp* the end of loops, to control to store and execute the decoded loop instructions.

The formats of these two instructions (based on SimpleScalar PISA [20] architecture) are illustrated in Fig. 3.2(a) (More explanation will follow). The hardware components related to the cache operation are given in Fig. 3.2(b).

Apart from the multiple entries in the DLIC cache for storing decoded instructions, there are two special registers: R_{LC} (for loop count in terms of the number of loop iterations) and R_{LS} (for loop size in terms of the number of instructions in the loop), and two special counters: C_{instr} (for counting number of executed instructions in a loop) and C_{iter} (for counting number of executed loop iterations). The counters are initialized for each loop execution.

Depending on whether the loop count is known at compile time, the instruction pair will have two different treatments, which is controlled by flag f in the instruc-

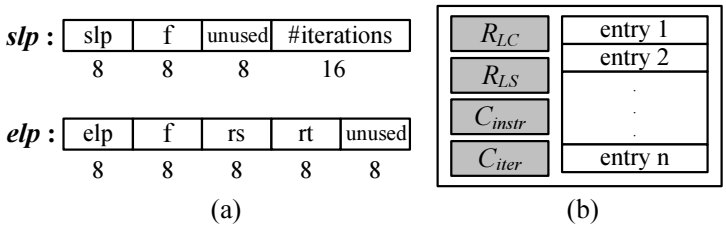

Figure 3.2: (a) Formats of special instructions, (b) components in DLIC cache.

tion.

CASE 1: Loops with Determinate Loop Counts

For the loop whose iteration count is known before execution, the flag *f* of the *slp* instruction is set to *0xFF* and the flag in instruction *elp* is set to *0x00*. Examples of such loops are given in Fig. 3.3 (a) and (b), where both *while-loop* and *for-loop* have a determinate loop count at compile time. Fig. 3.3 (c) demonstrates the corresponding instructions with the loop count equal to 10.

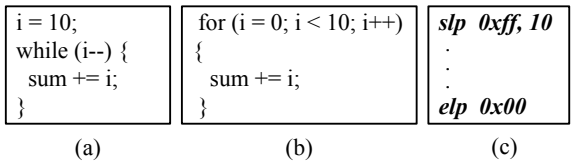

Figure 3.3: Determinate loop count: (a) while-loop, (b) for-loop, (c) special instruction pair.

When executing *slp* with the *0xFF* flag value, the processor saves the *#iterations* value given in the instruction in register R_{LC} and enables the decoded instruction caching function. For each instruction executed in the first loop iteration, its decoded instruction value is sequentially stored in the DLIC cache and counter C_{instr} is incremented by 1. Therefore, at the end of the first iteration when instruction *elp* is encountered, C_{instr} records the number of instructions in the loop. This number is then saved in register R_{LS} (the register for loop size) and C_{instr} is reset to 0 for the next loop iteration. For each loop iteration, counter C_{iter} is incremented by 1 until it reaches the loop count value stored in register R_{LC}, which means the decoded instruction loop is finished and the processor is back to the normal execution state.

CASE 2: Loops with Indeterminate Loop Counts

For loops with unknown loop counts at compile time, as the examples shown in Fig. 3.4(a) and (b), the *f* flag for instruction *slp* is set to *0x00* and for instruction *elp* instruction, it is set to *0xFF*.

During executing *slp* with the *0x00* flag value, register R_{LC} and counter C_{iter}

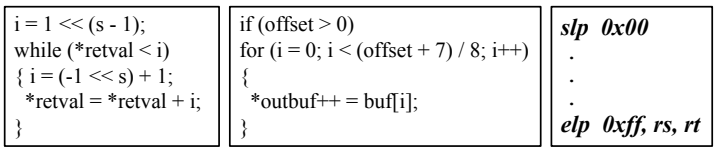

Figure 3.4: Indeterminate loop count: (a) while-loop, (b) for-loop, (c) special instruction pair.

are not used. But register R_{LS} and counter C_{instr} work in the same way as in Case 1. R_{LS} stores the total number of decoded instructions that should be executed for each of loop iterations and C_{instr} counts the number of executed instructions for the current iteration. Unlike in Case 1, where *elp* is executed only once for a loop, *elp* in Case 2 will be executed at the end of each iteration to determine whether the loop is finished. When the condition that registers *rs* and *rt* have the same value is satisfied, the loop execution is terminated.

It is worth to note that by using the special instructions and related hardware design, the loop control instructions in the original program code can be removed, reducing the total instruction count of the application, hence improving performance.

3.2 Instruction Prefetching Control Scheme

Due to different program control flows, some prefetched instructions may not be actually used, which not only wastes time but also incurs unnecessary energy lost.

To improve the prefetching efficiency, we try to prefetch instructions on the execution path of high operating frequency. Take the execution control flow shown in Fig. 3.5 (a) as an example. It contains 7 basic blocks; each block consists of a sequence of instructions. Blocks *B1* and *B6* are loops (*L1, L2*), whose decoded instructions will be cached; between the two loops are four basic blocks connected by two branches, which leads to various execution paths. The frequencies of the branches to different targets are shown in the flow. Blocks *B2*, *B4*, *B6* form an execution path with a higher execution frequency. We, therefore, want to prefetch the instructions in those blocks during the *L1* execution.

The frequent execution path for each decoded loop can be found by profiling and is stored in a table, called prefetching target table (PTT). Each entry in the table associates with a decoded loop and holds the information of the instruction blocks on the frequent execution path. For each instruction block, its start address and size in terms of the number of instructions are provided so that all instructions of the block in the memory can be located by *Prefetcher*. Fig. 3.5 (b) illustrates the PTT table for the execution flow in Fig. 3.5(a). The first block in each table entry is always the immediate block of the related loop.

During execution of a decoded loop, we use the PTT table to find the instruc-

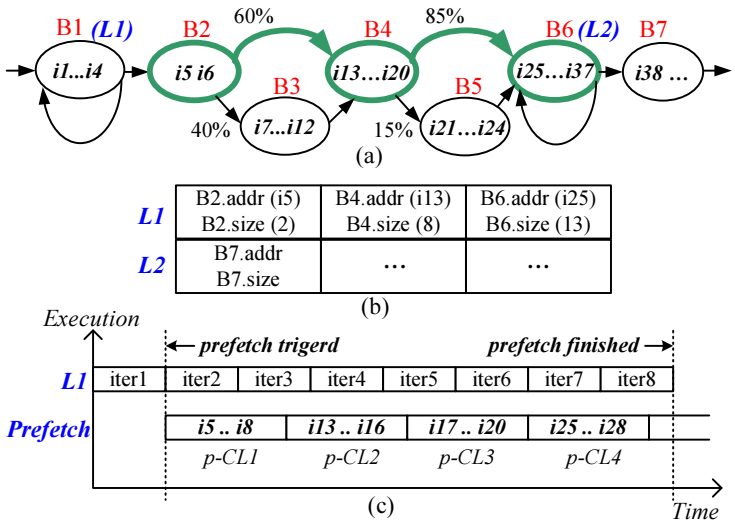

Figure 3.5: (a) An example of execution control flow, (b) the corresponding prefetching target table, (c) execution diagram of the DLICP scheme.

tions on the frequent execution path and to prefetch them from memory if they are not available in the cache. To explain, we use the execution of *L1* as an example (see Fig. 3.5 (c) for the execution timing diagram).

Assume L1 has 8 iterations. After the first iteration, all instructions in the loop have been decoded and saved in the DLIC cache. The processor is now in the state for the decoded loop execution, where the Program Counter (PC) is temporarily disabled with its value statically pointing to the instruction immediately after the loop (i.e. instruction *i5* in the example) during the whole decode loop execution.

When the L1 execution enters the second iteration, the instruction prefetching is triggered. The *Prefetcher* (see Fig. 3.1) searches the PTT table (based on the current PC value) for the first instruction block on the frequent execution path. The block is then checked to see whether instructions of the block are available in the cache, if not, the related cache line(s) will be prefetched; otherwise, continue to the next basic block. This process is repeated for the rest of the instruction blocks in the PTT entry until the execution of the decoded loop is finished, as illustrated in Fig. 3.5 (c), where a 4-word cache line is assumed and each instruction is one-word long. Here we also assume all instructions on the frequent execution path are not available in the cache and are prefetched in four cache lines (denoted by *p-CL1* to *p-CL4* inFig. 3.5 (c)).

As can be seen, the cache line size and duration of the decoded loop execution affect the number of instructions that can be prefetched. The larger the cache line and the longer the decoded loop execution, the more instructions can be fetched. But the large cache line may allow *Prefetcher* to fetch instructions that are not needed; for example, instructions *i7, i8* in block *B3* were brought along by prefetching the first cache line *p-CL1*.

3.3 Enhanced DLIC Prefetching

With the DLICP design, the prefetching operation is restricted by the availability of basic loops and distribution of these loops in the program. For an application with a small number of such loops, or the loops are located at the end of the program, only limited prefetching operations can be performed, hence limited cache miss savings. This limitation can be circumvented by incorporating the existing NLP prefetching scheme that always prefetches instructions on a cache miss.

It must be emphasized that the cache miss saving from DLICP does not incur performance overhead because of the parallel prefetching operations (see Fig 3.5(c)), while the saving from NLP is accompanied with the cache miss performance penalty. With DLIC, such penalties can be reduced. Therefore, combining both DLICP and NLP, we can achieve high cache miss reduction with a smaller performance overhead. This NLP scheme is implemented in the *Fetcher* memory control component in our system (see Fig 3.1).

4 Experimental Results

To examine the efficiency of our DLIC-based prefetching scheme, we applied it to a set of applications from Motorola's Powerstone [21] and MiBench [22] benchmark suites, which are widely used in the embedded application domain of automotive control, image processing, audio/vedio coding. The reference input data of each program are used in our experiments.

4.1 Experimental Setup

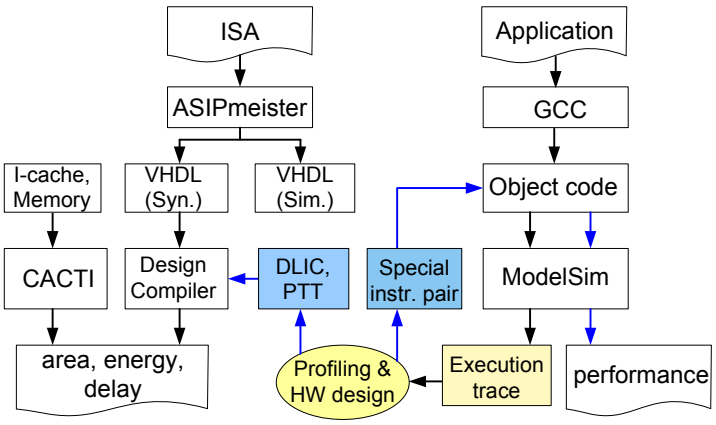

Figure 4.1: Experimental setup.

Fig. 4.1 shows our experimental setup. We selected the Simplescalar PISA [20] as the target processor instruction set architecture. An in-house VHDL model of PISA processor, generated using the commercial tool ASIPMeister [23], was

used as the platform for the application simulation. The experiment started with a given application written in C, compiled by the simplescalar-gcc cross compiler and then simulated on the VHDL model. The loop behavior of each application was extracted from the execution instruction trace, based on which the frequent basic loops were modified with special instruction pairs for decoded loop instruction caching. Hardware designs of the related prefetching schemes were then integrated to the processor model for evaluating their logic cost and energy overhead with Synopsys DesignCompiler. The area and energy consumption for the I-cache and main memory were obtained from CACTI 5 [24], which is a widely used cache and memory model for evaluation of access time, area, and energy consumption.

In our experiment, we assumed the on-chip I-cache was 2-way set associative of 2K bytes, with the line size of 32 bytes. The small 2K I-cache is suitable for ubiquitous embedded systems where the costs are very restrained.

4.2 Performance Improvement

Performance can be evaluated in terms of total execution time, which is the product of the total number of clock cycles used and the clock cycle time when running an application. Cache performance affects the execution clock cycles. Therefore, we first investigate the cache performance.

Cache Misses and Miss Penalties in Prefetching

With prefetching, apart from the two normal cache states (*cache hit* and *cache miss*), there is an extra case – the data requested is not yet in the cache, but is on the bus, being transferred from the memory to the cache by prefetching. We refer to this special case as **false miss** since it does not incur a new memory access.

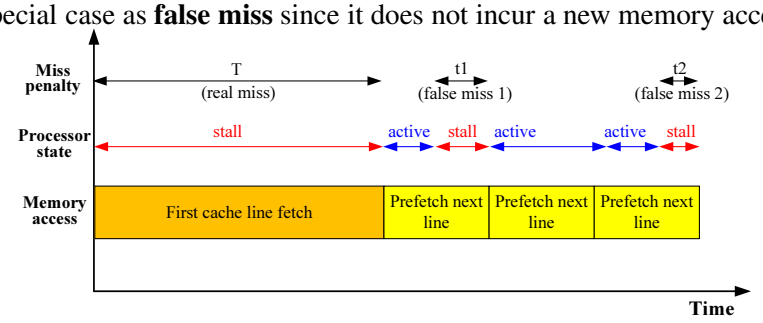

Figure 4.2: Cache miss penalty

Unlike a real cache miss, which has a fixed miss penalty, the false miss has a reduced and varying miss penalty due to the parallel operations of processor and prefetching, as illustrated in Fig. 4.2, where on a real miss, the miss penalty is fixed (T); for the false misses, the miss penalty varies (t_1 and t_2 in the example), depending on the relative time needed for the processor to finish available instructions during the next line prefetching.

Table 4.1 lists the measurements of real cache misses (namely, false misses being excluded) when running different applications under the three prefetching schemes (columns 3-5). For a comparison, the baseline design without prefetching is also given in the table (column 2). The baseline is a 2 way associative instruction cache of 2K bytes and has a cache line of 32 bytes. The cache access time is assumed as 1 clock cycle. The last row shows the average value. The normalized cache miss ratios as compared to the baseline design, are plotted in Fig. 4.3.

Table 4.1: Cache Misses

	Baseline	**NLP**	**DLICP**	**Ehd' DLICP**
blit	59	34	49	31
crc	57	33	45	29
dijkstra	19774	12440	17845	17828
g3fax	1072	914	1028	992
jpeg	66186	52192	64849	64796
qsort	83	55	62	43
rc4	98	53	69	62
rijndael	1020	600	684	656
salsa	446	280	265	242
seal	1908	1278	1123	1114
sha	3197	2163	2581	2563
AVG	8536.4	6367.5	8054.5	8032.4

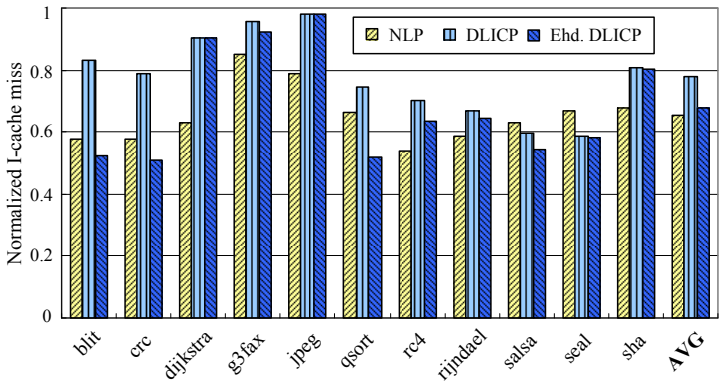

Figure 4.3: Normalized I-cache miss ratio

As can be seen from the measurements, NLP provides a better cache-miss reduction (an average of 34.6%) than DLICP (22.1%). This is due to the insufficient basic loops available. Therefore, fewer prefetching operations in DLICP were performed, hence less cache misses reduced. NLP is not restricted by the loop patterns in the application program. Two exceptions are the *salsa* and *seal* benchmarks,

where there are a large amount of basic loops evenly distributed in the program such that DLICP can reduce more cache misses than NLP by prefetching. With combined DLICP and NLP, we can, however, improve the cache miss reduction by an average of 32.4%.

Performance Improvement

Since prefetching does not affect the processor instruction set architecture and organization, all designs can have the same clock cycle time. NLP executes an equal number of instructions for a given application as the baseline design. DLICP and enhanced DLICP, however, need extra special instructions inserted in the program at compile time and thus the number of executed instructions are different from the baseline design. The system performance of the three designs can be, therefore, compared in terms of total execution clock cycles.

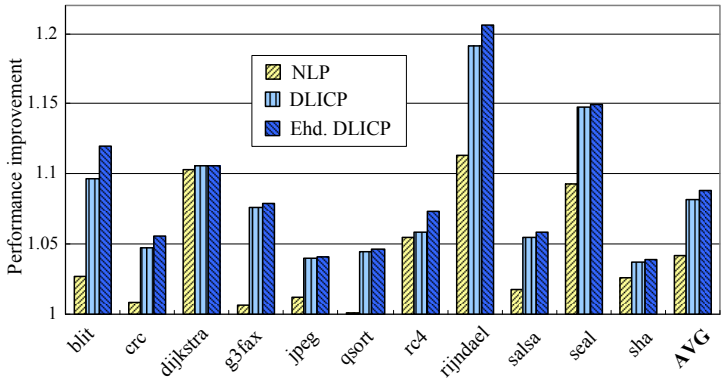

Figure 4.4: Performance improvement

Fig. 4.4 shows the program total execution cycles normalized to the baseline design without instruction prefetching and the cache miss penalty is 32 clock cycles. As can be seen, both DLICPs present better results than NLP for all applications; For some applications, such as *blit*, *crc*, and *rijndael*, the enhanced DLICP achieves remarkably higher performance improvement than DLICP. Up to 21% performance can be improved by the enhanced DLICP, as compared to the maximal 11% improvement from NLP. On average, the performance improvements of NLP, DLICP and enhanced DLICP are 4.2%, 8.2% and 8.9%, respectively. The enhanced DPLCP is two times better than NLP in terms of performance improvement. The performance improvement is largely due to the savings of *false cache misses* which are significant in NLP. *False cache misses* suffer a longer processor stall than *cache hits*.

4.3 Implementation Costs and Energy Overhead Reduction

To evaluate the area costs and power consumption of prefetching, we have modeled the NLP, DLICP and enhanced DLICP designs in VHDL and each design are estimated using Synopsys Design Compiler. For the cache and memory, we first obtain their area costs and the energy consumption per access from CACTI 5 [24], based on which we then estimate the total energy overhead of the prefetching. Both the Synopsys Design Compiler and CACTI simulations are based on the 65nm technology.

Table 4.2: Area Cost and Energy Consumption

	Area [μm^2]	Energy/Access [pJ]
NLP	1070	0.54
DLICP	3561	1.31
Eh'd. DLICP	3932	1.61
DLIC (32 x 192bits)	118958	20.41
PTT (32 x 40bits)	18915	3.62
I-cache (2KB)	184255	42.66
I-memory (2MB)	3652402	1871.44

Table 4.2 lists the simulation results. Rows 2 to 4 show the area cost of each prefeching logic and their energy consumption per memory access. The costs of the decoded instruction cache (DLIC) used by the two DLIC prefetching schemes are given in row 5; followed by the costs of the Prefetching target table (PTT) that is used by Enhanced DLICP. It is worth to mention that because most applications have loops with less than 32 instructions and each number of decoded control signals for each instruction is less than 192 in our instruction architecture, we therefore set the decoded cache size as 32×192 bits. The last two rows (Rows 7&8) present the area and energy consumption of cache and memory per memory access measured from CACTI.

As can be seen from the table the NLP scheme shows small overheads as compared to our two DLICP approaches, in terms of area cost and energy consumption per memory access. However, energy per access of the off-chip instruction memory is much higher than that of the on-chip I-cache, 50x times higher for a 2M memory over the 2KB cache, which results in the savings on the overall energy overhead.

Energy Overhead Reduction

Some instructions prefetched may never be used, namely never accessed by the processor before being flushed from the cache. Such useless prefetches do not aid performance improvement rather than waste valuable energy.

Table 4.3 shows our measurements of the total prefetches and useless prefetches

when executing each application. As can be seen from the table, both DLICP schemes demonstrate low useless prefetches (336 and 350, respectively) as compared to the 4199 found in NLP.

Table 4.3: Useless Prefetches

	NLP		DLICP		Enh. DLICP	
	#pref.	**useless**	**#pref.**	**useless**	**#pref.**	**useless**
blit	34	9	12	2	34	6
crc	33	9	14	2	37	9
dijkstra	12440	5106	3108	1179	3130	1184
g3fax	914	756	162	118	215	135
jpeg	52192	38198	2339	1002	2414	1024
qsort	55	27	33	12	57	17
rc4	53	8	47	18	66	30
rijndael	600	180	564	228	627	263
salsa	280	114	415	234	454	250
seal	1278	648	1288	503	1301	507
sha	2163	1129	1012	396	1060	426
AVG	6367.5	**4198.5**	817.6	**335.8**	854.1	**350.1**

Fig. 4.5 gives the percentages of useless prefetches over the total prefetches number, which shows the DLICP is most effective – of all prefetches, 39.5% are useless and the rest contribute to the cache hits (data accessed from cache instead of memory), hence performance improvement and energy reduction.

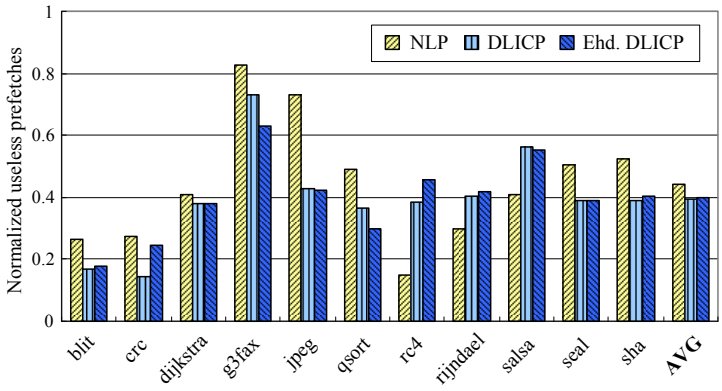

Figure 4.5: Normalized useless prefetches.

To calculate the energy overhead of the three prefetching schemes, we use the run-time profile of the I-cache, main memory, and the prefetching logic activities (number of accesses, number of hits/misses, number of useless prefetches, etc.) collected during simulation, together with the energy per access values as given in

Table 4.2.

Fig. 4.6 shows the results when using the main memories of different sizes ranging from 64K to 4M bytes, where the energy overheads (displayed in lines) are normalized to the energy consumption of main memory access (in columns) of the baseline design without the prefetching function.

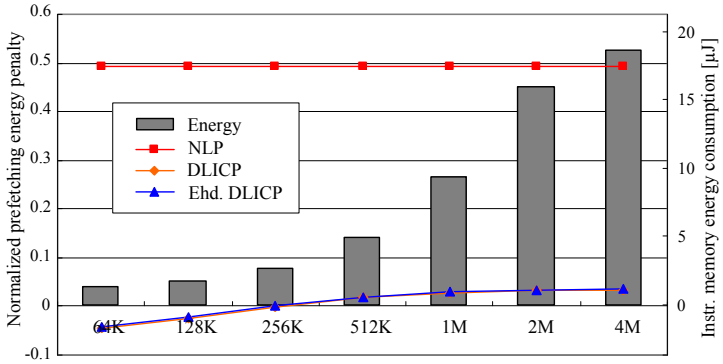

Figure 4.6: Energy penalties of the three schemes normalized to the energy consumption of the instruction memory of the baseline design.

It can be seen from the plots, NLP consumes much higher energy than the other two schemes, consistently about 49% energy consumption over all different memory configurations. This is because such energy overhead is decided largely by the useless prefetching, where the number of useless prefetchings of NLP is about 49% of the I-cache miss in the baseline design. On the other hand, the energy overheads of DLICP and enhanced DLICP are under 3.5%. When the main memory size is reduced to below 256KB, even energy savings can be observed. For example, a saving of 4.5% can be achieved when the main memory is as small as 64KB. This is because the energy overhead of DLICP prefetching can be canceled out by the energy savings due to fetching decoded loop instructions from the energy-efficient DLIC instead from the energy-expensive I-cache during execution.

5 Conclusions

Our experiment results show that even Next Line Prefetching (NLP), an existing low cost prefetching scheme, incurs a high energy overhead (around 49% of memory energy consumption), which is impractical for energy-aware embedded and ubiquitous systems.

In this paper, we presented an energy efficient instruction prefetching design for ubiquitous embedded systems with two-level memory hierarchy (on-chip cache and off-chip memory). We exploit the decoded loop cache and maximally parallelize the instruction prefetching with the decoded loop execution to reduce instruction cache misses while at a low energy overhead. The decoded loop cached based

prefetching (DLICP) can be enhanced with the NLP approach for applications, where limited loops available for the decoded loop cache.

Our experiments show that both DLICP schemes outperform NLP with improved performance and much less energy overhead, an average of 3.5% extra energy consumption as compared to 49% extra energy consumed by NLP. For some applications, the enhanced DLICP scheme offers a markedly better performance than DLICP. Up to 21% performance can be improved by the enhanced DLCIP, as compared to the 11% performance improvement by NLP.

Bibliography

[1] A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, no. 12, pp. 7–21, December, 1978.

[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach, 3rd Edition*. Elsevier Science Pte Ltd, 2003.

[3] 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 Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 5, no. 4, pp. 417–424, December, 1997.

[4] J. E. Smith and W.-C. Hsu, "Prefetching in supercomputer instruction caches," *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pp. 588–597, 1992.

[5] E. H. Gornish, E. D. Granston, and A. V. Veidenbaum, "Compiler-directed data prefetching in multiprocessors with memory hierarchies," *Proceedings of the 4th International Conference on Supercomputing*, pp. 354–368, 1990.

[6] C.-K. Luk and T. C. Mowry, "Cooperative prefetching: compiler and hardware support for effective instruction prefetching in modern processors," *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pp. 182–194, 1998.

[7] A. Cristal, O. Santana, F. Cazorla, M. Galluzzi, T. Ramirez, M. Pericas, and M. Valero, "Kilo-instruction processors: Overcoming the memory wall," *IEEE Micro*, vol. 25, no. 3, pp. 48–57, May-June, 2005.

[8] F. Dahlgren, M. Dubois, and P. Stenstroem, "Sequential hardware prefetching in shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 7, pp. 733–746, July, 1995.

[9] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 364–373, 1990.

[10] J. Pierce and T. Mudge, "Wrong-path instruction prefetching," *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 165–175, 1996.

[11] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," *ACM SIGMICRO Newsletter*, vol. 23, no. 1-2, pp. 102–110, 1992.

[12] S. Kim and A. V. Veidenbaum, "Stride-directed prefetching for secondary caches," *Proceedings of the 1997 International Conference on Parallel Processing*, pp. 314–321, 1997.

[13] M. J. Charney and T. R. Puzak, "Prefetching and memory system behavior of the spec95 benchmark suite," *IBM Journal of Research and Development*, vol. 41, no. 3, pp. 265–286, May, 1997.

[14] D. Joseph and D. Grunwald, "Prefetching using markov predictors," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 121–133, February, 1999.

[15] Z. Hu, M. Martonosi, and S. Kaxiras, "Tcp: Tag correlating prefetchers," *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, pp. 317–326, 2003.

[16] G. Reinman, B. Calder, and T. Austin, "Fetch directed instruction prefetching," *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 16–27, 1999.

[17] V. Srinivasan, E. S. Davidson, G. S. Tyson, M. J. Charney, and T. R. Puzak, "Branch history guided instruction prefetching," *Proceedings of the 7th International Conference on High Performance Computer Architecture*, pp. 291–300, 2001.

[18] Y. Zhang, S. Haga, and R. Barua, "Execution history guided instruction prefetching," *Proceedings of the 16th International Conference on Supercomputing*, pp. 199–208, 2002.

[19] J. Villarreal, R. Lysecky, S. Cotterell, and F. Vahid, "A study on the loop behavior of embedded programs," *Technical Report UCR-CSE-01-03, University of California, Riverside, 2002*.

[20] D. Burger and T. Austin, the Simplescalar Tool Set, Version 2.0. tech. report CS-TR-1997-1342, Dept. of Computer Science, Univ. of Wisconsin, Madison, 1997.

[21] J. Scott, L. H. Lee, J. Arends, and B. Moyer, "Designing the low-power m-core architecture," in *International Sympsium on Computer Architecture Power Driven Microarchitecture Workshop*, 1998, pp. 145–150.

[22] 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," in *In IEEE 4th Annual Workshop on Workload Characterization*, 2001, pp. 83–94.

[23] M. Itoh, S. Higaki, Y. Takeuchi, A. Kitajima, M. Imai, J. Sato, and A. Shiomi, "Peas-iii: An asip design environment," in *Proceedings of the 2000 IEEE International Conference on Computer Design*, 2000, pp. 430 – 436.

[24] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "Cacti: An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model," *Technical Report HPL-2008-20, HP Laboratories, 2008*.