Optimization Of ST231 Processor

Prepared By

Prerna Maheshwari

08MCE006

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NIRMA UNIVERSITY

AHMEDABAD-382481

MAY 2010

Optimization of ST231 Processor

Major Project

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology in Computer Science and Engineering

Prepared By

Prerna Maheshwari

08MCE006

Guided By

Sumit Sharma

STMicroelectronics

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NIRMA UNIVERSITY

AHMEDABAD-382481

MAY 2010

Declaration

This is to certify that

i) The thesis comprises my original work towards the degree of Master of Technology in Computer Science and Engineering at Nirma University and has not been submitted elsewhere for a degree.

ii) Due acknowledgement has been made in the text to all other material used.

Prerna Maheshwari

Certificate

This is to certify that the Major Project entitled **”Optimization Of ST231 Processor”** submitted by **Prerna Maheshwari (08MCE006)**, towards the partial fulfillment of the requirements for the degree of Master of Technology in Computer Science and Engineering of Nirma University of Science and Technology, Ahmedabad is the record of work carried out by his under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of my knowledge, haven’t been submitted to any other university or institution for award of any degree or diploma.

Sumit Sharma	Prof. D. J. Patel
Guide,	HOD, Professor,
STMicroelectronics,	Institute of Technology,
HED(Home Entertainment Devision),	
Greater Noida	Nirma University, Ahmedabad

Dr. S.N.Pradhan
Professor, PG Coordinator,
Department of Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad

Abstract

ST delivers its own embedded processor ST231 core and already has presence across the globe which is widely used as an audio, video or graphics accelerator within Home Entertainment Application(audio, video, set top boxes, digital TVs) as well as lajer jet and ink jet printer. ST keeps on developing the new version of ST231 processor with the new added features as the customer requirements. The ST231 is the latest processor of the ST2xx in the market of embedded VLIW computing. It is a integer 32bits VLIW processor, 3 stages pipelined, which contains 4 integers units, 2 multiplications units and 1 load/store unit. It has a 64KB L1 cache. The latency of the L1 cache is 3 cycles. The cache is blocking, i.e. in the case of load cache-miss, the pipeline stalls until the commit of the pending load. This was degrading the performance in terms of memory latency for some applications which uses this ST231.The cache is separated Data/ Instruction. The Data cache is 4 way associative. It operate with write back no allocate policy. A 128 bytes write buffer is associated with the Dcache. In this project the aim is to optimization of ST231 by resolving this blocking cache problem by implementing non-blocking cache in Dcache side and Icache optimization by machine learning appraoach.Genetic programning is able to reduce the L2 Icache misses by cache content selection. A non-blocking cache can improves processor performance by adding the overlapping of cache misses to the ability of overlapping execution with cache misses provided by limited blocking.

Acknowledgements

I am deeply indebted to my thesis guide **Sumit Sharma** for his constant guidance and motivation. He has devoted significant amount of his valuable time to plan and discuss the thesis work. Without his experience and insights, it would have been very difficult to do quality work.

I would also like to extend my gratitude to **Dr.S.N.Pradhan**, Coordinator of M-Tech Computer Science Department, Nirma University for his continuous encouragement and motivation.

I am thankful to **Pravat Kishor Nayak and Geeta Joshi** my Team Member for his valuable support in my thesis.

I am also thankful to **colleagues of my class** for their delightful company which kept me in good humor throughout the year.

Last, but not the least, no words are enough to acknowledge constant support of **my family members** because of whom I am able to complete the degree program successfully.

- Prerna Maheshwari
08MCE006

Contents

List of Figures

List of Tables

Abbreviation

MSHR Miss Information/Status Holding Registers

DTLB Data translation lookaside buffer

SPQ Size of pending load queue size

BINOPT Binary optimization tool

GA Genetic Algorithm

BSP Board support package

WCET Worst case execution time

CRTA Cache-aware Response Time Analysis

Chapter 1

Introduction

1.1 Optimization of ST231 Processor

The 32-bit ST231 is a member of the ST200 family of cores.This family of embedded processors uses a scalable technology that allows variation in instruction issue width, the number and capabilities of functional units and register files, and the instruction set.

The ST200 family includes the following features:

- parallel execution units, including multiple integer ALUs and multipliers
- architectural support for data prefetch
- predicated execution through select operations
- efficient branch architecture with multiple condition registers
- encoding of immediate operands up to 32 bits
- support for user and supervisor modes and memory protection

1.1.1 ST231 overview:

The ST231 includes the ST231 core and associated peripherals. fig.1.1 shows the arrangement of these components in a block diagram.

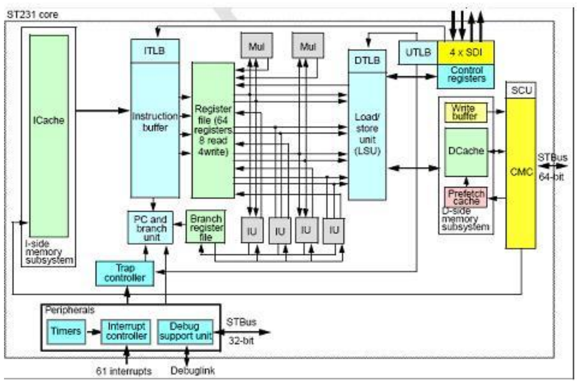


Figure 1.1: Architecture of ST231 Processor

1.1.2 Objective

ST delivers its own embedded processor ST231 core and already has presence across the globe which is widely used as an audio, video or graphics accelerator within Home Entertainment Application(audio, video, set top boxes, digital TVs) as well as Lajer jet and Ink jet printer.

The scope of this project is to optimization of ST231 processor verification cycle The key features to be targeted are,

- implementing a Non-Blocking Cache in ST231 rather than blocking cache.
- Reducing the memory latency by using prefetch cache.
- Adjust the load latencies in order to improve the gain of non-blocking cache.

Chapter 2

Literature Survey

2.1 Influence of cache effects on ST231 processor

It has been measured the impacts of different cache architecture and also impact of the compiler. and by following a practical approach with common benchmark(mediabench) and a less common application (ffmpeg). This is a typical embedded multimedia application used by STmicroelectronics to design their chip. It a video compression basing h263 standard which are precisely simulated. The used simulator model an embedded processor which is ST231 core. As the ST231 is the latest processor of the ST2xx in the market of embedded VLIW computing. It is a integer 32bits VLIW processor, 3 stages pipelined, which contains 4 integers units, 2 multiplications units and 1 load/store unit. It has a 64KB L1 cache. The latency of the L1 cache is 3 cycles. The cache is blocking, i.e. in the case of load cache-miss, the pipeline stalls until the commit of the pending load. The cache is separated Data Instruction. The Data cache is 4 way associative. It operate with write back no allocate policy. 128 bytes write buffer is associated with the Dcache. So this blocking cache is degrading the performance in terms of speed and time complexity. So we are targeting to reduce memory latency which causes the reduction in speed. As it demands for the application is being used.

After collecting the simulation results of the benchmark on the ST231 using two cache scheme Blocking cache and Non-Blocking cache.

Blocking cache which in ST231 processor somewhat degrading the performance in terms of increasing the memory latency. As with blocking cache when the data cache miss occurrs the ST231 processor stalls completely.

As the gap between the processor cycle time and memory latency increases, the cache miss penalty become more severe and that result in lower processor utilization.

Several enhancements to cache design have been proposed to reduce the miss penalty:

- Multilevel cache hierarchies which lower the average memory access time in a cost effective way.
- Hit ratio can be improved by complementing caches with small buffers or specialized caching structures.
- Fast context switching can hide the memory latency of a process.

Rather to exploit these schemes we decided to use another better approach namely how to exploit the overlap of ST231 processor computation with data access within one process by using Non-Blocking cache and prefetching cache these features does not provided by above mentioned schemes.

Usually, a processor must stall on a cache miss until the miss is resolved. In the case of write misses, this can be avoided by the use of a write buffer. The basic idea in non-blocking and prefetching caches is to hide the latency of (read and write) data misses by the overlap of data accesses and computations to the extent allowed by the data dependencies and consistency requirements. A non- Blocking (or lockup-free) cache allows execution to proceed concurrently with cache misses until an instruction that actually needs a value to be returned is reached. Such caches exploit the overlap of memory access time with post-miss computations. Hardware and/or software Prefetching can eliminate the miss penalty by generating memory requests to bring the data into the cache before its actual use. These techniques exploit the overlap of computations prior to a cache miss.

2.1.1 Non-Blocking Caches

In order to allow non-blocking operations and multiple misses, it is necessary to implement Miss Information/Status Holding Registers (MSHRS) that are used to record the information pertaining to the outstanding requests. Each MSHR entry includes the data block address, the cache line for the block, the word in the block which caused the miss, and the function unit or register to which the data is to be routed. Subsequently, the view is that non-blocking loads are features specified in the ST231 processor, non-blocking writes are supported by buffering writes, whereas whether the cache allows multiple pending accesses or not depends not only on the presence of MSHRS, but also on the available cache bandwidth as defined by the interface between caches and memory modules. a non-blocking cache will be a cache supporting non-blocking reads and non-blocking writes, and possibly servicing multiple requests. Non-blocking loads require extra support in the execution unit of the ST231 processor in addition to the MSHRS associated with a non-blocking cache. If static instruction scheduling in pipelines is used in the ST231 processor, some form of register interlock (like a full/empty bit for each register) is needed for preserving correct data dependencies. Under dynamic instruction scheduling, introducing out-of-order execution, some scoreboarding mechanism is required. Both scheduling strategies need interrupt handling routines that can deal with interrupts generated by the non-blocking operations.

2.2 Performance issues

As we analyzed by the survey, the non-blocking operations exploit the post-miss overlap of computation and memory access while prefetching exploits the pre-miss overlap. We give now a brief qualitative view of the expected benefits for both types of overlap. Non-blocking loads delay processor stalls until the necessary data dependence is encountered. They will become necessary for ST231 processors capable of issuing multiple instructions per cycle. However, the non-blocking ,in which the number of instructions that can be overlapped with the memory access, is likely to be small in the case of static scheduling. It can be increased when compilers produce code optimized for this potential overlap. A larger non-blocking distance can be obtained with dynamic scheduling and out-of-order execution. By comparison, non-blocking writes can be more advantageous in reducing the write miss penalty because the nonblocking distance is usually equal to the memory access time. Moreover, the write buffer, a FIFO queue lbuffering pending writes, does not need a supporting unit in the processor. On the other hand, the write miss penalty may not be a large fraction of the total data access penalty, even without a write buffer.

2.2.1 Architectural Model

The ST231 is the processor of the ST2xx in the market of embedded VLIW computing. It is a integer 32bits VLIW processor, 3 stages pipelined, which contains 4 integers units, 2 multiplications units and 1 load/store unit. It has a 64KB L1 cache. The latency of the L1 cache is 3 cycles. The cache is blocking, i.e. in the case of load cache-miss, the pipeline stalls until the commit of the pending load. The cache is separated Data/ Instruction. The Data cache is 4 way associative. It operate with write back no allocate policy. A 128 bytes write buffer is associated with the Dcache. The next formula describes the execution time of a VLIW code on an ST231 in function of different stalls sources resulted from dynamic hardware mechanisms:

$$\mathbf{T = Calc + DC + IC + + Br}$$

- T : is the total execution time in processor clock cycles,
- Calc : is the effective computation time in cycles,

- DC : is the number of stall cycles due to Dcache misses,
- IC : is the number of stall cycles due to instruction cache misses,
- InterS is the number of stall cycles due to the interlock mechanism and finally
- Br : is the number of taken branch (for each branch, there is one penalty cycle).

STmicroelectronics provided us a precise pipeline accurate simulator of the ST231. we wanted improve performance of application benchmarks using full precise, but long, simulation. next section presents our performance analysis of ST231 using a regular blocking cache

2.3 Blocking Cache Architecture Results

For a coarse grain profiling we use a simulator named ST200run with the simulator option -a statistics. It prints precise and detailed execution statistics. simulation results has been collected of the mediabench and ffmpeg execution. It can observe in Fig. .2.1 that a mean of 3.5% of time is lost in stalls due to Dcache misses. We focus on pegwit and jpeg benchmarks while Dcache miss represent 96.91% and 15.66% resp. Fig. .2.2 shows that 33.34% of execution time is wasted in Dcache stalls.

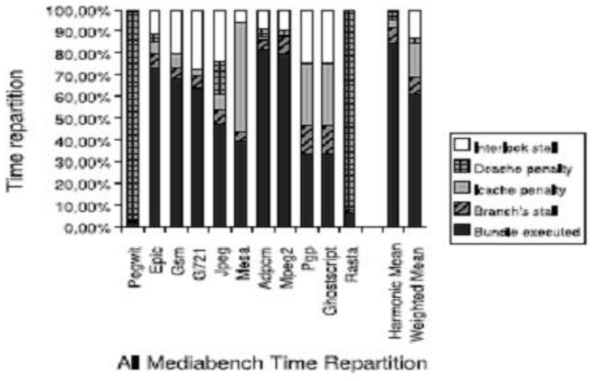


Figure 2.1: Blocking Cache result

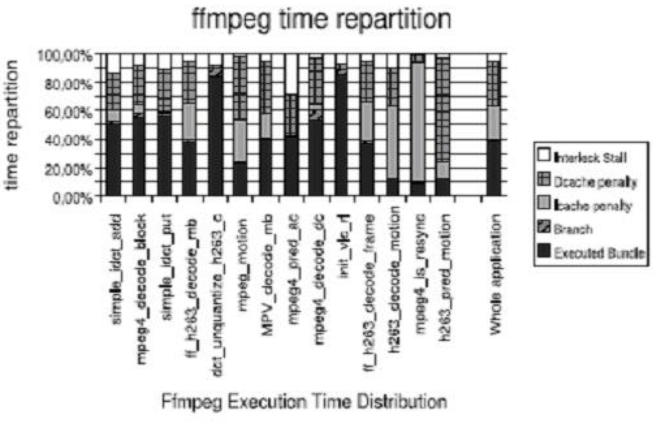


Figure 2.2: Nonblocking cache result

Chapter 3

Techniques used to optimization of ST231 Processor

There were two points arising for Dcache optimization

- How to reduce the memory latency.
- How to get rid from Blocked Stalls.

Two Answer these question the proposed schemes are:

- Using the Non-Blocking caches in St231 core rather than blocking caches.
- and by using multiported caches.

3.1 Block diagram to implement queue for Non-blocking cache

3.2 Dcache Organization

Instruction cache addressing is illustrated in Figure .3.2 The data cache is 32 Kb four way set associate and built from 4 x 256 x 32 byte lines. The virtual address bits

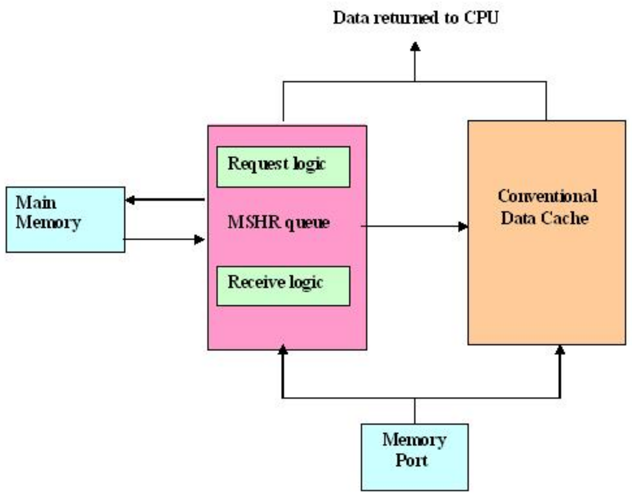


Figure 3.1: Block diagram of MSHR queue

[12:05] are used to index the data cache RAMs. Virtual address bits [31:13] are sent to the DTLB for translation. The translated physical address bits [31:13] from the DTLB are then compared against the data cache tag. Virtual address bits [04:00] are used to select the correct bytes from the cache line.

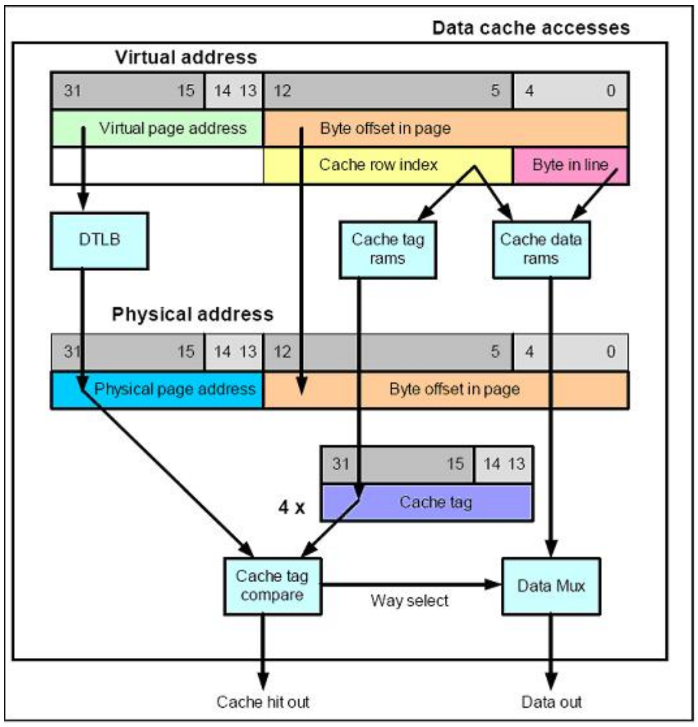


Figure 3.2: Data cache addressing

3.2.1 Algorithm for DcacheAccessCheck before implementing MSHR Queue

Read in the data to ensure we don't have cache misses

If Dcache line in byte is divided by integer value

then divide that offset by 2

elseif

Dcache line in byte is divided by integer value and also added with the divided offset value

then substract offset value from Dcache line in byte which is divided by an integer value

select the random value

write data into cache

```
read data from the cache
again read data from the cache
if data having the expected value then okay
else data not matching the expected results
put the counter to check DMiss and DHit
if counter == 2
then DMiss expected not matching
(if counter == 1)
then DHit expected not matching
add address with offset
allocate RAM for physical space
allocate 64K region for cached accesses
make the uncached region cached
map pages in
exit from DcacheAccessCheck.
```

3.3 Handling Misses in ST231

In this section, a hardware data structure capable of providing nonblocking cache access is presented. Its operation is thoroughly described, The overall design is shown in Figure. The MSHR queue contains the MSHRs which contain the information about which words have been written during the time the line has been requested from memory. In this thesis, MSHR and MSHR queue entry are used interchangeably. The MSHR queue must also contain the circuitry to request cache lines from memory and to handle their return. It must also contain the associative logic to identify when a memory request has hit in the MSHR queue.

3.3.1 MSHR queue

As stated above, the MSHR queue contains a set of MSHRs organized as a FIFO queue. Each MSHR contains buffer space to handle writes during the time the line is being accessed from memory. Each time an access occurs, the MSHR queue is associatively searched, looking for a valid MSHR entry whose line number matches the line number of the memory request. This type of match is referred to as a ”hit” in the MSHRs. Each time a write hit in the MSHRs occurs, the word is written into proper data space in the matching MSHR, and the V bit corresponding to the written word is set. When a future read request hits in the MSHR queue, the V bit for the corresponding word in the matching MSHR is checked. A V bit for an individual word in an MSHR will be referred to as Vi, as opposed to V which indicates the V bit for the MSHR itself. If the Vi bit is a one, the value in the data space is returned to the cpu. If the Vi bit is a zero, the read request is entered into the read queue and is satisfied when the line returns from memory. The action of the read queue will be discussed in the following subsection. The number of data word entries in an MSHR is equal to the number of words in the cache line. Figure shows the MSHR queue for a 4-word cache line. The MSHR queue needs to be N ported to handle the memory requests from N cache ports. The total number of comparators needed is N * M where M is the number of MSHRs.
Note that the inorder return of cache lines is assumed here. In this case, the MSHR queue can be managed as a simple FIFO. Aside from the MSHR queue itself, there exists other logic consisting mainly of pointers which controls the allocation and removal of items from the MSHR queue. This associated logic will be discussed later. The data structure for the MSHR queue is summarized below. The design assumes a cache line size of four words.

- V bit. This bit indicates that this specific MSHR entry contains valid information.

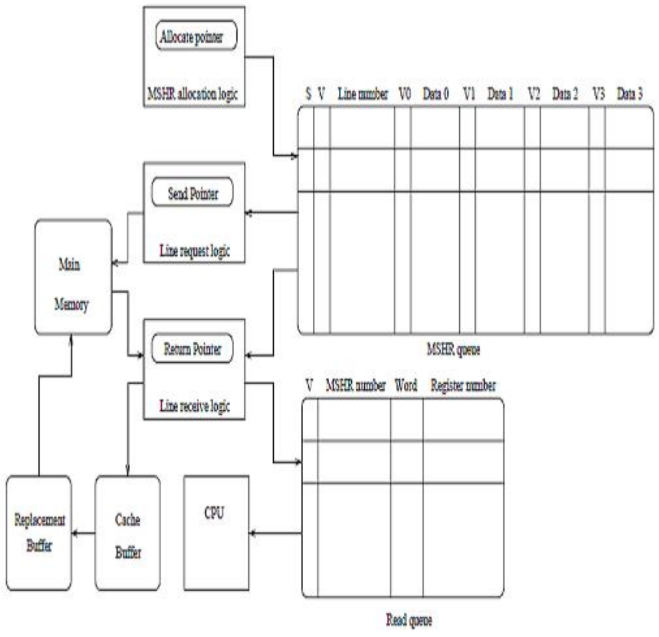


Figure 3.3: Block diagram of the data structures necessary to handle non-blocking cache access for a four-word cache line

- S bit. This bit indicates that this specific MSHR entry needs to request a line from memory.

- Line number. This is the address of the line that is being requested from main memory.

- Data 0 - Data 3. These entries are reserved as storage space to hold any writes to the cache line which occur before the line has been returned from memory.

- V0 - V3. These are the valid bits that indicate that data has been written into the corresponding word.

3.3.2 MSHR Usage

There is particular concern with the number of MSHRs used because each MSHR greatly influences system cost. Recall that in this model, each MSHR represents an additional memory request that can be overlapped in the main memory. Therefore, the phrase "adding an extra MSHR" implies increased complexity in many parts of the memory system in ST231, particularly the main memory. Note that for further improvement an MPNBC may be designed with multiple MSHRs, each contending for a main memory system that supports only a single outstanding request, but this is not considered here.

Table I: MSHR results on various benchmark

Benchmark	**MSHR0**	**MSHR1**	**MSHR2**	**MSHR3**	**MSHR4**
compress	.561	.300	.114	0.017	.007
espresso	.973	.024	.002	.000	.000
fpppp	.941	.044	.012	.002	.000
matrix300	.486	.402	.025	.030	.056
sort	.864	.125	.011	.000	.000
tbl	.959	.028	.005	.002	.006

- MSHR usage is examined both as an average over the 5 million instructions executed, and dynamically, looking at how the instantaneous MSHR usage changes with time in ST231 processor. Table shows the frequency of usage for varying numbers of MSHRs. For instance, Table shows that compress has 2 valid MSHRs 11.4% of the time.

  All four MSHRs are rarely used except by matrix300. For programs with high hit rates, 1 MSHR is used less than 5% of the time, and 2 or more MSHRs are used less than 1% of the time. For programs with poor hit rates, MSHR usage is higher. Compress, matrix300, and tomcatv use one MSHR 20% to 40% of the time. Two or more MSHRs are used 10% to 15% percent of the time. Note that these values are highly dependent on cache size and memory latency which are fixed in this experiment at 32K and 10 cycles respectively. The results obtained

for MSHR usage may vary greatly if different cache sizes and memory latencies are assumed. Next, the dynamic usage of MSHRs is considered. Figures .3.3 and .?? show the dynamic usage of the MSHRs. Each point shown is the average number of active MSHRs per cycle within a 100,000 instruction window. This gives a good indication of instantaneous MSHR usage.
The behavior of benchmarks with high hit rates is easily explained from these figures. In espresso, fpppp, and yacc there is an initial amount of MSHR usage. After the cache fills up with usable data, the number of active MSHRs per cycle drops to near zero for the remainder of the execution time. It can be seen that the programs with poor hit rates (e.g., compress, matrix300, sort, and tomcatv) are constantly using MSHRs to request data throughout the program's execution.

3.3.3 Implementation of MSHR queue for Non-Blocking Cache in ST231 processor

structure pointer t fptr: pointer to node t, count: unsigned integerg **structure node t** fvalue: data type, next: pointer tg **structure queue t** fHead: pointer t, Tail: pointer tg

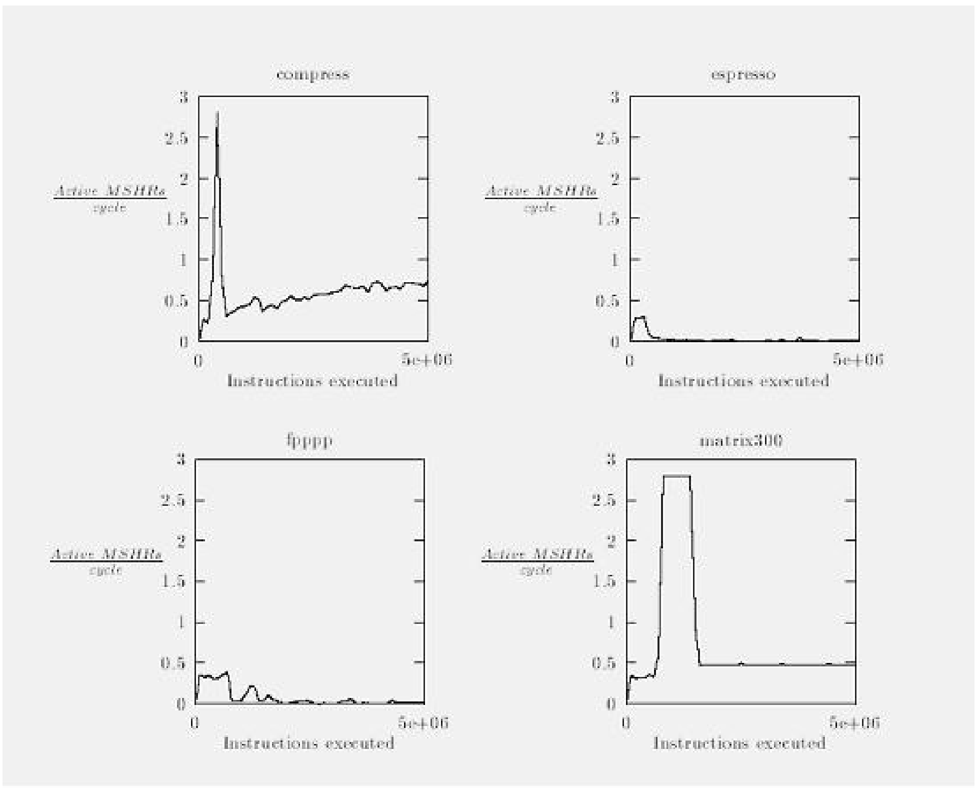
```
initialize(Q: pointer to MSHRqueue t)
node = new node() % Allocate a free node
node->next.ptr = NULL % Make it the only node in the MSHR queue
Q->MISS= Q->HIT = node % Both HIT and MISS point to it
enqueue(Q: pointer to MSHRqueue t, value: data type)
E1: node = new node() % Allocate a new addresses from the free list
E2: node->value = value % Copy enqueued value into node
E3: node->next.ptr = NULL % Set next pointer of node to NULL
E4: loop % Keep trying until Enqueue is done
```

```
E5: MISS = Q->MISSl % Read MISS.ptr and MISS.count together
E6: next = MISS.ptr->next % Read next ptr and count fields together
E7: if MISS == Q->MISS % Are MISS and next consistent?
E8: if next.ptr == NULL % Was MISS pointing to the last node?
E9: if CAS(&MISS.ptr->next, next, <node, next.count+1>) % Try to link node at the end of the MSHR queue
E10: break % Enqueue is done. Exit loop
E11: endif
E12: else % MISS was not pointing to the last node
E13: CAS(&Q->MISS, MISSl, <next.ptr, MISS.count+1>)% Try to swing CACHE MISS to the next node
E14: endif
E15: endif
E16: endloop
E17: CAS(&Q->MISS, MISS, <node, MISS.count+1>) % Enqueue is done. Try to swing CACHE MISSl to the inserted node
```

Dequeue the MSHR queue(Q: pointer to queue t, pvalue: pointer to data type): boolean

```
D1: loop % Keep trying until Dequeue is done
D2: HIT = Q->HIT % Read HIT
D3: MISS = Q->MISS % Read MISS
D4: next = HIT->next % Read HIT.ptr-¿next
D5: if HIT == Q->HIT % Are HIT, MISS, and next consistent?
D6: if HIT.ptr == MISS.ptr % Is queue empty or MISS falling behind?
D7: if next.ptr == NULL % Is MSHR queue empty?
D8: return FALSE % Queue is empty, couldn't dequeue
D9: endif
D10: CAS(&Q->MISS,MISSl, <next.ptr, MISS.count+1>) %MISS is falling behind. Try to advance it
```

```
D11: else % No need to deal with CACHE MISS Read value before CAS, otherwise another dequeue might free the next node
D12: *pvalue = next.ptr->value
D13: if CAS(&Q->HIT, HIT, <next.ptr, HIT.count+1>) % Try to swing CACHE HIT to the next node
D14: break % Dequeue is done. Exit loop
D15: endif
D16: endif
D17: endif
D18: endloop
D19: free(HIT.ptr) % It is safe now to free the old dummy node
D20: return TRUE % MSHR Queue was not empty, dequeue succeeded
```



3.4 MSHRs Versus Cache Ports

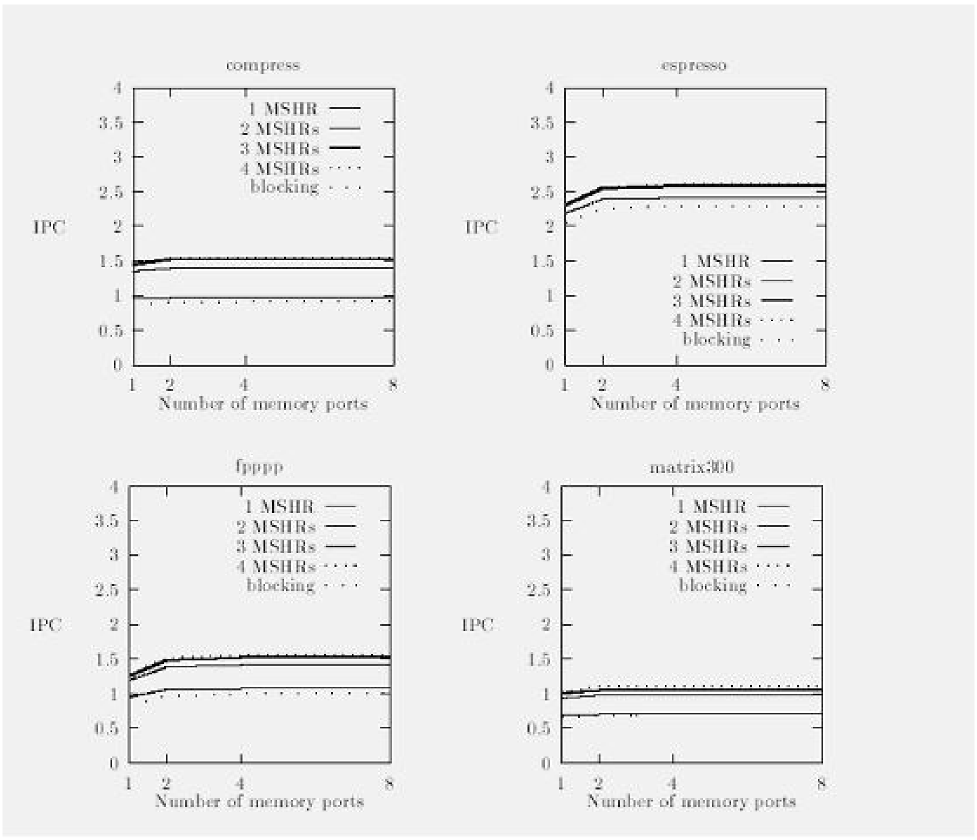
Both the MSHRs and cache ports are highly influential in determining the cost of the memory system in ST231. The purpose of this section is to see how tradeoffs between MSHRs and cache ports affect the performance of ST231 for large and small cache sizes. Simulations were performed for each benchmark with the number of ports equal to 1, 2, 4, and 8. The number of MSHRs was varied from 1 to 4. A blocking cache was also simulated. The blocking cache is similar to the case with 1 MSHR except that the blocking cache is unable to handle any other memory requests, even hits, during the time in which the MSHR is valid. For this and the next two sections, the assumed memory latency is 25 processor cycles.
Next figures show the performance of the benchmarks with a 32K cache in ST231. For the benchmarks with moderate and poor hit rates, increasing the number of MSHRs from 1 to 2 demonstrates a notable performance increase. Typically, the worse the hit rate, the more performance is gained by adding an extra MSHR. Notice, however, that the performance gain quickly drops off after a second MSHR is added. Matrix300, compress, and tbl are the only programs which show performance increase when going from 2 to 3 MSHRs, and even so, this gain is slight. The programs with very high hit rates such as espresso, yacc, and fpppp show little performance gain at all by going beyond a single MSHR.

Going from one memory port to two increases performance substantially for most programs. However, matrix300 shows little performance gain when going beyond one memory port. Perhaps this is due to the fact that the extra cycles caused by a lack of cache ports are insignificant compared to the time taken to constantly load in new data from memory. Time spent waiting for data to arrive from memory can be used to service requests that were stalled due to unavailable ports. Only tomcatv shows a performance increase when the number of memory ports is increased above two.

The blocking cache performance is similar to the performance of the one MSHR case in ST231, only slightly degraded because hits cannot be serviced during the time in which any MSHR is valid. The benchmark tomcatv encounters significant performance degradation with the blocking cache in St231. This may be due to large amounts of cache hits and MSHR hits which are on critical paths. The MPNBC case can service these while a miss is outstanding in ST231 data cache. A blocking cache can not.

Shown in following Figures are the results of the previous experiment performed with a 4K cache. For a small cache size, increasing the number MSHRs becomes much more important than increasing the number of cache ports. All benchmarks, even the ones with good hit rates, show a significant performance increase as the number of MSHRs is increased from 1 to 2. Many benchmarks experience further performance increase as the number of MSHRs is increased from 2 to 3. This can be explained by the fact that as the miss rate of the cache increases, more misses are now simultaneously occurring.

The more the misses can be overlapped, the better the overall system performance(ST231). Cache ports are not as important for programs which are constrained by a lack of MSHRs. The extra cycles spent waiting for results to return from memory can be used to issue the memory instructions which were stalled due to unavailable ports. Most of the 1 MSHR plots are relatively at with respect to the number of cache ports for most benchmarks. As more MSHRs are added, the effect of additional cache ports becomes slightly more pronounced.
The blocking cache case showed a slightly worse performance than the 1 MSHR case for all benchmarks in ST231. Unlike the case for the 32K data cache, the blocking case for tomcatv is only slightly worse than the case for 1 MSHR. This may be due to the small cache size. With a small cache, the accesses made during a miss will be misses and will not improve performance anyway.



3.5 MSHRs Versus Cache Size

From the above section, it can be seen that as the cache size is made large, the incremental value of the MSHRs becomes smaller because the hit rate has increased. In this case, the cache ports are an important factor in determining system performance for many of the benchmarks. For smaller caches, it is seen that the ability to overlap misses becomes the dominating factor and the number of MSHRs plays a more important role. However, some programs such as matrix300 make poor use of the cache, and it is expected that cache size will not affect performance.
Next figures expand on this by providing results with varying cache size and MSHRs.

The number of cache ports is fixed at four. The cache size is varied from 4K to 128K, and the number of MSHRs is varied from 1 to 4.

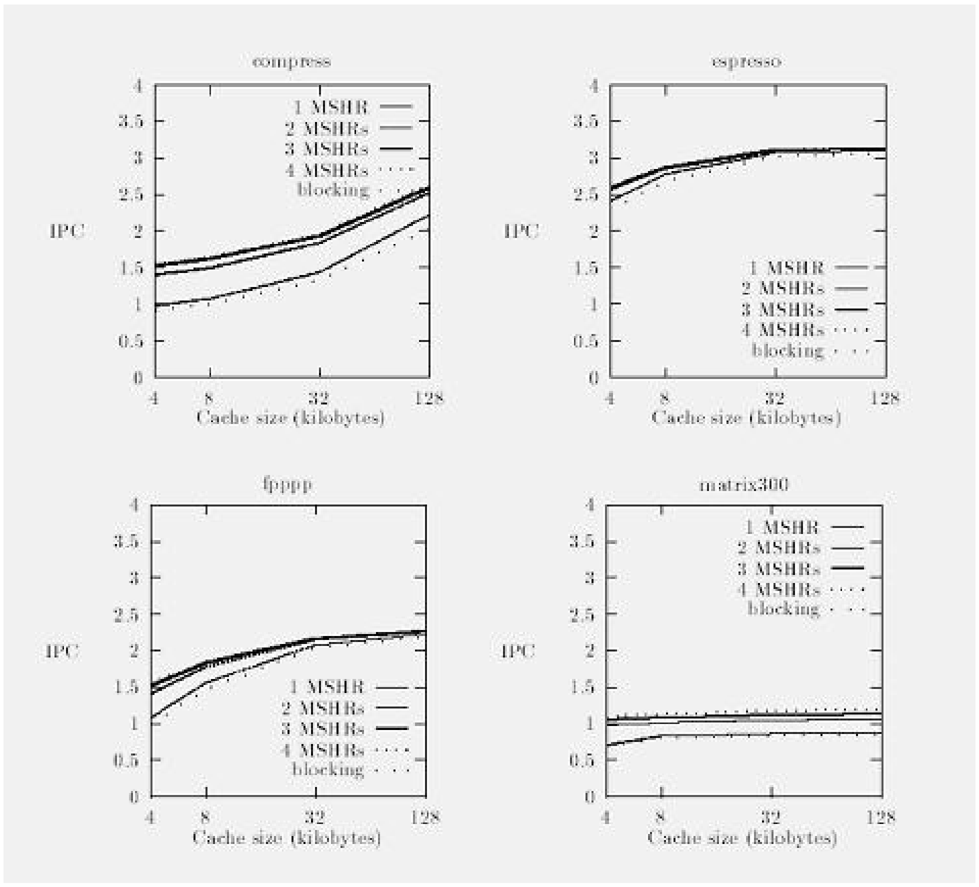


Figure 3.4: Performance with varying MSHRs and cache size for compress, espresso, fpppp, matrix300

3.6 MSHRs Versus Memory Latency

This section investigates the design of the second-level memory system in ST231. The interest is in whether it is more beneficial to allow for more overlapping in the second-level memory system (i.e., more MSHRs), or to provide for faster access. The number of MSHRs and the memory latency are varied from 1 to 4 and from 5 cycles to 30 cycles, respectively. It has been seen that, for some programs, using a cache size of 32K shows little about the second-level memory system since the cache hit rate is high. Therefore, this experiment uses a reduced cache size for the benchmarks with high hit rates. This experiment uses cache sizes of 1K and 4K for the benchmarks espresso, fpppp, tbl, and yacc. Cache sizes of 4K and 32K are used for compress matrix300, sort, and espresso.

Next figures show the simulation results for small cache sizes and Figures 4.16 and 4.17 show results for large cache sizes. It can be seen that there is definite tradeoff between extra MSHRs and decreasing memory latency in ST231. For instance, tomcatv and fpppp benefit more going from one to two MSHRs than from reducing the memory latency by 15 to 20 cycles. Additional MSHRs provide benefits as well. Other programs such as sort and yacc depend more on memory latency, though increasing the number of MSHRs does provide a notable increase in performance

For a small cache, the benefits provided by extra MSHRs in ST231 seem relatively stable with respect to memory latency. In other words, increasing the number of MSHRs in ST231 provides performance increase even when the memory latency is small. For large cache sizes, the importance of additional MSHRs in ST231 is relatively small for small memory latencies and increases as the memory latency is increased. Overlapping will always be a win when either the cache size is small or the memory latency is large.

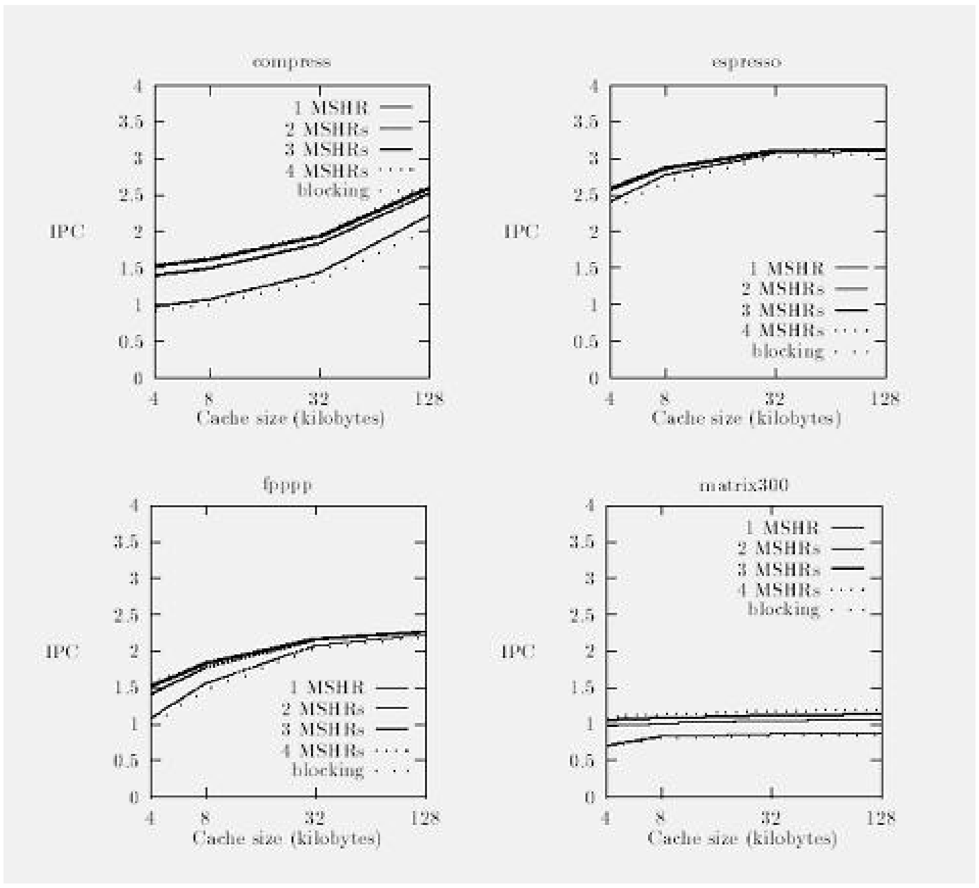


Figure 3.5: Performance with varying MSHRs and memory latency for compress, espresso, fpppp, matrix300

Chapter 4

Non-blocking Cache Simulation Results

4.1 Analysis

All the observed small speed-ups are due only to Dcache stall reduction. When considering exactly the same binary codes executing them on the same ST231 processor but with changing the blocking cache to a non blocking one seems to do not alter other dynamic performance metrics: Icache stalls, branch penalties and interlock stalls remain the same except Dcache stalls. This would improve the predictability of the execution time. The experimental results of this section can be summarized as follows:

1. A disappointing cache stall reduction when changing cache configuration from blocking to non-blocking ones. The maximum obtained performance gain is 2.62% in the application.

2. All the performance gains are calculated in the whole applications, not just in functions which make numerous Dcache misses. The performance improvement is of

course better when the amount of Dcache misses is important.

3. The codes were not changed or tuned for the new cache architecture, the same binaries were executed over the two cache platforms.

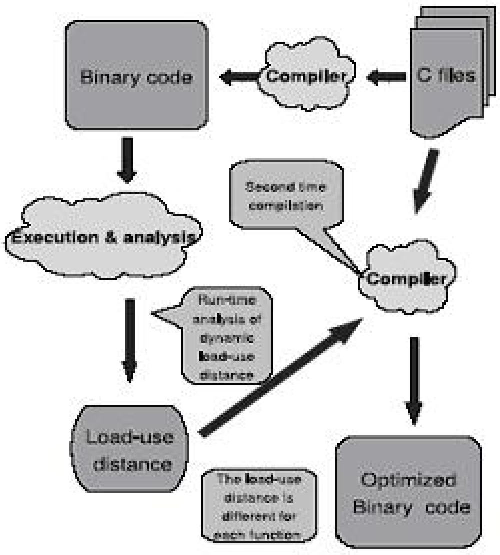


Figure 4.1: Used Methodology

In the non-blocking caches. The interesting aspect of this architecture is the ability to overlap the execution and the memory data loading. When a cache miss occurs, the processor continues its execution of independent operations. This produces an overlap between bringing up the data from memory and the execution of independent instructions. , it has been seen that a non blocking cache can significantly improve the performances of an ST231 processor. So, many high performance ST231 currently adopted this cache architecture. Embedded processors do not have non-blocking caches yet because: its cost is not negligible (energy consumption and price), and its benefit in cache of in-order processors is not demonstrated. In order to make a full exploitation of non blocking, the memory architecture should also be

improved. Indeed, memory must now become fully pipelined and ported (These architectural enhancements are not an obligation in case of blocking cache).
This improvement allows memory to serve multiple pending cache misses in a pipelined way. This makes a precise performance evaluation resulting from adding a non-blocking cache inside an processor (ST231). In the first step, we collect the same execution statistics of the blocking cache experiences i.e. the number of cycles of effective calculation, the stall cycles due to Dcache misses, the stall cycles due to instruction cache misses, the cycle lost in branch and the interlock stalls. We made distinct simulations, changing each time the size of the pending load queue size from 0 to 32 entries. A pending load queue equal to zero means that the architecture implements a blocking cache. A pending load queue with n entries means that at most n cache misses can be issued concurrently by the nonblocking cache. For a pending load size equal to zero, Figure .4.2 shows that the results are similar to the simulation results obtained with the blocking cache simulator in Figure .4.2 The performance improvement is 1.62% for the whole ffmpeg application .The result is similar in case of mediabench applications.

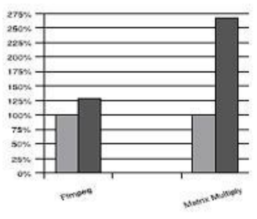


Figure 4.2: Blocking Cache in ST231Processor and Non-Blocking Cache in ST231Processor

4.1.1 Comparison

1.Blocking Cache in ST231

With the Blocking Cache, the performance was very low with respect to speed and complexity. because of dcache miss, when it occurs then If a request is made to the cache and there is a miss, the cache must wait for the memory to supply the value that was needed, and until then it is ”blocked

2.Non-Blocking Cache in ST231

With the Non-Blocking Cache the performance has improved with respect to speed and complexity, because it was able to work on other requests while waiting for memory to supply any misses

4.1.2 Results

These are the results of blocking & non blocking caches.

Table I: Result of blocking Cache

Blocking cache	**value**
Clock Cycle	5’670’1670
Real time(ms)	14,175
Num issued bundles	1’348’470
Icache hits	1’381’949
Icache misses	720
Icache miss penalty cycles	109’673
Dcache hits	207’269:
Dcache misses	257’986
Dcache miss penalty cycle	4’202’045

Nonblocking cache	**value**
Clock Cycle	3’876’123
Real time(ms)	14,175
Num issued bundles	1’348’470
Icache hits	1’381’949
Icache misses	720
Icache miss penalty cycles	109’673
Dcache hits	340’143:
Dcache misses	180’846
Dcache miss penalty cycle	3’014’023

Chapter 5

Icache optimization in ST231

As in previous part of project we have done optimization of ST231 processor for D-side memory subsystem, that means on the side of data cache we implementd the nonblocking caches to make ST2321 more efficient in the sense of clock cycle and time complexity. Which is successfully done.

The next aims is to define a methodology for optimizing I-cache on the ST200. It is most relevant to cores that have a direct mapped I-cache such as the ST231. It is not so applicable to the ST240 core that has a four-way associative I-cache.

5.1 Introduction of Icache

The I-cache architecture of the ST200 processors family is a 32-Kbyte direct mapped cache with 64-byte lines. The direct mapped cache is the simplest and cheapest type of cache architecture, but as a drawback, it results in higher cache miss rates, compared to more sophisticated cache models, like associative multi-way caches. Furthermore, the ST200 is an instruction-level parallel VLIW processor, thus it has a large instruction bandwidth requirement (inducing an increase in the cache miss rate) and also a large cache miss penalty. For these reasons, it is important to ensure that code running on the ST200 makes effective use of the instruction memory system.

5.1.1 Icache in ST231:

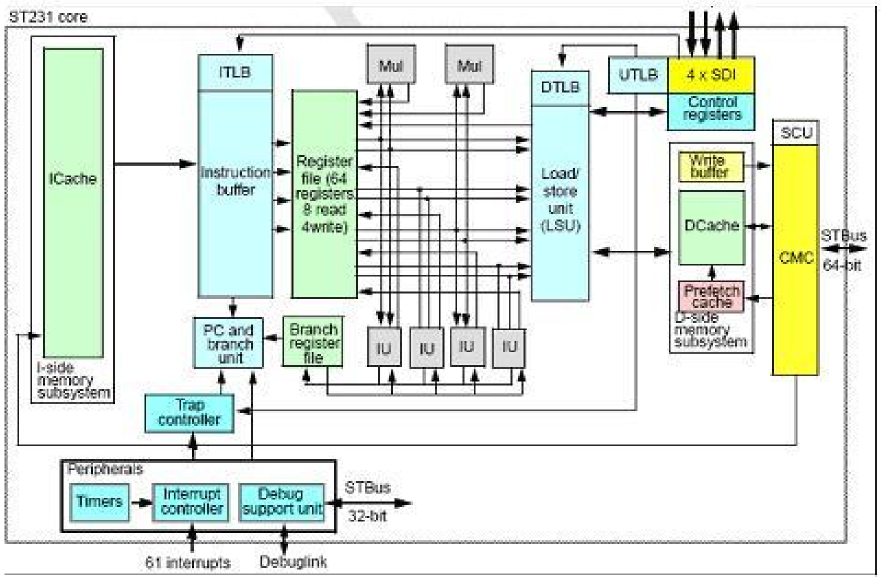


Figure 5.1: Icache in Architecture of ST231

To increase this effectiveness, STMicroelectronics has developed a binary optimization tool called for ST200 which includes an I-cache optimization tool. The I-cache optimization phase of binopt controls the layout of functions in the binary code to be executed on the ST200. This is achieved by using cache line coloring algorithms to reorder the functions in the final executable. The cache line coloring algorithms minimize I-cache conflicts by optimizing spatial locality and thus optimize I-cache usage.

5.2 Problem Defination

The problem with the cache line coloring algorithm was speed loss. Because for L1 cache it was succeed to reduce the instruction cache misses, But for L2 cache coloring algorithm causes an average increase of 10.13percent instruction cache misses. This is likely due to the fact that coloring algorithm can only concern itself with one of the two caches. As it tries to reduce L1 instruction cache misses, it may increase the L2 cache misses.

So we are proposing to avoid this kind of issues by using Genetic Algorithm for improve the performance in the sense of speed.

- The evaluation of previous work on i-cache optimization based on cache line coloring in the context of ST231 architectures. We extend the design to handle set-associativity, and show that while performance differences are generally variable and within noise, significant positive improvements, up to nearly 6% can be achieved. The coloring implementation is less successful.
- We will explore the use of genetic algorithms for i-cache optimization. With increased complexity due to evolving hardware, OS, and software designs, learning-based approaches have significant promise as a means of increasing performance. We are able to show a consistent if small improvement 0.5%, and up to 10% for specific benchmarks.

Here, a set of candidate solutions are evolved toward an optimal solution, as measured by program execution time. This design functions as a general optimization heuristic driven by actual performance, and follows current interest in machine learning approaches to optimization.

So the cache coloring algorithm we consider has only a simple and coarse model of cache design, while from learning algorithm we will have an ideal model in the sense that it adapts to the actual performance of the underlying architecture.

We are using the Genetic algorithms, for instance, in loop tiling optimization, which offering fast convergence and near optimal solutions. We examine register allocation and data prefetching in particular, and show promising, but variable results, suggesting sensitivity to genetic algorithm parameters

The aim is to define a methodology for optimizing I-cache usage on the ST231, which applies to real user application code. Firstly, the context of this application note is defined. Before implementing the genetic algorithm for this tool we are using the different I-cache optimization techniques that are stated and described. Finally, the results on a chosen test-case application will presented.

5.3 Context:

The context of this application note is the I-cache optimization of libraries. This was motivated by the fact that key applications running on the ST231, such as audio-video application code, are composed of one main driver (or system), and several modules (or packages), organized as separate individual libraries (a module is a set of source files, compiled and then grouped together in a library).

For instance, in the DVD/ACC audio codec application suite, each library contains the code for a specific audio process (coder, decoder or post-processing) and all these processes are driven by a main system. Supposing there are N libraries (lib1.a, lib2.a,...,libN.a), the classical build commands are the following:

```
build and creation of the libX.a library
for process X (X=1,2,..,N)
st200cc -c libX*.c
st200ar libX.a libX*.o
build of the main driver objects
st200cc -c main*.c
creation of the final executable
```

```
st200cc -o main.exe main*.o lib*.a
```

The result is the executable main.exe, containing the main system objects and the N libraries. Within this context, the code placement in the final executable is not controlled at all, and thus, depending on the process to be run, the I-cache miss cycles can be very high and lead to significant performance loss.

5.4 Icache miss issues diagnosis

The purpose of this section is to give us some hints on how to investigate the following issues:

- How to detect that an application is subject to I-cache miss issues,
- How to detect which are the incriminating functions

To diagnose I-cache issues on application code, the first thing to measure is the I-cache miss cycle count, compared to the total cycles. A high value of the ratio I-cache miss cycles/total cycles is a good indicator of I-cache related issues in the code. These values can be obtained either through the simulator statistics (if simulation is possible), the performance monitoring hardware block (by instrumenting the code or using the Board support Package facility).

However, these issues cannot always be completely solved by I-cache optimization techniques; because an I-cache miss can either be a conflict miss or a cold miss (occurring the first time a function is loaded in the cache). In the later case a reordering of the functions in the binary code will not be able to remove the miss. The number of these non-removable I-cache misses is proportional to the code size of the library to be executed.

In fact, the best figure to track is the number of I-cache conflicts occurring at runtime. This value gives the upper bound of the attainable performance in terms of reducing I-cache miss cycles. If the number of cycles due to conflict misses (one I-cache miss roughly costs 150 is negligible compared to the total number of cycles, there is no point in trying to optimize function placement. Conversely, if a lot of I-cache conflicts are measured, this indicates inefficient code placement which should be improved by applying I-cache optimization techniques.

The st200gprof profiling utility, included in the ST200 Micro Toolset, is a valuable tool for detecting which functions may conflict. For this purpose, the profiling mode of the simulator must be used during execution to generate the profiling I-cache information file gmon.outICACHE. This file can then be processed by st200gprof to produce a view of the I-cache miss cycles function tree, including the percentage of miss cycles for each function, which may indicate which functions are subject to conflicts (even if there is no information on conflicting function pairs.)

- **PHASE 1** Generate the trace files from the cache simulators and use the simulator analysis tool to report the occurrence of cache events. To analyze the cache behavior of the code, proceed to phase 2.

- **PHASE 2** Visualize the profile results in the cache analysis tool to identify the areas of code that are incurring cache misses. To improve the efficiency of cache, proceed to phase 3.

- **PHASE 3** Apply optimization techniques and transformations to improve cache efficiency. Use the Simulator Analysis tool to check the improvement. If the code is still not as efficient as you would like, repeat steps in phase 2 and 3 until you are satisfied

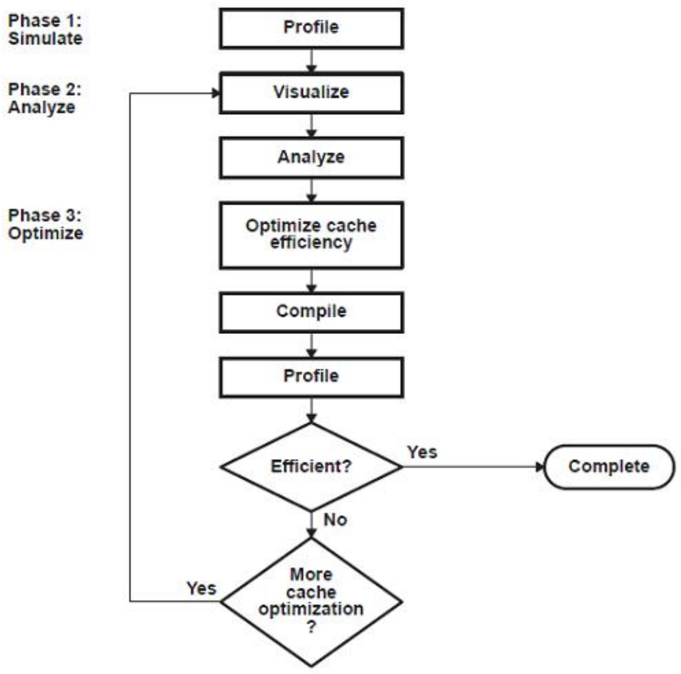


Figure 5.2: Development Flow to Increase Cache Efficiency

5.5 I-cache optimization technique:

- Relocatable linking
- Static I-cache optimization
- Dynamic I-cache optimization
- gprof file driven icache optimization method

5.5.1 Relocatable linking

Before modifying the ST200 binopt tool(I-cache optimization) with the new algorithm, a first attempt at code placement can be achieved by performing a relocatable link of the object files of each library, using the st200cc -r option, before creating the library. Here is the modified build command sequence:

build and creation of the libX.a library
for process X (X=1,2,..,N)
st200cc -c libX*.c
st200ar libX.a libX*.o
st200cc -r -nostdlib -o libX.ro libX*.o
st200ar libX.a libX.ro
build of the main driver objects
st200cc -c main*.c
creation of the final executable
st200cc -o main.exe main*.o lib*.a

The purpose of the intermediate link in relocatable mode (-r) is to improve spatial locality by ensuring that all functions in the same module are placed close together.

The -nostdlib option, which disables the inclusion of the standard libraries at link time, has to be added in order to prevent multiple definitions at the final link stage

Though it is very simple, this first optimization step is crucial to achieve optimal I-cache performance and will always be associated with more complex optimization methods.

5.5.2 Static Icache optimization:

As stated in the previous section, performing an intermediate link of each module helps to improve the spatial locality of the binary code. Nevertheless, the placement of the functions an each module library is not controlled at all by this method. The use of the binary optimization tools is needed to optimize the function layout, by performing a function reordering before creating each library. This I-cache optimization phase is automatically performed at link-time when using optimization levels -O2, -Os or -O3 (the command line option has to be given both at compile and link-time). I-cache optimization can also be explicitly turned on using both –icache-opt=on and –icache-static=on options at link-time.

By default, a static I-cache optimization is performed, which means that the binary optimizer reorders the functions according to static compiler-estimated frequencies. These frequencies correspond in fact to the weight of each edge of the static call-graph, which is computed by binopt and used by the reordering algorithm.

At the final global link phase, the I-cache optimizer must be turned off, to avoid modifying the previously locally optimized function layout in each library. It is turned off by default at optimization levels -O0 and -O1, but can be explicitly turned off using the –icache-opt=off option

The build commands sequence is modified in the following way: Optimized compilation of the libX*.c sources
for process X (X=1,2,..,N)
Intermediate link and I-cache optimization
phase (triggered by the -O2 flag)
st200cc -O2 -r -nostdlib -o libX.ro libX*.o
creation of the I-cache optimized libX.a library
for process X (X=1,2,..,N)
st200ar libX.a libX.ro
build of the main driver objects
st200cc -c main*.c
creation of the final executable
(with I-cache optimizations turned off)
st200cc -o main.exe main*.o lib*.a –icache-opt=off

A set of options is provided to help the static I-cache optimizer better estimate the static call-graph of the application. These can be passed to the binary optimization phase through the st200cc driver using the -Wo,[option] syntax.

in the context of several modules packaged in separate libraries, the use of an I-cache call-graph user configuration file (.icg file) may be helpful, as it allows edges to be added to or removed from the static estimated I-cache call-graph. For example, in the case of indirect calls, the simple relocation analysis performed by binopt is not able to catch these calls. The corresponding edges can then be added manually into the .icg file.

The .icg file consists in a list of items:
Proc1 proc2 freq
where proc1 and proc2 are function names and freq is a floating-point value which is an estimation of the numbers of execution of proc2 each time proc1 is executed. The value of freq is in the range [0.0,+inf).

Then, the .icg file is passed to the binary optimization phase at module link time: The build commands sequence is modified in the following way:
Intermediate link and I-cache optimization
phase with .icg file use
st200cc -O2 -r -nostdlib -o libX.ro libX*.o -Wo,–icg,libX.icg
Note: The edges described in the .icg are added to the pre-existing call-graph estimated by binopt and thus the .icg file must not contain all the call-graph edges. To remove an edge, for instance between proc1 and proc2, a line with a frequency value of 0.0 must be added:

remove the edge between proc1 and proc2
proc1 proc2 0.0
Now for a practical example. Suppose you want to optimize the I-cache miss cycles for an application containing the following piece of C code:
define N 10

```
extern void proc2(void);
void proc1(void (*proc)(void))

int i;
for (i=0; i¡N; i++)
(*proc)();
void proc3(int i)
if (i)
proc1(proc2);
```

By default, the binary optimizer only sees the edge between proc3 and proc1 (corresponding to the call to proc1 in proc3), putting a frequency of 0.5 (the default frequency value for a single block if construct). But supposing that on average, then if condition in proc3 is true for 90% of the cases, this information can be given to the I-cache optimizer using an .icg file.

In the same way, the call of proc2 from proc1 is indirect, so it is not taken into account by binopt. To fix this, we added an entry to the .icg file.

The .icg file for this example could be the following:
add the edge between proc1 and proc2 (frequency of N=10.0)
proc1 proc2 10.0

refine the frequency estimation of the proc3/proc1 edge,overriding the compiler estimation of 0.5
proc3 proc1 0.9;

Chapter 6

Methodology for optimizing Icache in ST231

The next is to define a methodology for optimizing I-cache on the ST200 based on machine learning approach i.e Genetic algorithm. It is most relevant to cores that have a direct mapped I-cache such as the ST231.and also using the "dynamic I-cache optimization and " gprofile driven icache optimization method".

6.1 Dynamic I-cache optimization method

Static I-cache optimization can be inaccurate because it relies on static estimations of frequencies based on heuristics. To improve the accuracy of the function reordering, the use of dynamic I-cache optimization methods is preferred. Basically, this uses run-time data instead of estimations. It implies that the application must be first run to collect this data by profiling, before rebuilding the executable to take advantage of the profiling information collected. There are two kinds of dynamic I-cache optimization methods: using data from a gproffile to drive the I-cache optimizer, and using the profiling feedback optimization (PFO)

6.1.1 gproffile driven icache optimization method

In this method, the module sources are compiled using the -pg option, which instruments the code to collect gprof-formatted profiling information at run-time. The commands for this first build are:
build and creation of the libX.a library
for process X(X=1,2,..,N)
st200cc -c libX*.c st200ar libX.a libX*.o
build of the main driver objects
st200cc -c main*.c
creation of the final executable
st200cc -o main.exe main*.o lib*.a

The mainpg.exe executable, instrumented by the compiler produces profiling information. The next step is to run each process with a significant input data set. Each run produces a file named gmon.out.000, which may be renamed to gmonX.out (for process X=1,2,..,N) to attach it to the process that it describes. In the case of several runs the same process (producing several gmonX.out.xxx files) the following commands can be used to sumup all the collected information in a single gmonX.out file. This may take a while to execute: sumup the profiling information in the default gmon.sum file st200gprof sum mainpg.exe gmonX.out.
Rename the output file gmon.sum to gmonX.out.
After having run all processes and produced all gmonX.out files, the last step is to rebuild the application, taking advantage of the profiling information. This is achieved by passing at module link-time the two options:
icache-profile=gmonX.out,
icache-profile-exe=mainpg.exe
So we are proposing to avoid this kind of issues by using Genetic Algorithm for improve the performance in the sense of speed. The optimization level at module

compile-time must be the same as the one used for the first compilation, to be consistent with the profiling instrumentation (-O2 in this case).

optimized compilation of the libX*.c sources
for process X(X=1,2,..,N)
st200cc -c -O2 libX*.c
Intermediate link in relocatable mode with profile file-driven creation of the I-cache optimized libX.a library
for process X(X=1,2,..,N)
st200ar libX.a libX.ro
build of the main driver objects
st200cc -c main*.c creation of the final executable (with I-cache optimizations turned off)
st200cc -o main.exe main*.o lib*.a (icache-opt=off)

This method we used on the ST200 simulator . But if it is possible to run the application code on the simulator, an alternative equivalent method can be applied, instead of using the -pg option of the compiler, the profiling feature of the simulator can be used to produce the gmonX.out file. The run command is as follows: run X process (with X args arguments) in ST231 littleendian mode using the profiling mode of the simulator.

st200xrun -c st200sp -t st231profsimle -e main.exe -a Xargs
Rename the output file gmon.out to gmonX.out.

6.2 Methods of Icache optimization

The instruction cache is a small memory with fast access. All binary instructions of a program are executed from it. In the ST231 processor from stmicroelectronics, the instruction cache is direct mapped: let L be the size of the cache; the cache line i can hold only instructions whose addresses are equal to i modulo L. When a program starts executing a block that is not in the cache, one must load it from main memory; this is called a cache miss. This happens either at the first use of a function (cold miss), or after a con ict (con ict miss). There is a con ict when two functions share the same cache line; each of them removes the other from the cache when their executions are interleaved. The cost of a cache miss is of the order of 150 cycles for the ST220, hence the interest of minimizing the number of con icts by avoiding line sharing when two functions are executed in the same time slot. This problem has in fact following objective functions:

- **COL** Minimizing the number of con icts for a given execution trace. This can be reduced to the Max-K-Cut and Ship-Building problems.
- **EXP** Minimizing the size of the code. This is equivalent to a traveling salesman problem (building an Hamiltonian circuit) on a very special graph called Cyclic-Metric.
- **NBH** Leaving spaces or uncalled procedures between successively-called procedures yields cache misses: if two successively-called procedures share the same cache line, a cache miss is saved when the second procedure is called, since it is already in the cache. Another similar phenomenon is instruction prefetch, where instructions are loaded from the memory before the processor needs them, according to a branch prediction policy. Therefore, increasing code locality is good in order to avoid cache misses.
- **GS-prof** The Genetic Search algorithm is based on a con ict graph built using profiling. In practice, it leads to a much better con ict reduction than any

other known static based approach. Genetic algorithms, for instance, have been used effectively in loop tiling optimization, offering fast convergence and near optimal solutions, genetic al- gorithm techniques Here, a set of candidate solutions are evolved toward an optimal solution, as measured by program execution time. This design functions as a general optimization heuristic driven by actual performance, and follows current interest in machine learning approaches to optimization

6.3 Design

The heuristic instruction cache optimization algorithm we are implementing is meant to serve as a point of comparison for our own optimization algorithm. In this section, we present the optimization framework on which both implementations are based, the motivation behind the two approaches, as well as the implementation details of our code rearrangement system. We also doing our investigation of code-expansion due to "padding" as a potential source of overhead introduced by our implementation.

6.3.1 Optimization Framework

We have designed a framework to facilitate the implementation and the comparative testing of multiple instruction cache optimization strategies. This framework is incorporates interfaces to tools such as the GNU compilers, the GNU profiler. Code optimizations are implemented as plug-in components that can access existing components of the framework.

Our framework is designed around the transformation of benchmark programs. These programs are provided as input in source code form, and can then be compiled either directly into executable binary form, or into assembly. Interfaces are provided to gather information about the assembly source (i.e., location and type of symbols in the said source), and to gather profiling data from executable binaries (i.e, running time, run-time call-graph, "hot functions", cache use profiling).

Code optimizations, which we refer to as "transformations", can use this information in deciding how to transform the assembly source. Because we are specifically interested in instruction cache optimizations, our framework includes a layout modifier component which can be used to modify assembly source so as to change the memory alignment of functions. This component can align functions so that they will map to specific cache lines, for example. Once transformed, assembly source can be compiled into binary form and profiled. It can then be transformed again based on new profiling data.

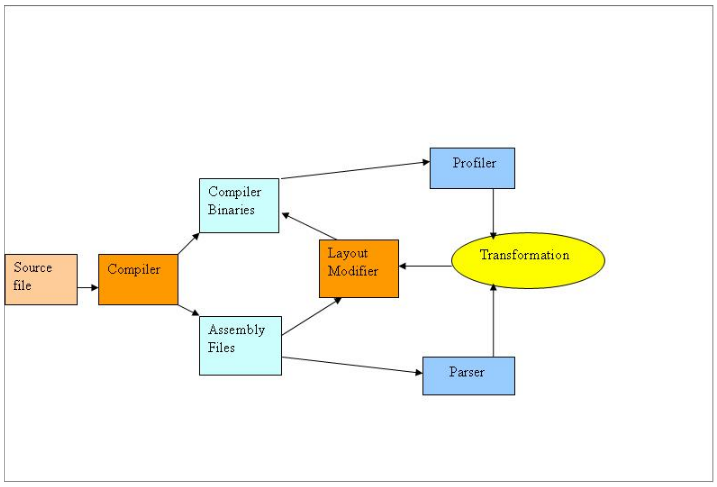


Figure 6.1: Information flow among the components of our framework

Figure 6.1 illustrates the basic idea behind the layout modifier component of our frame- work. In this example, four functions A, B, C and D are linearly mapped in memory, and take up 1, 1, 3 and 2 cache lines, respectively. Functions C and D originally map to cache lines 0 and 3 of an imaginary 4-line direct-mapped cache. To map functions C and D to cache lines 1 and 2, our layout modifier would insert "padding" space equivalent to one cache line in between B and C, and space equivalent to two lines between C and D. The resulting mapping has the same linear order as the original, but takes up slightly more memory. This approach has the advantage that it does not affect the branch prediction behavior of the processor.

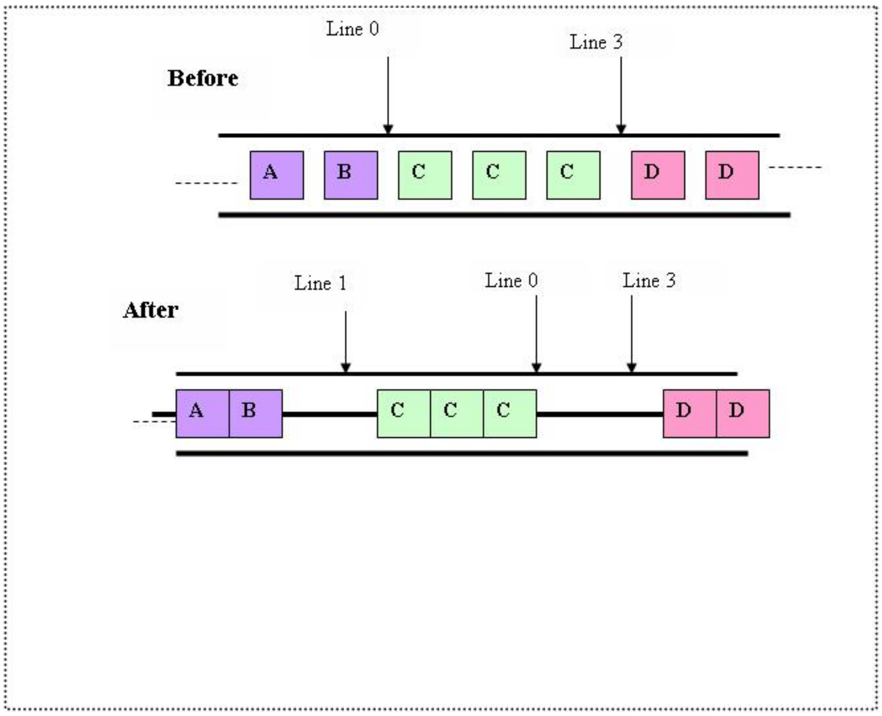


Figure 6.2: padding in cache line

Thus far, our framework incorporates support for programs implemented in C, through the GNU gcc, respectively. It also incorporates a benchmarking system that can automatically test multiple code optimizations on a sequence of benchmark programs and record the running-time of each program before and after optimization into a spread- sheet, along with profiling information such as the number of instruction cache misses.

6.4 Search algorithm

We base our learning approach to optimization on genetic algorithms. The main motivation is the principle of locality. That is, we believe that that memory mapping with similar parameters is likely to yield similar performance, and thus that better performing mappings are likely to be ”close” in the space of possible solutions. It is difficult to derive a simple yet accurate formula to predict the cache performance of a given mapping, Genetic algorithms are ideal for situations where the principle of locality holds and one has an effective way of evaluating the fitness or ”goodness” of a solution.

Our optimization strategy searches the space of possible instruction cache mappings for a given program. To do this, it generates versions of the said program with altered memory alignments. This is done in a typical genetic algorithms fashion. An initial set (population) of modified programs is generated. Then, for each generation, new individuals (new modified programs) are generated.

These new programs replace the existing programs whose fitness measurement is lowest, so that the population size remains fixed at each generation. The generational process is repeated as long as desired. In the end, the individual with the highest fitness measurement is chosen as the output of our optimization algorithm.

These new programs replace the existing programs whose fitness measurement is lowest, so that the population size remains fixed at each generation. The generational process is repeated as long as desired. In the end, the individual with the highest fitness measurement is chosen as the output of our optimization algorithm.

Genetic algorithms are designed to maximize the average fitness of the population over time. Hence, to optimize instruction cache performance, the fitness value of a given program should ideally be inversely proportional to the number of instruction cache misses the program obtain.. We must also consider that in processors possessing an L2 cache, maximizing the performance of the L1 instruction cache will not necessarily increase the performance. Finally, the number of instruction cache misses a program experiences can vary depending on run-time context, scheduling and I/O.

For all these reasons, we chose the running- time as our fitness criterion, which our algorithm tries to minimize over time.

The use of genetic algorithm in any search problem requires the definition of a set of elements and operators: representation of the solutions (codification), a fitness function to evaluate the different solutions, a selection scheme to sort candidate individuals for breeding, cross-over and mutation operators to transform the selected individuals.

- **Codification** Each individual, representing a possible solution, is a bitmap of size nl, with nl the total number of program lines for all tasks.A bit set to 1 means that the corresponding program line is locked into the cache.
- **Fitness** The fitness function is the weighted average of all tasks response times (see equation 1, where Ri denotes the response time of task ti. This fitness function aims at improving the average response-time of tasks, precluding that the genetic algorithm assigns all available cache blocks to the higher priority tasks.
  Fitness = R0 + R1 + 2R2 + 4R3 + ::: + 2N-2 RN-1 / 2N-1
- **Crossover and mutation** One point crossover is applied: an index into the parents chromosomes is randomly selected. All data beyond that point in the chromosomes is swapped between the two parent organisms, defining the children chromosomes. Three types of mutations have been introduced:

M1. random reduction of the number of locked program lines,
M2.random increase of the number of locked program lines,
M3. random modification of the identity of one locked program line, the total number of locked program lines being left unchanged.Rule M1. (resp M2.) applies to invalid individuals whose number of locked program lines is greater than (resp. lower than) B, while rule M3 applies to valid individuals.

6.4.1 Initial population and algorithm parameters

The initial population is made of valid individuals only. The bitmap of every individual in the initial population has B consecutive bits set, the index of this series of 1 being randomly selected. The other parameters of the algorithm are given in table. These parameters spring from a set of experiments where the behavior of the genetic algorithm was studied.

Table I: Parameter of Genetic algorithm

Parameter	**Value**
Population size	200
Number of Generation	5000
Num Probability of crossover	.6
Probability of mutation (rules R1..R3)	0.01
Probability of selection of the individual with the highest rank	0.1

An interest of genetic algorithms is that the produced results (here, cache contents) can be used at any time, that is, it is not necessary to wait the algorithm end to get partial results.

6.5 Experimental setup

6.5.1 Hardware configuration

We consider a ST231 processor in which an instruction cache and a 16B (4 instructions) instruction prefetch buffer. The cache configurations used in the performance comparison are given in table. In addition, since we are only concerned with timing

Table II: Cache Parameter

Cache Parameter	**Value**
Block Size	16 bytes (4 instructions)
Cache structure	direct-mapped
Num Cache size	[1Kb .. 64 Kb]

the cache behavior, we adopt a very simple timing model for instructions. An instruction is assumed to execute in Thit = 1 processor cycles in case of a hit in the instruction cache or prefetch buffer, and in Tmiss = 10 processor cycles otherwise.

6.5.2 Workload

The algorithms we will compare using 26 different synthetic task sets. Synthetic tasks are generated by an STautomatic tool. Input parameters to this tool are the size of the task, number of loops and nesting level, size and iterations of loops, number of if-then-else structures and their respective sizes. The user must provide the minimum and maximum desired values for these parameters. The tool randomly selects the actual value from this range. For the accomplished experiments, these parameters will be take out from usual embedded workload, Number of tasks per task set [3..8] Maximum task set code size 64KB Number of different tasks 50 Tasks code size [1KB..32KB]

Table III: Workload parameter of Genetic algorithm

Workload Parameter	**Value**
Number of tasks per task set	3..8]
Maximum task set code size	64KB
Num Number of different tasks	50
Num Tasks code size	1KB..32KB]

Within a task set, the task periods Ti, equal to their dead- lines Di is to be adjusted such that the system is schedulable with both conventional and locking cache. An experiment is defined by a pair (task set, cache size). 146 experiments is to be conduct. Only experiments whose total code size is larger than the cache size have been considered in the following analysis, on the one hand because this is the most realistic situation and on the other hand because algorithms would behave identically for task sets smaller than the cache size.

To increase this effectiveness, STMicroelectronics has developed a binary optimization tool called binopt for ST200 which includes an I-cache optimization tool. The I-cache optimization phase of binopt controls the layout of functions in the binary code to be executed on the ST200. This is achieved by using cache line coloring algorithms to reorder the functions in the final executable. The cache line coloring algorithms minimize I-cache conflicts by optimizing spatial locality and thus optimize I-cache usage.

As the problem with the cache line coloring algorithm was speed loss. Because for L1 cache it was succeed to reduce the instruction cache misses, But for L2 cache coloring algorithm causes an average increase of 10.13percent instruction cache misses. This is likely due to the fact that coloring algorithm can only concern itself with one of the two caches. As it tries to reduce L1 instruction cache misses, it may increase the L2 cache misses.

So we are proposing to avoid this kind of issues by using Genetic Algorithm for improve the performance in the sense of speed.

- The evaluation of previous work on i-cache optimization based on cache line coloring in the context of ST231 architectures. We extend the design to handle set-associativity, and show that while performance differences are generally variable and within noise, significant positive improvements, up to nearly 6 percent can be achieved. The coloring implementation is less successful.

- We will explore the use of genetic algorithms for i-cache optimization. With increased complexity due to evolving hardware, OS, and software designs, learning-based approaches have significant promise as a means of increasing performance. We are able to show a consistent if small improvement 0.5 %, and up to 10 percent for specific benchmarks.

Here, a set of candidate solutions are evolved toward an optimal solution, as measured by program execution time. This design functions as a general optimization heuristic driven by actual performance, and follows current interest in machine learning approaches to optimization.

So the cache coloring algorithm we consider has only a simple and coarse model of cache design, while from learning algorithm we will have an ideal model in the sense that it adapts to the actual performance of the underlying architecture.

We are using the Genetic algorithms, for instance, in loop tiling optimization, which offering fast convergence and near optimal solutions. We examine register allocation and data prefetching in particular, and show promising, but variable results, suggesting sensitivity to genetic algorithm parameters

The aim is to define a methodology for optimizing I-cache usage on the ST231, which applies to real user application code. Firstly, the context of this application note is defined. Before implementing the genetic algorithm for this tool we are using the different I-cache optimization techniques that are stated and described. Finally, the results on a chosen test-case application will presented.

Chapter 7

Implementation of GP for icache optimization

Genetic programming breeds computer programs to solve problems by executing the following three steps:

- Generate an initial population of random compositions of the functions and terminals of the problem (i.e., computer programs).

- Iteratively perform the following substeps until the termination criterion has been satisfied:

  Execute each program in the population and assign it a fitness value using the fitness measure.

  Create a new population of programs by applying the following operations. The operations are applied to computer program(s) chosen from the population with a probability based on fitness.

REPRODUCTION : Reproduce an existing program by copying it into the new population.

CROSSOVER : Create two new programs from two existing programs by genetically recombining randomly chosen parts of two existing programs using the crossover operation (described below) applied at a randomly chosen crossover point within each program.

MUTATION : Create one new program from one existing program by mutating a randomly chosen part of the program.

3 The program that is identified by the method of result designation is designated as the result for the run (e.g., the best-so-far individual). This result may be a solution (or an approximate solution) to the problem.

Before applying genetic programming to our problem, we must perform five major preparatory steps. These five steps involve determining:

a. the set of terminals,

b. the set of primitive functions,

c. the fitness measure, mutate

d. the parameters for controlling the run, and

e. the method for designating a result and the criterion for terminating a run.

The first major step in preparing to use genetic programming is to identify the set of terminals. The terminals can be viewed as the inputs to the as-yet-undiscovered program. The set of terminals (along with the set of functions) are the ingredients from which genetic programming attempts to construct a program to solve, or approximately solve, the problem.

Second step involves determining the function set which may be any arithmetic operation, standard programming operators, standard mathematical function, logical function, or domain-specific functions.

Fitness measurement controls the flow of GP which evaluates how well each computer program in the population performs in its problem.

The Primary parameters for GP are the population size and generation size while secondary parameters are quantitative and qualitative variables used to control the run of GP.

A precondition for solving a problem by GP is that the set of terminals as well as the set of functions satisfy sufficiency requirement in the sense that they are together capable of expressing a solution to the problem.

In the cache coloring algorithm, arriving at a high quality solution involved a very large number of evaluations and consequently is computationally demanding. Fortunately, the number of intrinsic parallel nature of Genetic algorithm makes them suitable for a parallel implementation. By GA, it is possible to:

- Reduce the time to locate a solution(Faster algorithm)
- Reduce the number of function evaluations (cost of the search),
- Have a larger populations
- Improve the quality of the solution worked out.

Our algorithm has three phases: the initialization phase, evaluation phase and tournament phase. Each of these phases contains one or more task as illustrated in the following algorithm:

```
/*INITILIZATION PHASE*/
Startup MPI();
Current gen := 0;
fitnessIndiv := none;
InitializePopulation (CurGen);
```

```
/* EVALUATION PHASE*/
While (CurGen ¡ MaxGen) do
/*Evaluation And Ranking */

    EvaluationAndRankPopulation (CurGen);
/* Preserve Fittest Individual */
Winner := SelectBestIndividual (CurGen);
FittestIndiv := Select (FittestIndiv , Winner);

    /* Migration */
Emigrant := SelectAndCloneIndividual (CurGen);
Asyncsend(emigrant);

    /*Crossover */ SelectParents (CurGen);
Offspring :=MateParents (CurGen);
/*Immigration*/
AsyncReceive (immigrant);

    /*Integration*/
IntegrateImmigrant (Offspring , immigrant );
/*Mutation */ CurGen := ApplyMutation (offspring);
CurGen := CurGen+1;
End while;
End program;
```

7.1 Description of the Algorithm

a. Each separate processing element independently generates its own initial semi-isolated random subpopulation. This process of initial random creation takes place in parallel at each processing element. As soon as each separate processing element finishes this one-time task, it begins the evaluation phase loop.

b. In the evaluation phase, the task of measuring fitness of each individual is first performed locally at each processing element, then the individuals are locally according to a given fitness function: this selection steps is performed locally at each processing element.

c. One individual in each subpopulation is selected at random for emigration from each processing element to other processing elements, to introduce new breeds into each subpopulation.

d. Crossover is performed locally at each processing element. The processing element operates asynchronously in the sense that each generation starts and ends independently at each processing element. Because each of these tasks is performed independently at each processing element, and because the processing elements are not synchronized, this asynchronous approach efficiently uses all the processing power of each processing element

e. Previous to the stage of mutation (or better yet, as a part of it) immigration takes place. It is done asynchronously, so that the algorithm does not stop wait for any slow processing element.

f. Once the termination condition in every processing element have been satisfied, in order to choose the best individual, a tree-like tournament selection will be used, requiring at most log2 P steps, where P is the number of processing element.

The GA provides, for every task in the task set, the subset of blocks to be locked. It also brings an estimation of the WCET of each task with the chosen set of blocks already loaded, and the WCRT of all tasks considering the estimated WCET.

Therefore, experiments were conducted to determine the reduction in execution time and the quality of the solution. The GA job is to select cache contents for different cache size ranging from 64 to 4096 cache lines and task sets with between 3 to 8 tasks each. Tasks have the usual statements found in any program. The whole set has 123 different experiments and it was executed on 1,2,4,6,8 and 10 nodes.

To measure the quality of the solution, Processor Utilization a commonly used metric to evaluate real time systems performance was used. The lower the processor utilization, the better, since this means that the task set demands less CPU time and thus other tasks might be included in the task set while the system remains schedulable (i.e. all task executing on time).

7.2 WCET and response time computation

Tasks worst-case execution times estimates (WCETs) will compute using a tree-based approach. The WCET estimate of a task is computed using recursive formulas that operate on the task's syntactic tree (a node in the tree represents a control structure, loop, conditional, sequence of blocks, a leave represents a basic block. branch-free sequence of instructions). As we voluntarily ignore hardware components other than the instruction cache, the WCET estimate of a basic block BB can be computed in a straightforward manner from the WCET of its program lines pl: WCET(BB) = Pnpl pl=1WCET(pl), with WCET(pl) = Thit or Tmiss depending on whether program line pl has been locked in the instruction cache or not.

The response times of tasks are computed using CRTA (Cache-aware Response Time Analysis), which extends the well-known exact response time analysis (RTA) schedulability test to take cache-related preemption delays into account. Given a task ti, CRTA works by considering the interferences produced by the execution of the higher priority tasks on ti within an increasing time window wn I (n is the recurrence index). The response time Ri of task ti is the fixed point of the sequence given in equation 2 below, where hp(ti) denotes the set of tasks that have a higher priority than ti and denotes the cache-related preemption delay (the equations assume tasks with distinct priorities).In our context, the value of gamma is the time needed to reload the prefetch buffer plus the related context switch penalty .

$$W_i^0 = C_i \tag{7.1}$$

$$W_i^{n+1} = C_i \sum_{t_j \epsilon h_p(t_i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil * (C_j + \gamma) \rightarrow R_i(2) \tag{7.2}$$

7.3 Comparison metrics

From the cache contents generated by the two algorithms, we present a statistical analysis of the performance of the resulting task sets. We focus on worst-case performance metrics, computed from the tasks WCETs and the tasks response times.

7.4 Processor (worst-case) utilization (U)

The processor utilization Σ Ni=1 C'i /Ti , where C'i is the worst-case execution time of task ti including all cache effects) is an interesting metric because it allows to know a lower bound of the overall spare ST231 processor capacity, which can be used for instance for executing soft real-time tasks. The lower the utilization, the better Genetic Search Algorithm.

7.5 Algorithm comparison

7.5.1 Comparison of utilization

Uci (resp. Ugi) denote the (worst-case) utilization of an experiment i when using the Cache Coloring algorithm (resp. genetic) cache contents selection algorithm. Since the lower the utilization the better the performance, values of Δ Ui = Uci - Ugi below zero demonstrate the superiority of the Cache Coloring algorithm, whereas values above zero demonstrate the superiority of the genetic algorithm. Values of Δ Ui are in the interval] - 1; +1[. Table gives statistics on values of Δ Ui = Uci - Ugi. The average value of Δ Ui is very close to zero, whereas its median value is zero. The standard deviation and quartiles show that the vast majority of values are very close to zero.

Table I: Statistics summary for Δ Ui = Uci-Ugi

Number of experiments	146
Average	0.0125386
Median	0
Standard deviation	0.0539221
Minimum	-0.05984
Maximum	0.281344
Experiments with p is less then 0	69 (47.3 in percentage)
Experiments with p=0	11 (7.5 in percentage)
Experiments with p is greater then 66	(45.2 in percentage)
95 percent confident interval for average	[0.0037184; 0.02135881]

Confidence interval for average, that contains the average value, allows considering the average as the true average. However, there are two peculiarities. Firstly, the number of experiments with ?Ui below zero is slightly greater than the number of Δ Ui values over zero. Secondly, the minimum and maximum, we found that Δ Ui values are greater when the genetic algorithm provides better performance.

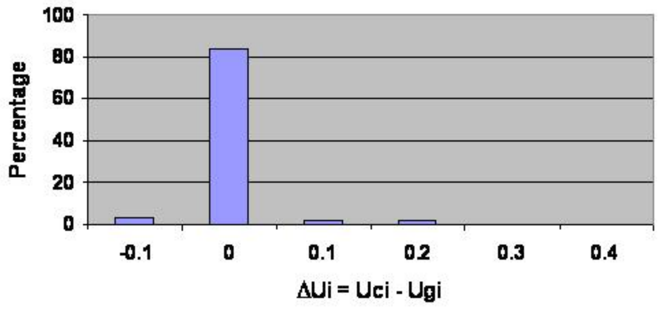


Figure 7.1: Frequency histogram (percentage of values of Δ Ui = Uci - Ugi inside the range pointed by the xaxis)

7.5.2 Execution time of algorithms

Blocks to be locked in cache are selected during design phase, and thus the speed of cache contents selection does not affect the performance of the actual system. However, the speed of the algorithm for cache contents selection may be important if source code of the system tasks is modified frequently. The execution times of both algorithms have been measured on ST231, 200Mhz processor running Linux. Both algorithms have been implemented in C. Over the 146 experiments we carried out, the Cache coloring algorithm always executes in less than two minutes. In comparison, the genetic algorithm took between two and six hours to execute for most experiments, and in some cases more than ten hours. Obviously, the Genetic Search algorithm is extremely faster than the Cache coloring algorithm.

7.6 Cache misses

Cache optimization work is typically evaluated in terms of cache performance, with reduced cache misses suggesting a program will improve in speed or cache resource requirements. In the presence of multiple cache levels, however, optimization with respect to one cache level does not necessarily translate into better overall performance.
In tables , we present for each of our benchmarks and for both optimization algorithms the relative percentage improvements (positive values are good) obtained in terms of running time on the ST231 system with compiler optimization level3. Along with this, we also show the relative improvements obtained in terms of instruction cache misses and in terms of L2 cache misses for instruction fetches, as well as the approximate number of cache misses before optimization.

Table II: Cache and performance effects of Cache line coloring algorithm

Benchmark	**Time**	**L1%**	**L2%**	**L1 Imisses**	**L2 Imisses**
ammp	-0.03	-32.11	-68.33	0.4M	76K
gcc	0.14	-8.37	-12.00	15M	3M
mcf	-0.10	-22.45	-22.80	1.4K	1.4K
mesa	-0.47	93.02	15.54	1.3M	77K
radiant	-1.90	93.79	-0.58	16M	2.7K
twolf	-0.73	-60.12	29.92	33K	28K
wupwise	0.72	-12.47	-12.69	4.7K	4.4K
average	-0.37	7.33	-10.13	4.7M	0.5M

We note that our genetic algorithm obtains a 0.5 %speed improvement on average, vs. a 0.34% speed loss, on average, for the coloring algorithm. What is most interesting is that both algorithms succeed at reducing the L1 instruction cache misses. However, whereas the genetic algorithm also reduces the L2 instruction misses by 2.36% on average, the coloring algorithm causes an average increase of 10.13% for L2 instruction misses. This is likely due to the fact that the coloring algorithm can only concern itself with one of the two caches. As it tries to reduce the L1 instruction

Table III: Cache and performance effects of Genetic algorithm

Benchmark	**Time**	**L1%**	**L2%**	**L1 Imisses**	**L2 Imisses**
ammp	0.42	-33.57	-33.97	0.4M	76K
gcc	0.45	29.95	28.92	15M	3M
mcf	0.31	-7.64	-7.80	1.4K	1.4K
mesa	0.51	-2.34	22.53	1.3M	77K
radiant	-0.33	95.73	-0.58	16M	2.7K
twolf	-0.41	40.73	32.60	33K	28K
wupwise	2.59	-24.12	-25.16	4.7K	4.4K
average	0.50	14.11	2.36	4.7M	0.5M

cache misses, it may increase the L2 cache misses. Our algorithm does not directly optimize for the cache, but rather for performance, and so avoids this kind of issue. Only concern itself with one of the two caches. so avoid this kind of issue.

Interestingly, we note that, according to our data, an improvement in cache performance does not necessarily correlate with a performance improvement and vice versa. The genetic algorithm, for example, obtains a 2.59% speedup for benchmark, while its L1I and L2 cache performance decrease significantly. On the other hand, the coloring algorithm reduces the L1I cache misses on the other benchmark by 93.02%, but its performance decreases by 1.90%. We also note that the degree of cache performance improvement obtained by both algorithms seems to be proportional to the number of cache misses before optimization. For example, both algorithms obtain a cache performance decrease on the wupwise benchmark, but this benchmark also has the least cache misses before optimization, while both algorithms improve the L1I cache misses of the radiant benchmark by more than 93 percent from a very high original count of 15 millions.

7.7 Convergence

One important concern when it comes to genetic algorithms and other machine learning algorithms is that of convergence. Our algorithm has many user-adjustable parameters that can affect the rate of convergence, and whether or not convergence occurs at all. The parameters we have chosen for our tests were based on educated guesses, preliminary experimentation, and experimentation time constraints.

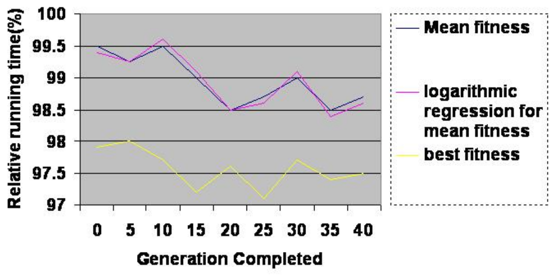


Figure 7.2: Average relative fitness over time

In figure 7.2, we examine the relative mean and best fitness (running-time) values averaged over our different benchmark and platform combinations. The mean fitness of the population appears to decrease in an asymptotic, logarithmic manner that is typical of machine learning algorithms. That is, the decrease becomes less and less significant as time passes by because the algorithm is converging to a theoretical maximum performance. To illustrate the convergence trend, we have performed a logarithmic fit of the mean fitness over time, which we also illustrate. The R2 coefficient of this fit is 0.96, supporting the idea that convergence is indeed occurring, and that the trend is logarithmic in nature. We can also observe that the best fitness obtained decreases in a similar fashion, and is noticeably lower than the average fitness of the population.

We have implemented our Genetic algorithm for Binopt tool in I-cache optimizer phase. This takes an executable and execution profile information as inputs, and then performs various optimizations level and produces an optimized executable.

The particular optimizations of interest in to us in binopt are inlining and code reordering. When performing inlining, binopt uses the following heuristics:

- " inline if this procedure is very small (i.e., if the call and return instructions will take up more space than the procedure's body),
- " inline if the call site being processed is the only call site for this procedure, or
- " inline if the call site is activated very frequently and the resulting cache footprint does not exceed the instruction cache size.

The decrease in execution time achieved with inlining alone with binopt is reported to be as much as 4.3%.

binopt also performs interprocedural basic block layout which can be guided by profile information. If profile information is available, binopt tries to avoid cache conflicts by grouping the basic blocks into hot, cold and zero sets and applying the bottom-up positioning approach that using this basic block layout algorithm can decrease execution by 17%.

Our procedure reordering algorithm starts after binopt completes basic block layout, and before the scheduler is invoked. Since binopt performs interprocedural basic block reordering, a procedure's basic blocks can be spread out all over the image space.

However, one of the procedure mapping algorithm that we have implemented requires that procedures are kept as a whole. So, before our Genetic algorithm starts, we first examine the layout created by binopt and merge all basic blocks associated with a single procedure together. After this phase, we have a sequence of procedures, with the basic blocks rearranged within a procedure, and with the hottest basic blocks at the beginning of each procedure in the image.

The procedure splitting algorithm that we implemented, if activated, starts processing the layout at this point. The algorithm analyses each procedure by using the profile information and brings the activated and unactivated basic blocks together in the layout.

7.8 Instrument the code

The first step for performing the PFO is to generate an instrumented executable. This is achieved by adding the -fb command line option at compile-time. This option must also be passed at final link-time, in order to tell the linker to add the libinstrC.a library that contains the routines collecting the feedback information (in particular the frequency counts on control flow).

The modified build commands are:

```
- compilation with profiling feedback instrumentation
- of the libX.c sources for process X (X=1,2,..,N)
st200cc -O2 -fb create fb data -c libX.c
- Intermediate link in relocatable mode
st200cc -O2 -r -nostdlib -o libXpfo.ro libX.o
- creation of the pfo instrumented libX pfo.a library
- for process X (X=1,2,..,N)
st200ar libXpfo.a libXpfo.ro
-build of the main driver objects
st200cc -c main*.c
- creation of the final executable including the instrumented libraries
st200cc -fb create fb data -o mainpfo.exe main*.o
```

7.9 Collect feedback data

The second step is to gather the feedback data by executing each process, either on the simulator or on the board. During each execution, a frequency data file named (name).instr0. is created (in this example (name) is fb data). Unlike the gprof file-driven method, there is no need to create a single summary feedback file in the case of multiple feedback information files for the same process. To associate each data file to its process, it is convenient to create one directory fb.X.dir for each process X (X=1,2,..,N), and to move the data files into the associated directory. Finally, as the instrumented and to move the data files into the associated directory. Finally, as the instrumented executable runs significantly slower than usual, it is recommended to use the fast mode of the simulator (if the program can execute on the simulator) to speed-up the execution. Here is an example of how to run and collect feedback information for process X with two different sets of arguments (X.args1 and X.args2):

Create the feedback data directory fb.X.dir for process X.
- run X process (for both X.args1 and X.args2 arguments) in ST231
- little-endian mode using the fast mode of the simulator

st200xrun -c st200sp -t st231fastsimle -e main.pfo.exe -a X.args1
st200xrun -c st200sp -t st231fastsimle -e main.pfo.exe -a X.args2

Put the collected feedback data files in the data directory for process X by copying all
fb.data.instr0. files to the directory fb.X.dir
After this step, we have a set of feedback data for each process, placed in the. corresponding data directory.

7.10 Re-Compile

The third and last step is to re-compile all modules adding the -fb dir/name option, that tells the compiler to annotate the code, using the information gathered in all feedback files name.inst0.* in ¡dir¿. These annotations occur at a very high level in the compilation back-end process and the compiler uses them in later optimization phases such as if-conversion and instruction scheduling. This also affects the I-cache optimization phase, as the feedback frequencies are also seen by the I-cache optimizer and used to make decisions on code reordering.

Here are the build commands for this last step:

- optimized compilation with feedback annotations of the libX.c sources
- for process X (X=1,2,..,N)
st200cc -02 -fb fb.X.dir/fb.data -c libX.c - Intermediate link in relocatable mode
- (indirectly using the feedback annotations)
st200cc -O2 -r -nostdlib -o libX.ro libX.o - creation of the I-cache optimized libX.a library
- for process X (X=1,2,..,N)
st200ar libX.a libX.ro
st200cc -c main*.c
- creation of the final executable (with I-cache optimizations turned off)
st200cc -o main.exe main*.o lib.a

Furthermore, a dramatic improvement in I-cache miss cycles is observed for some processes. For instance 211% for the ac3 decoder, 148% for the aac decoder, and 118% for the Mpeg1-Layer3 encoder.

Nevertheless, some regressions are also encountered, but they remain reasonable: at most 13% in I-cache miss cycles (5% in total cycles) for the Mpeg1-Layer3 decoder. These regressions can be explained by the fact that, in the reference experiment without any specific I-cache optimization, the code placement is arbitrary, boosting some processes at the expense of the others. specific I-cache optimization, the code placement is arbitrary, boosting some processes at the expence of the others.

Chapter 8

Tools and Technique used

- ST231 development tools (from ST internal), including simulator and C compiler.
- Linux environment, shell script, perl, analysis for C, HDMS - multi-site unix based database management system (ST internal), CHT - unix-based configuration management system.
- Verilog, system verilog, PSL and simulators : Incisive Unified Simulator (Cadence),
- Verification automation tools: XXX (YYYY), eManager (Cadence),

Chapter 9

Conclusion and Future work

Conclusion

we have optimized both the Dcache side and Icache side in ST231 Dcache optimization done by replacing blocking cache with Nonblocking caches and Icache optimization done by Machine learning approach. By using Non-Blocking Cache Memory Latency get reduced. and it has save lots of time because of it unblocking nature.For some benchmarks, instruction cache optimization has a significant effect, even in the context of other, aggressive optimizations. This performance is not necessarily easy to extract with a simple heuristic and cache model, but as our genetic algorithm optimization shows, useful improvements are possible. Part of the difficulty is due to greater complexity in processor design; multiple cache levels, and other hardware optimization features make simple cache models less heuristically valid. Our basic coloring implementation, for instance, does reduce L1 cache misses, but not in such a way as to generally improve overall performance. A learning approach has the advantage of being able to optimize with respect to execution speed. This allows it to avoid inadvertent performance degradations due to an imprecise or incomplete performance model

Future Work

There are many interesting routes for futurework. Our framework lends itself to exploration of other algorithms and i-cache optimization designs, and we aim to further investigate other optimization routes. Our implementations, for instance, are constrained by the use of easy to gather profiling and Performance data-with more detailed, basic-block level profiling, better results may be possible, both in the learning domain and for deterministic approaches.

All these techniques were applied to the DVD/ACC audio codecs benchmark and produce a significant performance gain of more than 8% in total cycles for the main set of processes (5% for the overall set of processes). Almost 80% of this speed-up is achieved using static I-cache optimization. The remaining 20% is achieved using performance-feedback.

Website References

[1] http://en.wikipedia.org/wiki/Non-BlockingCacheTechnique

[2] http://en.wikipedia.org/MSHRQueue

[3] http://www.linkinghub.elsevier.com/

[4] http://en.wikkipedia.org/wiki/EffectsofCachebehavior

[5] www.specman.sim.org.uk/specman/

References

[1] ST Internal: Rogard Shephered "Implementation of Non-Blocking Cache",2009

[2] ST Internal: Nichola , "ST231 architecture ",2009.

[3] D. Lee. Instruction cache effects of different code reordering algorithms, Department of Computer Science and Engineering, University of Washington,.

[4] J. Abella. Near-optimal loop tiling by means of cache miss equations and genetic algorithms.In ICPPW '02, page 568, Washington, DC, US

[5] A. H. Hashemi, D. R. Kaeli, and B. Calder. Efficient procedure mapping using cache line coloring. In PLDI '97, pages 171-182, New York, NY, USA, 2000. ACM.

Index