

# Loop Optimization with Decoded Cache

Thiruvél Thirumoolan (t\_thiruvél@yahoo.com)

School of Computer Science and Engineering

CEG, Anna University, Chennai, India

## **Abstract:**

*In this paper a small hardware specifically designed for handling loops either as indicated by the compiler or detected at run-time is presented. This design offers high performance for statically scheduled loops and almost the same performance for dynamically detected loops. The dynamic adaptation is done by detecting the code as loop, based on specific conditions. This also takes into account loops of known and unknown exit conditions. The ILP present in loops is exploited and this benefit is also extended to the code following the loop.*

## **1. Introduction:**

Loops are the code segments where most of the time is spent for any program. It is that part which has to be exploited more for ILP and also for faster execution. Traditionally two approaches are used to exploit ILP- static and dynamic scheduling. In static scheduling, compiler will identify the dependences and schedule the instructions such that the performance is maximized. In dynamic scheduling, the hardware has a view of some instructions and issues instructions based on the dependences it detects and resolves them at run-time.

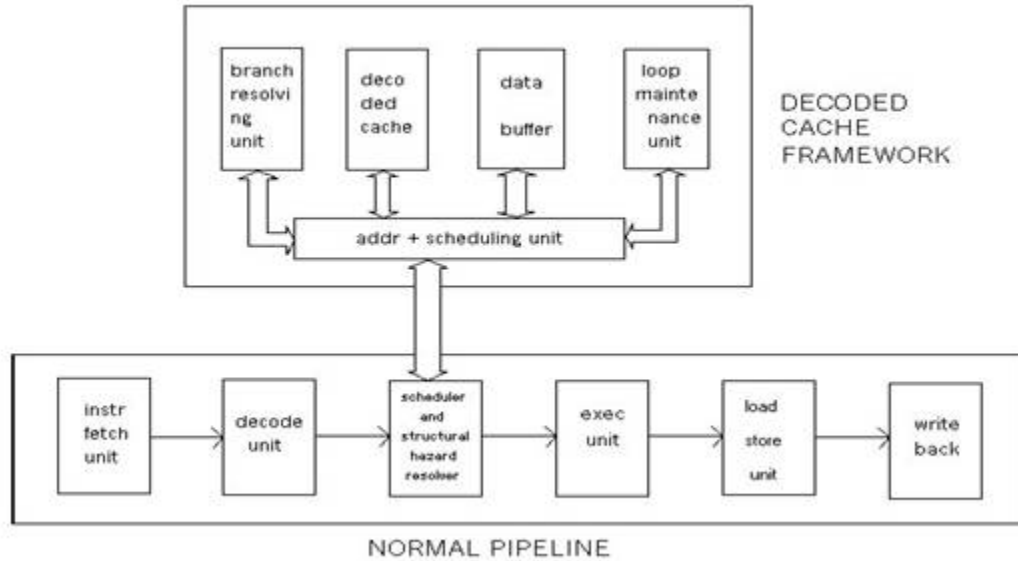
Some known techniques which are used in optimizing loops are loop unrolling [1], loop fusion [2], run time parallelization etc. However these have the disadvantages like assumption of the compiler on the latency of the instructions, unnecessary unrolling of loops, increased code size and cache miss, repeated fetching and scheduling of the same instructions.

In this paper a new technique for loop optimization using decoded cache framework that is a hybrid of both static and dynamic techniques is presented. This technique uses a hardware that is placed after the decode unit. In this technique, the compiler has to specify to the hardware that the subsequent portion of the code will form a loop and the exit conditions of the loop through some special instructions . The exit conditions can be known at compile time or at run-time. The registers in the hardware are appropriately loaded with corresponding values which are necessary to control the loop. In the first iteration as the compiler has indicated, the code sequence for a length ( indicated by the compiler by loading the value into a register-LOOP\_SIZE ) is decoded and the decoded signals are sent into the execution unit and also stored in the decoded cache after proper scheduling. The loop overhead instructions are not stored in the decoded cache. From the second iteration the decoded signals are sent into the pipeline from the decoded cache along with the data present in the data registers. During the execution of the loop the fetch and decode units are free and hence the code sequence that follows which is independent on the data produced by the loop can be issued after resolving the structural hazards with the decoded signals in the cache. The loop overhead is completely removed and implemented in the hardware. This is the main advantage of this technique. In nested loops the most interior loop will be executed many times and hence the most interior loop is decoded and stored in the decoded cache. The performance of loops is enhanced in this technique because of the hardware detecting parallelism in loops, decreased code size, decreased cache access and misses and single scheduling of loop instructions.

This paper is organized as follows. Section 2 consists of the proposed framework for the decoded cache which discusses the overall design. Section 3 discusses the implementation strategies for different hardware components in the framework. Section 4 discusses the performance estimate of the hardware.

## 2. Decoded cache framework:

The decoded cache framework consists of the following units as shown in Figure 1.



**Figure 1 Overall Processor Architecture**

*Decoded cache* – for storing the decoded instructions of the loop

*Loop maintenance unit* – for maintaining the loop indices and checking for loop exit condition

*Branch resolve unit* – this takes care of branches inside the loop

*Addressing & Scheduling unit* – this takes care of exceptions and schedules instructions and

*Data buffer*- this holds the data to be used by the loop.

The loop maintenance unit will vary the index of the loop and will check the exit condition of the loop by the hardware each iteration. The branch resolve unit will check for the branches in the loop and issue the decoded signals accordingly. So the loop overhead is eliminated. The addressing and scheduling unit will issue the scheduled decoded signals in the way they are organized in the decoded cache.

### 3. Implementation strategies:

The registers used by the different units of the framework are presented in Table 1.

Register name	Functions	User access	Instructions
LOOP_SIZE (Addressing & scheduling unit)	This is loaded with the size of the loop without the loop overhead statements.	Available.	MOV
VALUE (Loop maintenance unit)	This indicates the value by which the loop counter has to be added or subtracted.	Available. Default: 01	MOV
FINAL (Loop maintenance unit)	This indicates the bound of the loop.	Available.	MOV
COUNTER (Loop maintenance unit)	This is the index register	Available. Default:00	MOV
CTR_ENABLE (Flip Flop) (Loop maintenance unit)	This enables the COUNTER register.	Available.	DYN
PCSTART (Addressing & scheduling unit)	This has the starting address of the loop. This is used in exception handling.	No.	-
INDEX (Addressing & scheduling unit)	This is used to address into the decoded cache.	Available.	MOV

**Table 1 Registers used by different units.**

DYN: This is an instruction with no operands which indicates to the hardware that the loop exit condition is dynamic. This is discussed in detail in section 3.2.2.

The implementation strategies for the different units are presented below.

**3.1 Decoded cache:** The decoded cache is present inside the processor. The size of the cache determines the maximum size of the loop it can hold. However the word length of the Decoded cache will be large. Larger loops can be implemented with Virtual memory. The loop code needs to be scheduled only once and from the next iteration the loops are scheduled based on structural hazards detected for the other instructions which follow from the decode unit and then issued. The decoded signals are stored in the cache based on the parallelism detected and the organization of the cache. The design of the cache is also important. A known technique called banked cache can be used. In this technique, the cache is organized into banks [3]. Each bank contains independent instructions. The instructions at a particular index in the banks represent independent instructions that can be issued in parallel. However the disadvantage of this technique is that if in any case, only one instruction could be issued at any time then the corresponding entries in other banks would be empty. This results in wastage of space in a cache which is costly. An alternative method is suggested here. In this technique called *Bit technique* each instruction in the cache has an extra bit which is the MSB. Those instructions which can be issued in parallel are represent by the MSB as either '1' or '0'. The next sequence of parallel instructions will be represented by their corresponding MSB as '1' or '0' respectively. These bits are set when the loop body is first decoded and stored in the cache.

**3.2 Loop maintenance unit:** This unit has certain registers as indicated in Table 1 which are used in maintaining the loop indices. The situation is handled differently in the case of known and unknown bounded loops. However the loop exit condition is the same in both cases, the COUNTER being reduced to ZERO.

*3.2.1 Statically known loops:* The registers used for loop maintenance are loaded by the instructions as indicated by the compiler. When these registers are loaded the details about the loop is indicated to the framework. The dedicated adder does the loop overhead calculation (varying of the index). This adder can also be configured as subtractor depending on the values loaded into the registers. If the final value of the loop's index is more than the initial value it is configured as adder and vice versa. The loop exit condition is checked by the hardware by comparing the value in the COUNTER register with the FINAL register.

*3.2.2 Statically unknown loops:* Here as the index of the loop is not varied uniformly the technique used for statically bound loops cannot be used here. This situation is indicated to the hardware by the DYN instruction. The hardware loads the value 1 into the COUNTER and disables the COUNTER. The loop exit condition is checked every iteration. If the condition fails, then the COUNTER is enabled. The COUNTER will now reduce to ZERO and the loop will be exited.

**3.3 Branch resolving unit:** The branches are detected and are resolved in this unit and the calculated index value is loaded into the index register of the addressing unit. This unit only generates the index value. As the instructions are already decoded, the branch target namely the index of the target decoded signal is present at the end of the decoded word if it is a branch. Branches may be of two types.

*3.3.1 Internal branches:* The branches in the loop can target to an instruction within the loop. Once the branch is resolved the INDEX register is loaded with the target value if the branch is taken.

*3.3.2 External branches:* The index value generated is checked for its occurrence inside the loop based on the loop size and current INDEX value. The INDEX is added with the starting physical address of the loop and the PCSTART of the instruction fetch unit is loaded with the calculated value.

**3.4 Addressing and scheduler unit:** The decoded signals in the decoded cache are addressed by the INDEX which indicates its address in the cache [4]. The INDEX is varied in accordance with the organization of the cache. In a banked cache it is incremented by 1 until the LOOP\_SIZE is reached. In the proposed technique, this is varied so that each cycle a parallel chunk of code is issued.

**3.5 Data buffer:** The data buffer holds the values of the arrays used by the loops. The data are fetched into this buffer and are directly fed into the execution units along with the decoded signals. The computed data is forwarded to other instructions ( which follow the loop in the program ) which are scheduled from the fetch and decode units. This ensures that the same data used by two instructions in adjacent parts of the program need not fetch it two times.

#### **4. Performance estimate:**

Consider the following piece of code which is frequent in many programs. An analysis of this code is done and the performance is estimated.

```
for ( int i=0;i<100;i++)
{
    a[i]=b[i]*c[i];
    d[i]=b[i]+c[i];
}
for ( int j=0;j<100;j++)
    sum=a[j]+sum;
```

The loop body has to be fetched and decoded and the same data has to be fetched every iteration. Assume for a typical architecture the latency to fetch the instructions for a single loop is 5 cycles and 5 cycles for data, add and multiplication units have a latency of 1 and 3 cycles respectively and the start up overhead is 5 cycles. Then for a typical processor with two Integer add units, one multiplication unit and one load store unit, the performance is measured and the total clock cycles are calculated.

For the first loop no of cycles = start up overhead + for each iteration (Load latency + add and multiplication in parallel + 2 cycles for loop maintenance + storage latency) =  $(5+100*(10+3+2))$   
**= 1505 cycles.**

For the second loop no of cycles = start up overhead once + for each iteration (Load latency + add + 2 cycles for loop maintenance)+ one store latency =  $(5+ 100 * (10+5+1+ 2)+5)=$ **1810 cycles**

Total no. of cycles for the code = **3315 cycles.**

For the presented architecture with the same no of functional units and same latencies,

No of cycles for first loop= Inst. Load + start up over head + for each instruction ( data load + add & multiplication in parallel + store latency)=  $(10 + 5 + 100 (5 + 3)) =$  **815 cycles.**

Here the no of cycles is less because the loop instructions have to be loaded only once and the data for each loop has to be loaded every time and hence the memory port will be more freely available for fetching. Also the loop overhead of two cycles for each iteration is removed i.e. 200 cycles.

No of cycles for the second loop=for each loop (load latency + add + loop maintenance 2 )+ store latency =  $100*(5+1+2)+5=$ **805 cycles.**

In the second loop the case is similar to the case in the previous architecture. However as already discussed both these loops can be chained so that the array “a[ ]” need not be fetched in the second loop. When chaining is performed, the second loop can be issued after the first loop has calculated the first value in the array. This happens after 23 cycles. So the other part of the first loop can be chained with the second loop.

Total no of cycles for the code in the presented architecture  $23 + 805 = 828$  cycles.

Speed up in the presented architecture = 4 (approx)

The architecture has high performance with the introduction of decoded cache only [5],[6]. With the further reduction of loop overhead and further scheduling of the subsequent code and improved data locality, high performance can be obtained.

## **5 Conclusion:**

The main improvement has been done to optimize the loops which form the main part of most programs. A generic hardware design has been proposed to handle it more effectively. The decoded cache technique has been used and loops are decoded and scheduled only once. The other units in the proposed hardware have been used to control the loop and also to resolve branches in the loop. The proposed design enhances the performance by reducing the loop overhead completely by using decoded cache framework. A theoretical analysis is presented for a loop that is frequently used in many programs. However an extended analysis needs to be carried out in Simple Scalar suite.

## References:

1. John L Hennessey and David A Patterson, "Computer Architecture A quantitative approach", Second edition, Harcourt Asia PTE ltd Chapter 4 Section 4.1 pages 220-240.
2. Naraig Manjikian and Tarek S Abdelrahman "Fusion of loops for parallelism and locality" *IEEE transactions on Parallel and Distributed Systems* Vol 8, No 2, pages 193-209 February 1992.
3. Kenneth M Wilson and Kunlen OluKotun, "High bandwidth on-chip cache design" *IEEE transactions in Computers* Vol 50, No 4, pages 292-307 April 2001.
4. Kernal Ebcioglu, Erik Altman, Michael Gschwind and Sumedh Sathaye, "Dynamic binary translation and optimization" *IEEE transactions on Computers* Vol 50, No 6, pages 529-544 June 2001 .
5. Mark Smotherman and Manoj Franklin "Improving CISC Instruction Decoding Performance Using a Fill Unit" *28<sup>th</sup> Annual International Symposium on Micro architecture* pages 219-229 Year 1995.
6. G. Intrader and I Spillinger, "Performance Evaluation of a Decoded Instruction Cache for Variable Instruction-Length Computers", *Proc. of the 19th Symp. Comp. Arch.*, IEEE Computer Society, May 1992.