IPULOC — Exploring Dynamic Program Locality with Instruction Processing Unit for Filling Memory Gap

Huang ZhenChun (huang@itrc.cs.tsinghua.edu.cn) & Li SanLi (isl-dcs@tsinghua.edu.cn)
Department of Computer Science & Engineering, Tsinghua University, Beijing 100084, China.
FAX: 86-10-62771138

Abstract

Memory Gap has become an essential factor influencing on achieving peak performance of high-speed CPU based systems. To fill this gap, enlarging cache capacity has been a traditional method based on static program locality principle. However, the order of instructions stored in I-Cache before being sent to Data Processing Unit (DPU) is a kind of useful information that hasn’t ever been utilized before. So we propose an architecture containing an Instruction Processing Unit (IPU) in parallel with the ordinary DPU, the IPU can prefetch, analyze and preprocess a large amount of instructions otherwise laying in the I-Cache untouched. It would be more efficient than the conventional prefetch buffer that can only store several instructions for previewing. By IPU, Load Instructions can be preprocessed while the DPU is executing on data simultaneously. We term it as “Instruction Processing Unit with Lookahead Cache”(shortly for IPULOC) in which the ideas of dynamic program locality is presented. This paper describes the principle of IPULOC and illustrates the quantitative parameters for evaluation. Tools for simulating the IPULOC were developed. Simulation result shows that it can improve the program locality during program execution, and hence can improve the cache-hit ratio correspondingly without further enlarging the on-chip cache that occupies a large portion of chip area.

Keywords: Memory Gap, IPULOC, Instruction Processing Unit.

§ 1. Introduction

In recent decades, CPU performance has been increased in thousand times (1); meantime, the main memory speed is only increased by 11% per year (2). The speed difference between the CPU and the main memory in a system is increasingly enlarged, it results in difficulty of achieving the peak performance of a modern CPU based system. This issue constitutes an essential problem titled “Memory Gap” for architecture researches. For filling the memory gap, cache traditionally plays an important role due to the feature of program locality. In order to raise the cache hit ratio, the modern processor design traditionally employs the method of increasing the cache capacity. However, the program locality that the traditional method utilizes is static, furthermore, it exists considerable limitation. The simulation results of A.J. Smith (3) show, when the cache capacity increases to certain extent, its contribution to improving the cache hit ratio would be obviously decreased. Kai Hwang (4) concludes this simulation result into a formula:

\[ \text{Hit Ratio} = 1 - C^{-0.5}. \]

Where C is the cache capacity.

Besides, in those programs the locality of instruction address sequence for memory references appears very unsatisfactory, such as database retrieval with dimitiate search and hash search classical algorithm, there is a huge amount of random data accesses, the large capacity cache is still unable to contribute to high cache hit ratio. Patterson (5) and etc. in their simulation experiment with Digital Alpha 21164 illustrates this conclusion. (In the database applications, for 1000 instructions in average, it happens 311 times cache miss, the processor utilization ratio can only reach 23%).

Moreover, in the modern processor design, the cache with increasing capacity usually occupies a very large portion of total transistors integrated on chip. For instances, in the Digital Alpha 21164 chip, the cache area takes 37.4% of the total chip area, and the transistors for building the cache occupies 77.4% of the total transistors on chip; in Pentium Pro processors, the cache area holds 64.2% and the transistors occupy 87.5% (6). The other side effect of increasing

* This paper is supported by National Natural Science Foundation.
cache capacity induces the increase of processor cost. So, it is not the optimized way to improve the cache hit ratio and to fill the memory gap only by enlarging the cache capacity, especially when on-chip cache reaches to certain extent.

This paper presents a new architecture that improves the program locality during program execution, actually it employs the idea of dynamic program locality. This paper proposed an Instruction Processing Unit (IPU) executing concurrently in parallel with the Data Processing Unit (DPU). Because of the new architecture for taking advantage of dynamic program locality is based on IPU and prefetch the load instructions in I-Cache much “earlier” than the other cache prefetch structures, we called it “Instruction Processing Unit with LOokahead Cache”, shortly for IPULOC.

§2. Instruction Processing Unit

From the modern processors with superscalar, superpipeline, and superscalar-superpipeline architecture, we can find that the designers have focused with long time efforts on the parallelism of data processing with multiple instructions issuing and multiple function units. However, the order of instructions “laying” in the Instruction Cache (I-Cache) is actually a kind of useful information that hasn’t ever been fully utilized. Although the Instruction Prefetch Buffer (IPB) and Long Instruction Window (LIW) have been employed in the DPU of modern processor architecture, however, it only makes use of several prefetched instructions for solving limited data and branch dependencies. It scarcely takes effect on improving the instruction stream locality, meanwhile in the processors with IPB and LIW, it still exists the problem of wasting information of instruction order “laying” in the I-Cache.

This paper explores the issue of increasing the average speed of memory hierarchy system by exploiting the parallelism between a so-called Instruction Processing Unit (IPU) and the Data Processing Unit (DPU). The IPU undertakes the task of analyzing and processing on a large amount of instructions otherwise laying in the I-Cache untouched. The stream processing of instruction stream in IPU is executed in parallel with the DPU processing.

![Diagram of Instruction Processing Unit (IPU) and Data Processing Unit (DPU)](image)

Fig.1. The block-diagram of the IPU linked with the DPU in a processor

The block-diagram of the IPU linked with the DPU in a processor is shown in Fig.1. The DPU contains a Multiple Instruction Issue Window and several Execution Pipelines. The IPU contains a Prefetched Instruction Stream (PIS) and a number of corresponding instruction decoders. The instruction streams will be preprocessed on three aspects, branch dependency processing, data dependency processing and the Load/Store instruction analysis. Consequently, the IPU also includes the Reorder Buffer and its management unit. After these preprocessing procedures, the related instruction streams will be sent to the Multiple Instruction Issue Window in the DPU for dispatching to the multiple function units for execution. The IPU can remove many branch dependencies, data dependencies and memory reference dependencies as well. However in order to illustrate the memory gap problem solution with IPULOC, this paper will only discuss the instruction analysis in the following sections, for the sake of simplicity.
§3. Related Work

It is known, for increasing the cache hit ratio and improving the cache performance, a lot of research work has been done. Cache prefetch [3] is a well known method, which prediction method is to employ the main memory reference address of the recently used memory instruction access to predict the currently coming instruction addresses accessing to main memory. Hardware cache prefetching is specifically concerned with prefetching algorithms implemented solely by dedicated hardware and without software support. Prefetching algorithms have been concerned with two issues: which block to prefetch and when to prefetch.

One Simple prefetch algorithm is called always prefetch [3]. With this algorithm, every time there is a reference to block i, the cache is examined form block i+1 (i.e., the next sequential block, in terms of ascending memory addresses). If block i+1 is absent from the cache, it is prefetched. A variation which requires fewer prefetches and prefetch lookups (i.e., look into the cache to see if the block is there) is called prefetch on miss [3], which prefetches the next sequential cache block if and only if the access to the current cache block is a miss. A more complicated scheme, known as tagged prefetch [3], keeps the number of prefetch lookups low while issuing more prefetches than prefetch on misses. In this case, each cache block has a single bit, called the tag, which is cleared to zero whenever the block does not reside in the cache. When a block is referenced by the processor, its tag will be set to one. A block brought into the cache by a prefetch, however, retains its tag of zero. Whenever a tag changed from zero to one, a prefetch is initiated for the next sequential cache block. This is similar to always prefetching, but it avoids repeated cache lookups and does not again prefetch a line that was prefetched and later replaced without having been referenced.

More sophisticated schemes like threaded prefetching [6] and bi-directional prefetching [7] have also been proposed. In threaded prefetching, cache block i has associated with it a list of pointers known as threads. Each thread points to a cache block, which is most likely to be referenced after block i has just been accessed. Suppose that the processor is accessing cache block i in cycle T. If block j is referenced in cycle T+1, a new thread which contains the address of block j will be attached to block i in cycle T+1. The new thread can be stored together with block i in the instruction or data cache, or it can be stored in a separate cache. As soon as block i is re-accessed by the processor, all threads associated with block i will trigger the prefetching of block j and other cache blocks the threads point to. Bi-directional prefetching attempts to capture the forward and backward accessing behavior found in data caches. If the current and the previous sequential cache blocks are found in the cache, the next sequential cache block will be prefetched; otherwise, if the current and the next sequential cache blocks are found, the previous sequential cache block will be prefetched.

Conceptually, these schemes make sense because instructions are fetched sequentially (except for branches) and data is often referenced sequentially (when in arrays) or, at least, locally (when the compiler allocates related variables in contiguous locations, as when they are stored together in a stack); i.e., Cache Prefetch still depends on the “static” address locality of memory reference instruction stream in Data Processing Unit (DPU). The effect of this method remains constrained by the better program locality for ensuring the improvement of cache performance; it is unable to solve the decreased cache performance problem in execution of the programs with worse program locality. Furthermore, The simulation of John Tse and Alan Jay Smith [8] shows that Cache Prefetch will not increase the Cache performance unless Cache, Bus, and the Memory supply some requirements difficult to realize. The number of instructions that could be utilized for prediction in a prefetch buffer is very restricted in comparison with the Instruction Processing Unit (IPU) proposed in this paper, besides, the scheduling of instructions and the prediction of the next memory access address occupy the DPU processing time. For comparison, the instruction processing in IPU mentioned above is being carried out in parallel with the data processing in DPU. Therefore, the IPULOC architecture presents a more effective measure in improving cache performance.

§4. IPULOC: the Principle, the Structure and its Implementation

The central idea of IPULOC is to build a considerable large Instruction Reservation Area in
the Instruction Processing Unit (IPU), to prefetch a large amount of instructions before those Load instructions to be executed and the data to be read out from the main memory. The IPU carries out the speculative prediction analysis for these Load instructions, and predict the main memory addresses for the data going to be read, then read the memory block containing that addresses into cache. Therefore, while the Load instructions are being executed and the addresses are being accessed, these data already had been read into cache in advance, and there are no needs to access the main memory. Thus, it can increase the cache-hit ratio, and can improve the performance of memory system as a whole. The prefetch procedure is similar to the conventional instruction prefetch, however, the difference lies on that the former is to prefetch the required instructions, and the later is to prefetch the data that Load instructions need to access after the instruction processing.

The algorithm of instruction processing in IPU is shown in Fig. 2.

In this algorithm, Inst is an instruction, Addresses is a set of addresses. The role of GetNextInstruction (NextN) function is to fetch the NextNth instruction that is going to be executed. And the role of GetPrefetchAddress (Inst) function is to take speculative prediction of the instruction returned by GetNextInstruction function, meanwhile, to return the address that instruction will possibly access. In order to increase the possibility of prefetch hit, that the GetPrefetchAddress function returns is a set of addresses that the Inst could access. The role of AddPreload is to send the addresses that GetPrefetchAddress returned into the Cache Prefetch Queue, waiting for the bus empty for prefetching data.

To realize this algorithm, a few of hardware parts are added into IPU. The block diagram of IPULOC is shown following (Fig.3):

In Fig 3, all parts marked with bold lines are the supplementary parts added to the conventional processor. These parts accomplish the cache prefetch task; its procedure is illustrated as follows:

1. By the controlling of Cache Prefetch Algorithm, those instructions that will be executed later are fetched from Prefetched Instruction Stream according to the Cache Lookahead Pointer, and decoded.

2. If these instructions are Load instructions or its equivalent instructions, then the Cache Prefetch Address Generation Part will create the prefetch address complying with the current machine status according to the GetPrefetchAddress function.
3. To decide whether the prefetch addresses are located inside the cache, if the cache doesn’t contain these addresses, then put the prefetch addresses into Cache Prefetch Queue.

4. By the decision of Cache Coordination part, it takes out the address in the Cache Prefetch Queue when the data bus is idle; it reads the cache line content indicated by that address to complete the Cache Prefetch.

GetPrefetchAddress function plays an important role in IPULOC algorithm, its accuracy of predicting the memory reference address may significantly influence the success ratio of cache prefetch. There are three kinds of addressing modes in CPU memory references: (1) Direct addressing mode without interrelation to CPU register content; (2) Indexed addressing mode and base addressing mode interrelated with CPU register content; and (3) Indirect addressing mode required memory accessing twice.

The simplest algorithm of GetPrefetchAddress function is: for direct addressing mode it has no interrelation to CPU registers, it will send the content of address field in Inst as a prediction address. For Index and Base addressing modes related to CPU register content, it will calculate the address based on the current CPU registers, then send the calculated address back as a prediction address. If it encounters indirect addressing mode that currently is rarely used, then it treats this kind of addressing mode as the direct addressing mode processed twice separately.

Because in these direct addressing modes that are related to CPU registers, the corresponding register content could be changed after the cache prefetch, then the cache prefetch being conducted ahead is possible to incorrectly prefetch the other cache lines. To solve this problem, it is better to modify or reorganize the compiler for avoiding the appearance of the instruction stream shown in Fig. 4(a) at the output.

During the compiling procedure, it can employ the method of instruction rearrangement to eliminate the instruction sequence shown in Fig. 4(a). The rearrangement method is to insert a series of instructions without dependencies between the instructions of modifying the register content and the load instructions, it can make that the register content to be prefetched is the modified correct content, shown in Fig. 4(b).

Although the instruction rearrangement can solve the problem of incorrect address prediction during processing these instructions with register-dependent addressing modes, but it requires the modification of compiler to support instruction rearrangement and it needs the modified compiler to compile the original program. In many cases, such as in lack of source code, it is unable to carry out recompilation; then for the sake of increasing the success ratio of cache prefetch, it requires to improve the GetPrefetchAddress function for instruction sequence similar to that of Fig. 4(a). The analysis of many programs has shown: the instructions of register-dependent addressing modes (e.g. index addressing mode) generally are used in array or structure, under this situation the memory references are underway from the low address end to the high address end successively, the corresponding instruction sequence is shown in Fig. 4(c) (here, c represent a constant, usually it lies in the range between 1 -16).

GetPrefetchAddress function can be improved for these instruction sequences: to take the currently calculated address and the address corresponding to the result to return back, since the register increment value rarely surpasses 16, i.e. rarely surpasses the next cache line (usually a cache line at least contains 128 bits). Therefore it can solve the cache prefetch failure problem for these instruction sequence mentioned above.

Furthermore, in the IPULOC algorithm, there is a parameter playing important role in the effect of IPULOC, it is the amount of lookaheaded instructions NextN of GetNextInstruction function. Its effect on the result of IPULOC will be given by the simulation experiment described later.
§5. The Software Simulation of IPULOC

In order to verify the advantage of IPULOC, to verify the effect of two parameters (NextN and PreloadBufLength) on the IPULOC, we use IBM SP2 supercomputer to conduct a large amount of software simulation, the simulation tool has been developed on the basis OPRMAS (Object-Oriented Parallel Reconfigurable Micro-Architecture Simulation) [9]. This simulation tool can carry out the analysis for the program run on IBM SP2, and obtain its instruction stream, then submit them to the Cache simulator for simulation. For ensuring the simulation result to be more closing to the practical situation, we choose several general purpose benchmarking programs and testing the programs of classical algorithms. They are:

1. c4: a little game,
2. fft; 1-D FFT translate,
3. flops; test program for float point performance,
4. heapsort; sort of heap,
5. sim; find the most similar part between two sequences,
6. test1; diminiate search,
7. test2; hash search.

Cache simulator assumes that it needs a CPU clock for reading data from cache; and it needs 5 CPU clocks for reading one cache line into cache, i.e. cache latency is 5. And one cache line contains 128 bits (16 Bytes), the cache capacity is 16 Bytes * 256 lines = 4KB.

For evaluating the effect of IPULOC architecture from the load instruction view, we define a speed-up factor named $S_{IPULOC}$:

$$S_{IPULOC} = \frac{R_{Load(normal)}}{R_{Load(IPULOC)}}$$

The result of simulation analysis and the discussion of cache parameters are given in the following:

(1). The Effect of NextN on Speedup Factor

The selection of NextN is important for raising $S_{IPULOC}$, if the NextN is selected too small, then the time remained for prefetch would be too short, $S_{IPULOC}$ cannot be increased significantly; if the NextN is too large, then the wrong address probability of the GetPrefetchAddress prediction would be increased. Fig. 5 shows the simulation result with NextN = 2,3,4,5,7,8,16,32

From Fig. 5, we can find that the optimum $S_{IPULOC}$ is reached at NextN = 3, 4 or 5, it means that taking NextN slightly less than or equal to cache latency is a better solution.

For the sake of excluding the other factors of interference (such as the basic block size) and confirming the essential effect of cache latency on NextN selection, we conducted the same simulation with cache latency=20, the result is shown in Fig. 6.

Fig. 6 shows that for the system with cache latency = 20, the speedup factor $S_{IPULOC}$ maximum value NextN=14-15, slightly less than cache latency; meanwhile, when NextN = cache latency (20), speedup factor still maintain at a high value. This experiment proves the conclusion of that NextN selection is mainly decided by cache latency rather than by other factors of interference such as basic block size.
(2). The Effect of PreloadBufLength on Speedup factor

The effect of PreloadBufLength on the speedup factor principally is to limit the length of Cache Prefetch Queue. As a general consideration, the larger the PreloadBufLength is, the better factor S_{IPULOC} will be obtained.

The Table 2 lists several minimum PreloadBufLength values for ensuring every prefetch abling to enter into the prefetch queue under running several benchmarking programs.

The Table 2 also gives that the largest value of the tested PreloadBufLength is only 7. So, we might set up the PreloadBufLength value as 16 for ensuring the good performance. The current VLSI technology is easy to implement a Cache Prefetch queue contains 16 items.

<table>
<thead>
<tr>
<th>Program</th>
<th>NextN</th>
<th>c4</th>
<th>fit</th>
<th>flops</th>
<th>heapsort</th>
<th>sim</th>
<th>test1</th>
<th>test2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Several minimum PreloadBufLength values

(3). Analysis on Implementation of GetPrefetchAddress

In the last part of §4, we present and discuss an improved algorithm for GetPrefetchAddress function, the experimental data in Fig. 7 and Fig. 8 shows the effect of this algorithm.

From Fig. 7 we can find, when NextN = 4, the unimproved GetPrefetchAddress function provides a better result (with the only exception in C4 program); while NextN = 5, the improved GetPrefetchAddress function gives a better result (the exception is test2 program). This is because of the improved GetPrefetchAddress function requiring double clocks to invoke two cache lines into cache. Besides, from the Fig. 7 it is seen that the improvement of GetPrefetchAddress function plays a negative role, it decreases the speedup factor S_{IPULOC} in test2 program. This is owing to that test2 program is a hash search, it rarely appears the instruction sequence shown in Fig 2(c). Under this situation, the improved GetPrefetchAddress can only increase the P_{err}, consequently it obtains a worse S_{IPULOC} in comparison with the unimproved GetPrefetchAddress function.

The above analysis is conducted on the premise of that invoking each line requires the same clocks, if the memory system can support special transfer modes, such as burst transfer mode, it would allow the shortened time required for invoking two successive cache lines, so the improved GetPrefetchAddress would offer greater contribution to increasing of speedup factor S_{IPULOC}.

Fig. 8 shows the experimental result with the assumption of memory system supporting burst transfer mode, where the invoking of two successive cache lines requires time cache latency +1 = 6, rather than 2 * cache latency = 10.

From Fig. 8 it is observed that if memory system supports the burst transfer mode or the similar modes, then the improved GetPrefetchAddress can better increase the S_{IPULOC}. The average
speedup factor lies in 1.22 to 1.23, i.e. to increase the memory performance about 20%, that usually requires enlarging the cache capacity (with million transistors) to achieve it.

(4). The Effect of Cache capacity on Speedup factor

For the further study of IPULOC effect, we had conducted simulation with 5 different cache capacities, i.e. 2KB, 4KB, 8KB, 16KB and 64KB. The simulation result is shown in Fig 9.

From Fig. 9, it is shown that when the cache size is in the range of 2KB to 64KB, the speedup factor $S_{IPULOC}$ with IPULOC architecture can be obtained from 1.12 to 1.25. Correspondingly, the CPU utilization efficiency can be increased to 5% to 10%. So the conclusion is that IPULOC architecture can increase the average speed of memory system under different cache capacity sizes, it can help in filling the memory gap and increase the processor utilization efficiency with the cost of limited hardware increase, that is rather small in comparison with the cost of further enlarging the cache transistors and cache area on chip.

§6. Conclusion

Load instructions in RISC and its counterpart load operations in CISC constitute a significant factor, which possibly could cause latency in accessing main memory. Instruction Processing Unit exploits the useful information of instruction order otherwise laying in the I-Cache untouched; by utilizing this information, IPU can reorganize the instruction order. Experimental simulation shows, it is a very powerful measure for increasing the processor performance. In our IPULOC structure, IPU can improve the dynamic program locality during execution, and then can considerably improve the cache hit ratio via the improved dynamic program locality in comparison with the traditional measure of increasing the on-chip cache capacity, generally adopted in the current modern processor design. Consequently, it can help in filling the memory gap and hence can improve the system performance. Beside the function of filling memory gap, IPU can work together with Virtual Register, proposed in our another paper [10, 11] to solve the data dependency and branch dependency.

Reference: