PRACTICAL PRECISE EVALUATION OF CACHE EFFECTS ON LOW LEVEL EMBEDDED VLIW COMPUTING

Samir Ammenouche, Sid-Ahmed-Ali Touati, William Jalby
University of Versailles St-Quentin en Yvelines, France
Email: saam@prism.uvsq.fr, Sid.Touati@uvsq.fr, William.Jalby@uvsq.fr

KEYWORDS

non-blocking cache, load-use distance, ILP.

ABSTRACT

The introduction of caches inside high performance processors provides technical ways to reduce the memory gap by tolerating long memory access delays. While such intermediate fast caches accelerate program execution in general, they have a negative impact on the predictability of program performances. This lack of performance stability is a non-desirable characteristic for embedded computing. We will present the progress of our experimental study about the influence of cache effects on embedded VLIW processors (*ST2xx* processors). We are trying to understand qualitatively and quantitatively the interactions between cache effects (Data cache) and instruction level parallelism at different granularities: applications and functions (coarse grain), program regions (medium grain) and instructions (fine grain). Our aim is to come up with experimental arguments helping to decide whether non-blocking caches would be a *reasonable* architectural design choice for embedded VLIW processors. By reasonable, we mean bringing opportunities at two levels: 1) program execution acceleration with tolerable performance predictability, and 2) active interactions with compiler optimization techniques. Our study is based on many months of full-time simulations on tens of workstations producing many terabytes of data to analyse.

Introduction

Cache effects permit to hide the existing gap between memories and processors performances. However, caches have a negative impact on loads latency predictability, depending on dynamic data location in the memory hierarchy. Our goal is to understand the execution behaviour by taking into account the cache effects; we measure the impact of different cache architectures and also the impact of the compiler. We follow a practical approach with common benchmarks (*mediabench*) and less common applications(*ffmpeg*) This is a typical embedded multimedia application used by STmicroelectronics to design their chips. It a video compression basing on h263 standard which are precisely simulated. The used industrial simulator (implemented by STmicroelectronics) models an embedded processor which is the *ST231* of STMicroelectronics. We are targeting reducing memory cache penalty. Reducing processors stalls due to memory access latencies is an old goal for the community. Some software techniques are available, like tiling (2), loop permutation, loop fusion, loop unrolling, loops jam, software prefetching (3). Hardware techniques are also available such as hardware prefetching and non blocking caches (4).

The authors in (1) explain the interest of non-blocking cache architecture for the Out-Of-Order processor. We try to measure the benefit of such cache architecture for an In-Order processor. To achieve that, we used an embedded VLIW processor, the *ST231*. We have collected the simulation results of the benchmarks on the *ST231* using two cache architectures: a blocking cache architecture and a non blocking one. We make comparison of obtained results, and we proposed a compilation method to better exploit the non-blocking cache feature.

Current commercial and academic backend compilers schedule all loads using the cache hit latency. That is, cache misses latencies are not used during instruction scheduling. Assuming statically that all data reside in low level caches involves a great difference between real and expected execution times. Current compilers do not schedule the loads operations using a miss latency because: 1) The benefit of the cache would disappear. 2) The register pressure would increase. 3) It is not always possible to schedule operations to hide such long load latencies (ILP extraction is limited in some applications). So, when a Dcache miss occurs, the VLIW processor stalls completely if the cache is blocking. If the cache is non-blocking, the VLIW processor continues to execute other pending operations but stalls quickly because the compiler schedule loads with short latencies (the distance between the issued load and the first reader is equal to a cache hit latency). We consider this gain too slender. We propose to adjust the load latencies in order to improve the gain of the non-blocking caches.

This paper is organized as follows. Sect. 1 first presents some related work. Our target embedded processor is described in Sect. 2. Sect. 3 presents the collected simulation results of the blocking cache architecture. Sect. 4 presents the collected results on the non-blocking cache architecture. Before concluding, we also present our proposal for a compile-time pre-loading technique to generate better embedded VLIW codes in presence of a non-blocking cache.

1 Related Work

Several research on cache effects at fine grain level have been carried out. Touati included the impact of the compulsory misses in an optimal acyclic scheduling problem (6) in a single basic block. He models the exact scheduling problem by including the constraint of data dependences, functional units, registers and compulsory misses. Our current work is different because we try to cover all the kinds of misses (compulsory, capacity and conflict). Also, we do not focus in a single DAG (basic block) only, we are interested in optimising of whole application. Tien *et al.* studied the effects of non-blocking loads and the prefetch in MIPS3000 processor (7), and tried some compiler optimisation adapting loads to have more gain with the non-blocking loads. Whereas in our work, we study the cache effects for a VLIW (multiple issue) processor. We also use two phases compilation to adapt latencies to loads operations (as we will explain later). Oner *et al.* made a study of kernel scheduling over a MIPS processor (8). They increased the load-use dependency distance in loop kernel using loop pipelining. Ding *et al.* (9) made a static analysis of code to determinate which is a cache hit and which is a cache miss load instruction, this technique is called selective schedule. Abraham *et al.* (10) made a profiling of the load instructions, then the step of selecting loads which misses the cache. The final state is the prefetching of these delinquent loads. Our study is also based into profiling and analysis of trace, but we change the load-use distance rather than adding prefetch instructions. Furthermore, we aim to propose a pre-loading technique in conjunction with global scheduler (handling a whole function), and such scheduler does not necessarily target regular codes such as loop nests.

As far as we know, this is the first study demonstrating the practical effectiveness (or not) of a non-blocking cache inside an embedded VLIW processor. The next section presents our processor model.

2 ST231 Processor Description

The *ST231* (5) 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. 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:

$$T = Calc + DC + IC + InterS + Br$$

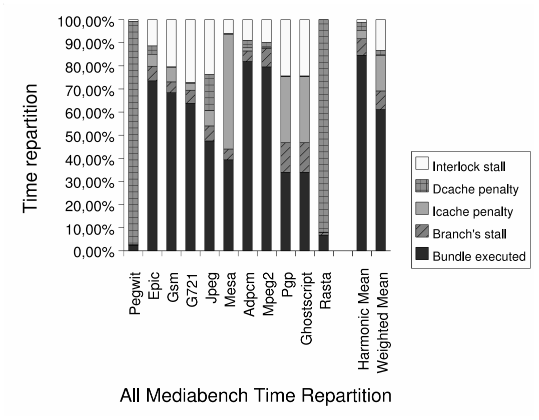

Figure 1: Mediabench Execution Time Distribution

where: 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*. Our approach does not focus on benchmark's kernel only, we want to study and improve performance of application benchmarks using full precise, but long, simulation. The next section presents our performance analysis study of *ST231* using a regular blocking cache.

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. We collect simulation results of the *mediabench* and *ffmpeg* execution.

We can observe in Fig. 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 shows that 33.34% of execution time is wasted in Dcache stalls. We calculate another parameter to quantify the Dcache misses, this parameter is the distance between a *load* operation and the first load's costumer operation. We call this distance as the *load-use distance*. There are two kinds of *load-use distances*. The first one is the load-use distance in cycles (Dynamic), which expresses the effective dynamic distance including all the stalls. The second one is the static distance set by compiler in the generated code (it is resulted from the static instruction scheduler).

Through several experiences, we observed a great difference between measured static distances and the dy-

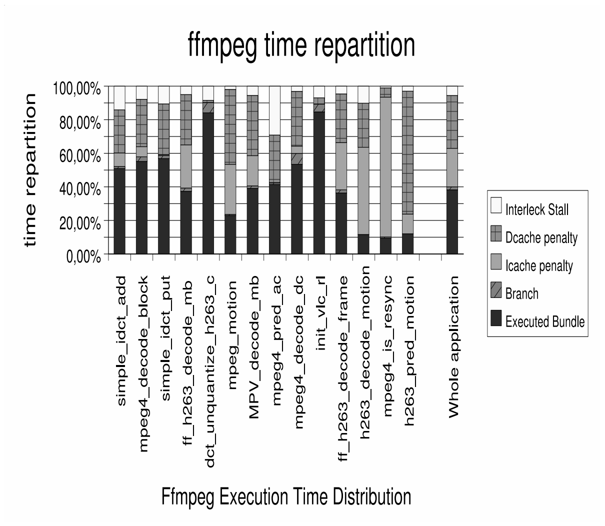

Figure 2: Ffmpeg Execution Time Distribution

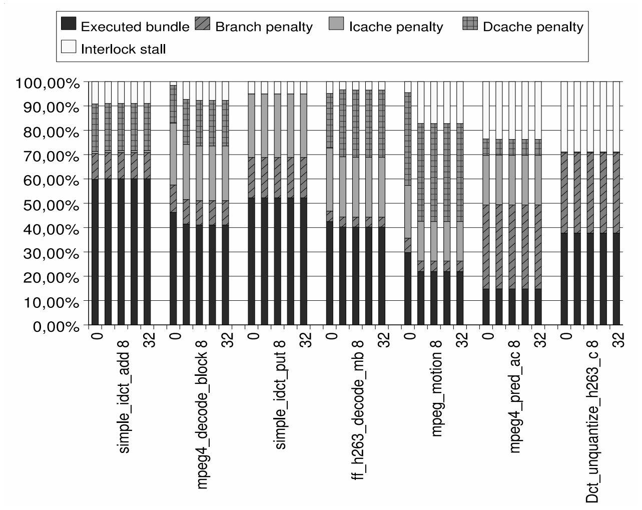

Figure 3: Execution Time Distribution of ffmpeg Different Sizes of Pending Load Queues (0, 1, 8, 16, 32)

namic ones. The mean of static distance values is around three bundles. This demonstrated that the compiler makes an optimistic load latency, it assumes that all data reside in the L1 cache (since the cache latency is 3 cycles). The dynamic load-use distances measured at the simulation time. Its mean value is around thirty cycles, which is far from the three optimistic cycles. The gap is due to Dcache misses. In the next section, we will see the contribution of a hardware mechanism (lock-up-free caches) generally designed for reducing stalls due to cache misses.

4 Non-blocking Cache Simulation Results

Kroft (11) defined the lock-up-free (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. In (1), the authors show that a non blocking cache can significantly improve the performances of an out-of-order (OoO) processor. So, many high performance OoO 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 multi-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. These cache misses are stored inside a queue (called *pending load queue*). The size of this queue, that we note *SPQ*, is a micro-architectural parameter which defines the number of concurrent loads waiting for memory service. Intuitively, when SPQ is large, more pending cache misses can be concurrent resulting in better load overlap. This section makes a precise performance evaluation resulting from adding a non-blocking cache inside an embedded VLIW processor (*ST231*). We also study the influence of SPQ, and the influence of the instruction schedules generated by the compiler.

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. The used binary codes are the same used in Sect. 3.

Fig. 3 shows execution distribution time of the *ffmpeg* benchmark. We made distinct simulations, changing each time the size of the pending load queue size (SPQ) 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 non-blocking cache.

For a pending load size equal to zero, Fig. 3 shows that the results are similar to the simulation results obtained with the blocking cache simulator (in Sect. 3, Fig. 2). The surprise is that there is a negligible performance improvement whatever the SPQ size. The performance improvement is 1.62% for the whole *ffmpeg* application with a SPQ equal to one! The result is similar in case of *mediabench* applications. Contrary to OoO processors, introducing a non-blocking cache in a VLIW in-order processor does not provide a performance gain, unless the codes are recompiled with some special instruction scheduling techniques (shown later).

Another result is shown in Fig. 3: when the SPQ is changed from 1 to 8, 16 and 32, we obtain the same performance gain 1.66% for the whole application. Contrary to OoO processors, increasing the SPQ size has little impact (unless we re-optimise the VLIW code as we will show later).

The *mediabench* benchmark simulation gives similar results, *i.e.* a weak performance improvement. this is shown in Fig. 4.

All the observed small speed-ups are due only to Dcache stall reduction. When considering exactly the same binary codes as in Sect. 3, executing them on the same VLIW processor but with changing the blocking cache non a non blocking one seems to do not alter other dynamic performance metrics: Icache stalls, branch penalties ans 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 pegwit application and the worst one is less than 0.1% in MPEG2.

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.

4. We do not observe any speed-down due to the non-blocking cache.

5. The negligible speed-up is observed as maximal with a pending load queue size of 8 entries only.

All the negligible speed-ups obtained with a non-blocking cache architecture are disappointing but can be explained: when a Dcache miss occurs, the processor does not stall, it still execute the next bundle (VLIW) thanks to the non-blocking cache opportunity. However, the consumer of the loaded data is too close (three bundles later). Thus, the processor stalls too early and the benefit of the non-blocking cache is limited. We believe that the in-order architecture can better exploit the non-blocking cache architecture as well as the out of order does. However, the binary codes must be adapted to take in consideration the cache model. To avoid the poor performance improvement of the non-blocking cache architecture, we propose to reschedule the instructions by increasing the static load-use distance. We change the load instruction latency, and adapt its to each code. We must calculate for each load instruction the most suitable latency whatever it hits or misses the Dcache. For the loads that hit the cache, we do not need to change their latency.

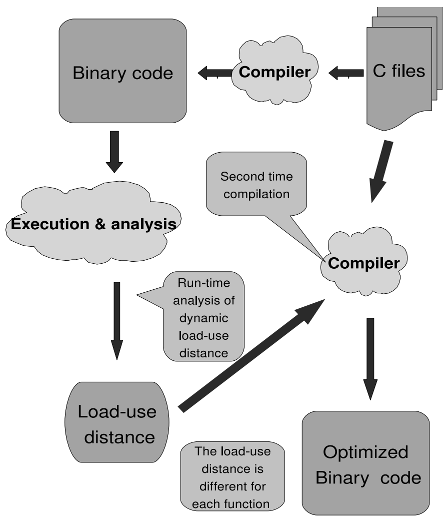

Figure 5: the used methodology

For the other loads, the static latency must be changed. For instance, the latency of the delinquents loads must be adapted.

In this section, we consider load-use intervals, where: load-use interval = [cycle of load , cycle of the user]. Thanks to the non-blocking cache, load-use intervals may overlap.

In order to compute a new metric each load capturing load-use intervals overlap, we propose the next formula:

$$NormalizedDistance = \left\lceil \frac{C_2 - C_1}{L} \right\rceil \tag{1}$$

where C_1 and C_2 are cycles when load or consumer load instruction occurs and L is the number of overlapped loads.

This *normalized distance* is our new metric that we use as a parameter to a new static compiler optimisation option. It sets a new static load latency in whole function scope: the normalized distance represents the number of additional static cycles to a cache hit latency. That is, a load that was initially scheduled with a cache hit latency by the compiler, becomes scheduled with a new latency equal to the cache hit plus the normalized distance. The overlap parameter L in Equ. 1 shows that when L is high then the normalized distance tends to zero. So, the static distance tends to a cache hit latency. This means that the compiler does not change its initial latency because a good machine usage (sufficient overlapped memory requests). If the value of L is low, the static load distance is increased to allow more pending loads to be executed in parallel during cache miss.

We have calculated this new normalized distance metric for all the *mediabench* and also for *ffmpeg* functions. We also experimented some regular usual codes such as, matrix-vector multiplication and matrix-matrix multiplication.

To decrease the Dcache stall penalty by adapting the static loads latencies. We use an on-the-fly trace analyser

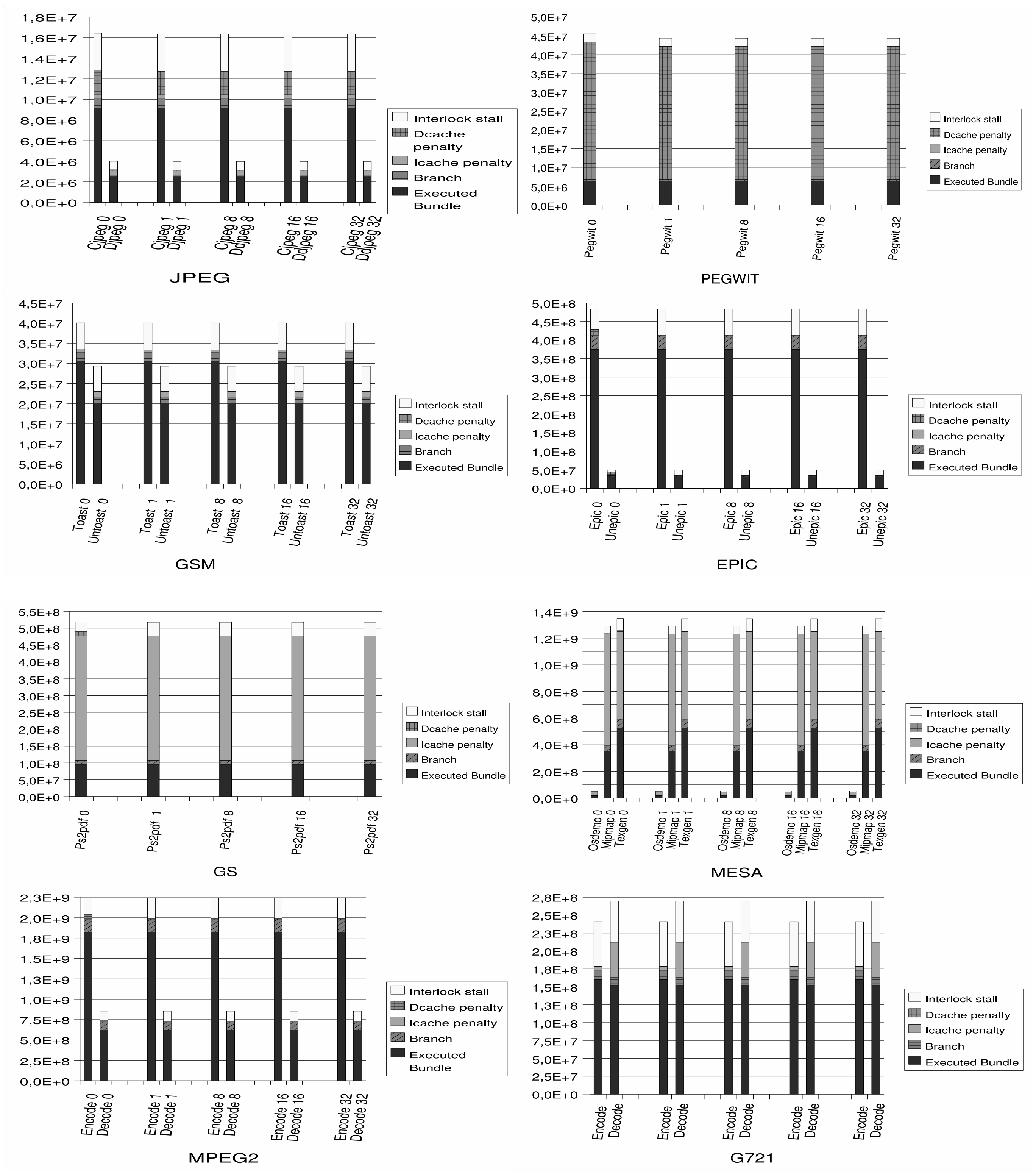

Figure 4: Execution Time Distribution of Mediabench with Distinct Pending Load Queue Sizes (0, 1, 8, 16, 32)

during the simulation to calculate the dynamic load-use intervals, and the number overlapped loads at each processor clock cycle. With Equ. 1, we manage to compute the new static load distance that we should apply in a re-compilation process. For this purpose, the regular optimising compiler of STmicroelectronics has been modified by the vendor to allow us such low level code optimisation. A new compiler version has been designed for our study. We can now adjust load latency by a special compiler option. Simply increasing the static load latencies without careful attention may produce many impacts in the final generated code:

Fig. 5 shows all steps of our methodology to decrease the data cache stall penalty by adapting the static loads latencies. We use an on-the-fly trace analyser during the simulation to calculate the dynamic load-use distances each processor clock cycle. With Equ. 1, we manage to compute the adapted load distance that should be applied in a re-compilation process. For this purpose, the regular optimising compiler of STmicroelectronics has been modified by the vendor to allow us such low level code optimisation. A new compiler version has been designed for our study. We can now adjust load latency by a special compiler option.

1. When instruction rescheduling, the code size may increase and consequently may have negative effects on instruction cache misses. So for some short loop, we force compiler to unroll it rather than pipeline it. In case of pipelined loop, increasing load latency can increase the II, however, in some case greater II can gives better performances than smaller one (due to cache effects which are not considered at scheduling time).

2. For the non-loop code, if the new latencies are too long, the compiler may not find enough ILP. To avoid that, several methods can be applied as tail duplication, region scheduling, Super-block instruction scheduling, trace scheduling, scheduling non-loop code with prologue/epilogue of loop blocks.

3. Increasing load latency increases the register pressure; the compiler can introduce spill code to reduce simultaneously alive variables. So, when scheduling, we must take care about register pressure.

Our normalized distance as proposed in Equ. 1 aims to reduce the negative impact described above. Furthermore, we should be aware that modifying a load latency may considerably modify the cache effects of a code: Since load operations are reschedule, some initial cache misses may become hits and *vice-versa*, because of the instruction rescheduling that modifies the spacial/temporal locality of the code. In order to guarantee that the cache effects stay the same before and after static load modification, we impose to the compiler (via a special pragma) to keep the same order for the loads before and after latency modification. Consequently, applying a new latency to loads does not modify the relative order between the loads, keeping the same cache effects (and thus, our normalised distances computed via an initial program simulation remains valid).

Also, we can see that the distance decreases while the size of the pending load queue increases until a limit where increasing the size of the pending load has no effect on the normalised. Not all functions are candidate for our code optimisation methodology. We consider the function that has two properties : 1) a considerable fraction of Dcache stall in the execution time of the function, and 2) the normalised distance should be larger that the cache hit latency (3 cycles). When we look for these two parameters (Dcache fraction plus considerable normalised calculated distance), we find that few functions in *ffmpeg* and *mediabench* are candidate to our optimisation. For ffmpeg application, we obtained 28.28% whole application speed-up using adapted loads latencies.

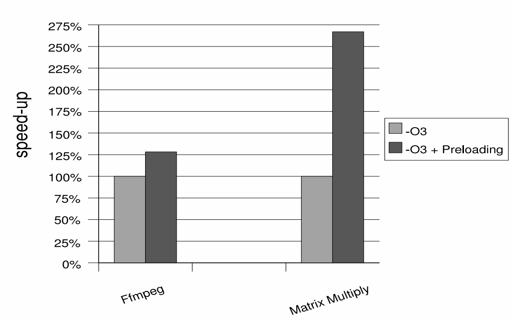

Figure 6: Original vs Optimized ffmpeg & Matrix-Matrix Multiplication Benchmarks Results

We can now apply our optimisation method in case of more well known benchmark such as square Matrix-Matrix (512*512) multiplication: we use a non-naive implementation, produced by ATLAS ("best" loop tiling, each tile contains 64 * 64 elements = 16KB which are kept by the Dcache). The obtained results are promising and conclusive. In Fig. 6, we can observe the positive effects of using the normalized distances. The main advantage of this optimisation is that it can be applied to any control flow graph, not necessary to loops. Finally the speed-up obtained thanks to Preloading and non blocking cache hides the cost of the added hardware (non-blocking cache)

5 Conclusion

Our study was based on precise full simulation of whole embedded applications (*mediabench* and *ffmpeg*). Our experimental study consumed many months of full simulation on tens of workstations producing tera-bytes of data to collect and analyse. We precisely measured the impact of a non-blocking cache inside a VLIW embedded processor (*ST231*) compared to a blocking cache architecture. As shown in our experimental results, if the binary codes are not modified, the performance improvement is poor (program acceleration less than 3% in the best case!). This situation has a concrete explanation. Many current compilers schedule load instruction too close to their consumers: this is a common heuristics to decrease the register pressure. Such scheduling heuristics assume that data reside in L1 cache, and consequently loads are scheduled as cache hits. Such scheduling heuristics reduces the benefit of a non-blocking cache in case of in-order and VLIW embedded processors. This situation is not altered when increasing the size of the hardware pending queue associated to the non-blocking cache.

Our experimental results are in opposition with the case of high performance out-of-order processors, where non-blocking caches provide positive effects in execution performance without changing program binaries. High performance out-of-order processors contain much more hardware mechanisms (resulting in higher costs) that allow program acceleration without instruction rescheduling at compile time. In the case of an embedded VLIW processor, we showed that if the code is not re-optimised in order to take into account the non-blocking cache, the benefit is negligible.

Our code optimisation methodology is based on data pre-loading. Our method performs in two steps. The first step computes normalised load-use distances using execution trace analysis. Then, this distance is used in the second step to reschedule the code at instruction level, while keeping the same loads order as the one analysed in the first step. Keeping the same loads order in the second step guarantees that the cache effects analysed in the first step are not altered in the second step. Our results on matrix-matrix multiply and ffmpeg show respectively a speed-up of 267.61% and 28.28% for the whole program execution. This provides us promising demonstration of the effectiveness of our ideas. In the future, we will combine pre-loading with data prefectching in order to optimise memory requests for both regular and irregular embedded VLIW codes.

REFERENCES

[1] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach,* Morgan Kaufman, CA, 1996.

[2] Michael E. Wolf and Monica S. Lam. *A Data Locality Optimizing Algorithm* . PLDI'91, pages 30 – 44, 1991.

[3] Randy Allen and Ken Kennedy.*Optimizing Compilers for Modern Architectures*. Morgan and Kaufman, 2002.

[4] James Edward Sicolo. *A Multiported Nonblocking Cache for a Superscalar Uniprocessor* B.S. State University of New York at Buffalo, 1989.

[5] STMicroelectronics ADCS 7645929F ST231 Core and Instruction Set Architecture Manual, 2005.

[6] S.-A.-A. Touati. *Optimal Acyclic FineGrain Scheduling with Cache Effects for Embedded and Real Time Systems*. ACM Proceedings of the Ninth International Symposium on Hardware/Software Codesign. Copenhagen, Denmark, April 25-27, 2001, IEEE.

[7] Tien-Fu Chen and Jean-Loup Baer. *Reducing Memory Latency via Non-blocking and Prefetching Caches*. Proceedings of the fifth international conference on Architectural support for programming languages and operating systems. Boston, Massachusetts, United States. 1992.

[8] Koray Öner and Michel Dubois.*Effects of Memory Latencies on Non-Blocking Processor Cache Architecture*. Proceedings of the 7th international conference on Supercomputing ICS. 1993.

[9] Chen Ding and Steve Carr and Phil Sweany.*Modulo Scheduling with Cache Reuse Information*. European Conference on Parallel Processing. 1997.

[10] Abraham Sugumar, Windheiser, Rau, Gupta. *Predictability of Load/Store Instruction Latencies*. Proceedings of the 26th annual international symposium on Microarchitecture. Austin Texas. 1993.

[11] David Kroft. *Lockup-free Instruction Fetch/Prefetch Cache Organization* . Proc. 8th International Symposium on Computer Architecture, Minneapolis, MN, May 1981, p. 81-85.

Acknowledgements

This research result has been supported by the ANR MOPUCE project (ANR number 05-JCJC-0039) and the French Ministry of Industry. We would like to greatly thank Francesco PAPARIELLO and Giuseppe DESOLI from STMicroelectronics-Milano for their valuable effort in implementing the non-blocking cache simulator. We would also like to greatly thank Benoit DUPONT-DE-DINECHIN and Christophe GUILLON from STMicroelectronics-Grenoble for their valuable effort in implementing our pre-loading technique in the ST VLIW compiler.