On Instruction-Level Method for Reducing Cache Penalties in Embedded VLIW Processors

Samir AMMENOUCHE, Sid Ahmed Ali TOUATI, William JALBY
University of Versailles Saint-Quentin en Yvelines, France

Abstract

Usual cache optimisation techniques for high performance computing are difficult to apply in embedded VLIW applications. First, embedded applications are not always well structured, and few regular loop nests exist. Real world applications in embedded computing contain hot loops with pointers, indirect arrays accesses, function calls, indirect function calls, non constant stride accesses, etc. Consequently, loop transformations [12] for reducing cache misses are impossible to apply, especially at the back-end level. Second, the strides of memory accesses do not appear to be constant at source code level, because of indirect accesses. Hence, usual prefetching techniques are not applicable. Third, embedded VLIW processors are "cheap" products, they have limited hardware dynamic mechanisms compared to high performance processors [8]: no out-of-order execution, reduced memory hierarchies, small direct mapped caches, lower clock frequencies, etc. Consequently, the code optimisations methods must be simple and take care of code size. This article presents a back-end code optimisation for tolerating non-blocking cache effects at the instruction level (not at the loop level). Our method is based on a robust combination of memory pre-loading with data prefetching, allowing us to optimise both regular and irregular applications at the assembly level. Our experiments with mediabench and SPEC2000 benchmarks suites on the ST231 VLIW processor show a positive performance gain (compared to codes generated with -O3 compiler optimisation flag). Our method induces negligible code size growth (less than 3.9 % in the extreme case).

Keywords Data prefetching, pre-loading, non-blocking cache, embedded systems, VLIW.

1. Introduction

Program transformations for reducing cache penalties is a well established research area in high performance computing and desktop applications. Nowadays high performance processors offer many hardware mechanisms helping either to hide or to tolerate memory latencies[8]: multiple cache levels, higher cache sizes and degrees of associativity, memory banking and interleaving, non-blocking caches and out-of-order execution, etc. All these hardware mechanisms combined with program transformations at the loop nest level produce positive speed-ups, in general.

In addition to a better harmony between hardware and software, cache optimisation has been also introduced at the operating system (OS) level. Thanks to multitasking combined with multicore architectures, we can now envisage methods where an independent parallel thread or OS service can prefetch application data. The OS can also detect some situations when dynamic re-compilation during execution is necessary to generate better codes regarding cache miss penalties.

Consequently, nowadays cache optimisation strategies for high performance and desktop applications require more and more the conjunction between multiple complex techniques at various levels: application (loop nest or CFG), operating system and hardware (processor and memory).

The case of embedded applications is quite different. First, an embedded VLIW processor is at least hundred times cheaper than a high performance processor: few hardware mechanisms for cache optimisation exist (if any); the computation power is also reduced, there is a little margin to tolerate code optimisation based on aggressive speculation. Second, some embedded systems execute with a light OS, or even at bare mode (without any OS): no dynamic services or tasks can be used in parallel to improve cache effects. Third, embedded applications are rarely statically controlled programs with regular control or regular data accesses: such applications cannot meet the model requirements for loop transformations [12] and for usual software prefetching with regular strides. Fourth and last, code size growth is an additional constraint to deal with.

In this article, we present our method to reduce processor stalls due to cache misses in presence of non-blocking cache architectures. We implement our method at the back-end level where loop structures disappear. Our principal aim is

not to reduce cache misses (as usually done with loop transformations) but to reduce the processor stalls due to them. It is a combination of software data prefetching (inserting special prefetch instructions) with pre-loading (increasing static load latencies) as studied in [13]. As we will explain later, it is especially designed for future VLIW in-order processor that would include non-blocking caches instead of blocking caches.

Our article is organised as follows. Sect. 2 presents some interesting related work on reducing cache miss penalties. Sect. 3 presents the problem of cache effects in instruction level parallelism (ILP). Sect. 4 presents the practical ST231 VLIW processor that we use as a target example in this study. Sect. 5 presents our methodology for pre-loading and prefetching in the compiler back-end. Before concluding, we demonstrate in Sect. 6 the efficiency of our code optimisation method on applications from mediabench and SPEC2000 benchmark suites executed on ST231.

2 Related Work

Improving the cache effects at instruction level is a well studied topic. We can classify related work following two directions: a theoretical one, where some studies were done on instruction level scheduling taking into account the cache constraints. The second direction is more practical. As a theoretical work we quote Touati who included the impact of the compulsory misses in an optimal acyclic scheduling problem [15] 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 kinds of cache misses (compulsory, capacity and conflict). Also, we do not restrict ourselves to a single DAG (basic block) only, we are interested in optimising an application as a whole.

We are interested here on practical ways which treat minimising cache miss penalties with two techniques: prefetch and instruction scheduling techniques. Using the prefetch solution, Al-Sukhni *et al.* [2] classified the load operations as *intrinsic* and *extrinsic* streams and developed a prefetch algorithm based on automaton taking into account the density and the affinity of these streams. The experiments were done on a simulator of a superscalar out-of-order processor (freescale): out-of-order execution helps hiding cache miss penalties at execution time, in opposition to our case which is an in-order VLIW processor. Abraham *et al.* [14] proposed a prefetch technique. They described their technique by automaton: the first step of this automaton is profiling of load instructions, the second one is the selection phase of loads that miss the cache. The final state is the prefetching of these delinquent loads. Another prefetch solution is dynamic prefetching as proposed by Beyler *et al.* [3]. They studied a dynamic prefetch mechanism using the load latency variation to classify the loads. The framework is based on finite state machine. They obtained positive results on Itanium processor where the Intel compiler (icc) automatically generates prefetch instructions. Always on dynamic prefetching, we quote Lu *et al.* [7] who developed a framework called ADORE. They proceed on three steps: tracking delinquent loads, selecting the data references and finally prefetching these loads. This solution is based on hardware monitor of the Itanium processor. The two previous work [3] and [7] were done on Itanium architecture which is used for high performance computing. Our work is done on a *light* embedded VLIW processor which generally executes a single task; so, the dynamic prefetch mechanism is an inappropriate solution for our kind of architecture.

Our work can target two cache architectures: a blocking cache architecture and a non-blocking one. In case of blocking cache architectures, only the prefetch method is used in our case. If non-blocking cache is present, prefetch is also used combined with pre-loading (as explained later). This later case is more interesting because future VLIW processor would include non-blocking caches. Blocking cache architecture and optimisations were treated in many studies. Tien *et al.* [4] studied the effects of pipelined loads and prefetch in MIPS3000 single issue processor, and tried some compiler optimisations such as changing static load latencies to exploit the pipelined execution of loads. Whereas in our work, we study the cache effects for a VLIW (multiple issue) processor.

For a non-blocking cache architecture, Oner *et al.* [9] made a study of kernel scheduling on a MIPS processor. The authors increased the load-use dependency distance in loop kernel using loop pipelining. In addition to the kernels, our method is applied on basic blocks, functions and whole applications. In other words, we have no code granularity restrictions.

Ding *et al.* [5] based their work on reuse information *i.e.* they made a first step static analysis to collect load statistics of selected kernels. Then, they used the collected statistics to combine data prefetching and instruction scheduling techniques to hide cache effects. Contrary to the work of Ding *et al.*, we do not restrict ourselves to loops and we do not use a virtual superscalar machine. Our target architecture is a real VLIW in the market (used in many embedded systems).

The authors in [6] did a performance evaluation to study the hardware complexity of non the blocking cache architecture using SPEC92 benchmarks. They showed that a simple hit-under-miss non-blocking cache implementation (*i.e.* only two overlapped loads at the same time) is, the best trade-off between hardware cost and performance. However, recent work done by Ammenouche *et al* [13] showed that non-blocking caches do not provide any performance

improvement in the case of embedded VLIW processors, because execution is in-order and no dynamic instruction scheduling is done to hide cache miss penalties as in the case of superscalar processors. However, Ammenouche *et al* showed on two applications that non-blocking caches may provide good performance improvement if low-level code optimisation based on pre-loading is used. Our current article confirms this fact, and extends the previous work by adding a prefetch method and making a more complete experimental study using mediabench and SPEC benchmarks.

To clearly explain the position of our contribution in the current literature, we say that our study aims to improve (at the software level) the efficiency of the non-blocking cache architecture on VLIW processors. We combine data prefetching and pre-loading in conjunction with a global scheduler that handles a complete function. Such global scheduler does not necessarily target regular codes such as loop nests. As we will explain later, our framework is based on profiling and trace analysis.

3. Problems of Optimising Cache Effects at the Instruction Level

Nowadays cache memory is widely used in high performance computing. It is generally organised in a hierarchical way making a trade-off between cost and performance. The drawback of this memory architecture is the unpredictability of the data location. Indeed, at any time during the program execution, we are uncertain about the data location: data may be located in any cache level, or in the main memory or in other buffers. This situation can be acceptable in high performance architecture, but cannot be appreciated in embedded *soft* real time systems because data access latencies are unpredictable. We focus our work on embedded systems, especially VLIW processors. In this case, one of the most important aspects is the instruction scheduling. A static scheduling method considering a cache model would be ideal to hide/tolerate the unpredictability of execution times. Nowadays, general purpose compilers like gcc, icc and the st200cc (the compiler provided for the VLIW ST2xx architecture) do not manage the cache effects: memory access latencies are considered fixed during compilation because the latencies of the load instructions are unknown statically. Many instruction scheduling techniques are developed and have been commented upon the literature, but they always suppose well defined latencies for all kinds of instructions. The fact is that the proposed models are simplified because of the lack of knowledge about data location and thus about load latencies.

Loop scheduling is a good example to assert our idea: software pipelining is a well-matured scheduling technique for innermost loops. It's aim is usually to minimise the Initiation Interval *II* and the prologue/epilogue length. The compiler assumes that the total execution time of the pipelined loop is the sum of the prologue and epilogue length and the kernel (*II*) multiplied by the number of iterations. Since almost all scheduling techniques assume fixed instructions latencies, the compiler has an artificial performance model for code optimisation. Furthermore, the compilers above quoted schedule the load instructions with optimistic latencies, since they assume that all data reside in lower cache levels, and they schedule the consumer of the loaded data close to the load operation. Consequently, low level instruction schedulers of compilers have optimistic view of the performance of their fine-grain scheduling. The case of the st200cc is relevant, this compiler schedules the consumers of a load only 3 cycles after the load (3 corresponds to the cache hit latency, while a cache miss costs 143 clock cycles). If a load misses the cache, the processor stalls for at least 140 cycles, since a VLIW processor has no out-of-order mechanism. The icc compiler for Itanium has also the same behaviour and schedule all loads with a fixed latency (7 cycles), a latency between the L2 (5 cycles) and L3 (13 cycles) levels of cache.

Another problem of instruction scheduling taking into account cache effects is the difficulty to precisely predict the misses in the front-end of the compiler. While some cache optimisation techniques are applied on some special loop constructs, it is hard for the compiler front-end to determine the cache influence on fine-grain scheduling and vice-versa. Sometimes, this fact makes compiler designers implement cache optimisation techniques in the back-end where the underlying target architecture is precisely known (cache size, cache latencies, memory hierarchy, cache configuration, other available buffers). However, in the compiler back-end, the high level program is already transformed to a low level intermediate representation and high level constructs such as loops and arrays disappear. Consequently, loop nest transformations can no longer be applied to reduce the number of cache misses. Our question becomes how to hide the miss effect rather than how to avoid the miss.

Another important criterion for applying cache optimisations at different levels is the regularity of the program. At compilation step, regularity can be seen on two orthogonal axis: regularity of control and regularity of data access. Due to the orthogonality of these two axis, four scenarios are possible:

1. Regular code with regular data access: Data prefetch can be used in this case, for instance to prefetch regular array accesses. For instance:
   `while(i ≤ max) a+=T[i++];`
2. Regular code with irregular data access: Depending on the shape of irregularity, data can sometimes be prefetched. Another possible solution is the pre-

loading (explained later in Sect. 5.2). For instance:
`while(i ≤ max) a+=T[V[i++]];`

3. Irregular code with regular data access: The data prefetching solution is possible, but inserting the prefetch code has to take care of multiple execution paths. For instance:
`while(i ≤ max) {if(cond) a+=T[i++]}`

4. Irregular code with irregular data access: also depending on the shape of irregularity data can sometimes be prefetched. The pre-loading (explained later in Sect. 5.2) is more suitable in this case. For instance:
`while(i ≤ max){ if(cond) a+=T[V[i++]]}`

Note that while data prefetching usually requires some regularity in data access, pre-loading can always be applied at the instruction level.

4 Target Processor Description

In our study, we use the *ST231* core [1] which is currently the latest processor of the *ST2xx* family from STmicroelectronics. These VLIW processors implement a single cluster derivative of the Lx architecture [11], and are used in several successful consumer electronics products, including DVD recorders, set-top boxes, and printers. At the end of 2008, the number of shipped processor was over 33 million units. ST231 is an integer 32 bits VLIW processor with five stages in the pipeline. It contains four integer units, two multiplication units and one load/store unit. It has a 64 KB L1 cache. The latency of the L1 cache is 3 cycles. The data cache is 4 way associative. It operates with write-back no-allocate policy. A 128 bytes write buffer is associated with the Dcache. It also includes a separated 128bytes prefetch buffer which can store up to eight cache lines. As for many embedded processors, the power consumption should be low, hence limiting the amount of additional hardware mechanisms devoted to program acceleration. In addition, the price of this processor is very cheap compared to high performance processors: a typical high performance processor costs more than one hundred times compare to the ST231.

Regarding the memory cache architecture, the current marketed ST231 includes a blocking cache architecture. In [8], the non-blocking cache is presented as a possible solution for performance improvement in Out-Of-Order (OoO) processors. So, several high performance OoO processors use this cache architecture. The interesting aspect of this cache architecture is the ability to overlap the execution and the long memory data access (loads). Thanks to non-blocking cache, when a cache miss occurs, the processor continues the execution of independent operations. This produces an overlap between bringing up the data from memory and the execution of independent instructions. However, the current embedded processors do not include yet this kind of memory cache because the ratio between its cost (in terms of energy consumption and price), and its benefit in terms of performance improvement was not demonstrated till the recent results of [13]. Furthermore, in order to efficiently exploit the non-blocking cache mechanism, the main memory must be fully pipelined and multi-ported while these architectural enhancements are not necessary in case of blocking cache, Kroft [10] proposed a scheme with special registers called MSHR (Miss information Status Hold Registers), also called pending load queue. MSHR are used to hold the information about the outstanding misses. He defines the notion of *primary* and *secondary* miss. The primary miss is the first pending miss requesting a cache line. All other pending loads requesting the same cache line are secondary misses - these can be seen as cache hits in a blocking cache architecture. The number of MSHR (pending load queue size) is the upper limit of the outstanding misses that can be overlapped in the pipeline. If a processor has n MSHRS, then the non-blocking cache can service n concurrent overlapped loads. When a cache miss occurs, the set of MSHRS is checked to detect if there is a pending miss to the same cache line. If there is no pending miss to the same cache line the current miss is set as a primary miss and if there is an available free MSHR, the targeted register is stored. If there is no available free MSHR, the processor stalls.

5. Our Methodology of Instruction-Level Code Optimisation

Our method aims to hide the cache penalties (processor stalls) due to cache misses. We want to maximise the overlap between the stalls due to Dcache misses with the processor execution. For this purpose, we focus our study on delinquent loads, wherever the delinquent loads occur in loops or in other parts of code. We do not limit our study to a certain shape of code, we consider both regular and irregular control flow and data streams. We study two techniques, each of them corresponds to a certain case:

- For the case of irregular data memory accesses, we use the pre-loading technique.
- For the case of regular data memory accesses, we use the prefetch technique.

It's well known that combining many optimisation techniques doesn't lead to better performances. This fact is due to the sensibility of these optimisations techniques one over the other. This leads to a hard phase ordering problem. Our methodology solves this problem for the two combined optimisations. Since these two techniques are complementary,

we can also combine them in the same function. Let us explain in detail the usage of these two techniques.

5.1. Low Level Data Prefetching Method

The cache penalty is very expensive in terms of clock cycles (more than 140 cycles in the case of the ST231). The current hardware mechanisms fail to fully hide such long penalty. In the case of a superscalar processor as the Intel Pentium, the out of order mechanism can partially hide the cache effects during few cycles (up to the size of a window of instructions in the pipeline). Rescheduling the instructions, with a software (compilation) method or hardware technique (execution) cannot totally hide the cache penalty.

The prefetching technique is an efficient way to hide the cache penalty. However, usual prefetching methods work well for regular data accesses that are analysed at source code level. In our embedded applications, data accesses do not appear to have regular strides when analysed by the compiler because of indirect access for instance. Furthermore, the memory access is not always inside a static control loop. Consequently, usual prefetching techniques fail. In our method, we analyse the regularity of a stride thanks to a precise profiling.

Our data prefetching is based on predicting the addresses of the next memory access. If the prediction is correct the memory access will be costless. In the case of bad prediction, the penalty is low (the *ST231* include a prefetch buffer, so the *bad* prefetched data does not pollute the cache). The only possible penalty consists of adding extra instructions in the code (code size growth) and executing them. However, in case of VLIW, we can take care of inserting these extra instructions inside free slots because not all the bundles contain memory operations. Consequently no extra cost is added, neither in terms of code size nor in terms of executions. So, the most important aspect with this technique is the memory address predictor, or how to generate a code that computes the address of the next prefetched data.

Our method of prefetching requires the process of three phases: profiling the code to generate a trace, then selecting some delinquents loads and finally inserting the prefetch instructions.

5.1.1 Application Profiling Phase

This step is the most expensive in terms of processing time, because we have to perform a precise profiling of the code by generating a trace. Classical profiling, as done with *gprof* for instance, operates at semi-coarse grain level (functions). In our case, we proceed in the finest profiling granularity, that is at the instruction level. To do this, we use a special software plugin device, which can manage the execution events and statistics. This plugin interfaces with the simulator which is completely programmable. We use the plugin to select all the loads which miss the cache, and for each load, collect its accessed addresses inside a trace. This trace highlights the delinquents loads. A load is said to be delinquent, if it produces a large number of cache misses. In practice, we sort the loads according to the number of cache misses they produce, and we defined the top ones as delinquents. We perform an on-line identification of these delinquent loads using the plugin explained above. The result of this profiling phase is a precise cartography of the accessed memory data addresses, tagged with the delinquent loads. The next step of our work is to select the right loads to prefetch within the set of delinquent loads.

5.1.2 Load Selection Phase

Selecting which delinquent loads to prefetch depends on two parameters: the number of cache misses and the regularity of memory accesses. The most important criterion is the number of misses. Indeed, in order to maximise the prefetch benefit, it is important to prefetch loads with a high frequency of cache misses. Choosing loads which produce many cache misses allows to hide the cost of extra prefetch instructions: prefetch instructions may introduce some additional bundles in the original code. Increasing the code size or changing the code shape, may produce very undesirable effects and may slowdown the performance because of the direct mapped structure of instruction cache. Consequently, for a given identified delinquent load, the higher number of misses we get, the better performance we can achieve. We do not care about the ratio of hit/miss of the delinquent load, we just measure the frequency of cache misses and sort the loads according to this value.

Once a delinquent load is selected as a good candidate for prefetching, we should analyse the second parameter, which is the memory access regularity. The authors in [16] classify the load with the next data stride patterns:

- Strong single stride: It is a load with a near constant stride *i.e.* the stride occurs with a very high probability.
- Phased multi-stride: It is a load with many possible strides that occur frequently together.
- Weak single stride: It is a load with only one of the non-zero stride values that occurs somewhat frequently.

Once we select delinquent loads with strong single stride or with phased multi-stride, we can proceed to the last step of prefetch instruction insertion.

5.1.3 Prefetch Instruction Insertion Phase

This step consists of adding a single or many prefetch instructions in the code. The syntax of a load instruction

on the *ST231* is: `LD Rx= immediate[Ry]`. The first argument of the instruction is `Rx` the destination register, while the second argument is the memory address defined as the content of the index register `Ry` plus an immediate offset. The prefetch instruction has the same syntax `PFT immediate[Ry]` except that it does not require a destination register. Executing a prefetch instruction brings data to the prefetch buffer and does not induce any data dependence on a register. However, we should take care of not adding an extra cost of the added prefetch instruction. In order to achieve this purpose, the prefetch instruction should be inserted inside a free memory slot inside a VLIW (each bundle may contain up to one memory access instruction). If no free slot is available, we could insert a new bundle but with the risk of increasing the code size and altering the execution time (making the critical path longer in a loop, disturb the instruction cache behaviour, etc.).

Now, let us give more details on the inserted prefetch instruction. If the delinquent load has this form `LD Rx= immediate[Ry]` and has a single stride s, then we insert a prefetch instruction of the form `PFT s[Ry]`. If the delinquent load has multiple strides $s_1, s_2, \ldots$, then we insert a prefetch instruction for each stride. However our experiments hint us that it is not efficient to prefetch more than two distinct strides. The left column of Tab. 1 shows an example of prefetching with a data stride equal to 540 bytes. The bundle following the load includes the prefetch instruction: it prefetches the data for the next instance of the load.

Now, if the used index register `Ry` is altered/modified by the code after the delinquent load, this index register cannot be used as base address for the prefetch instruction. We provide two solutions:

- Use `Rz` another free register (if available) to perform the prefetch. A copy operation `Rz=Ry` is inserted just before `Ry` modification. In almost all cases we found free slots to schedule such additional copy operations, but it is not always possible to find a free register.

- If no free register exists, then we insert a new VLIW bundle that contains the prefetch instruction. This new bundle is inserted between the delinquent load bundle and the bundle that modifies `Ry`.

The right column of Tab. 1 shows an example. Here, the base register `$r27` is changed in the bundle after the load. The register `$r27` is saved on a free register, say `$r62`. Then the prefetch instruction is inserted in a free load slot.

As mentioned before, the prefetch technique is an efficient low level code optimisation that reduces the frequency of cache misses. Its main weakness is the difficulty to make an efficient address predictor. It is especially hard to predict the right addresses to prefetch in irregular data accesses. For this case, the prefetch technique cannot be applied. Thus, we propose in the next section the pre-loading technique which can be applied for the case of irregular data access.

5.2 Our Pre-Loading Method

The pre-loading technique is used if the processor includes a non-blocking cache. The authors [13] performed experiments to check the efficiency of non-blocking cache architectures on In-Order processors (such as VLIW). Their results can be summarised in four points:

1. If the code is not transformed by the compiler (recompiled for considering the new cache architecture), replacing a blocking cache architecture with a non-blocking one does not bring benefit.

2. No slowdown was noticed due to non-blocking cache.

3. If pre-loading is used (to be explained later), then a performance gain is observed.

4. A maximal performance gain was observed with 8 MSHRs.

In high performance OoO processors, replacing a blocking cache with a non-blocking cache provides speed-up even if the binary code is not optimised for. In the case of VLIW In-Order processors, the benefit of non-blocking caches is zero if the code is not modified. In order to understand this fact we need to introduce the two following definitions:

- **Definition of Static Load-Use Distance:** Static load-use distance is the distance in the assembly code (in terms of VLIW bundles) between a load instruction and the first consumer of the loaded data. This static distance is equivalent to a static measure of clock cycles between a load and its first consumption.

- **Definition of Dynamic Load-Use Distance:** Dynamic load-use distance is the distance in terms of processor clock cycles between the execution time of a load instruction and the execution time of the first consumer of this loaded data.

In [13], we showed that the static load-use distance in the set of experimented benchmarks is short, about 3 bundles, *i.e.* the `st200cc` compiler has an optimistic compilation strategy regarding load latencies. It assumes that all data reside in the L1 cache. The VLIW compiler schedules the consumer of a data too close to its producer (load) in order to keep the register pressure low. In the case of an In-Order processor with non-blocking cache architecture, it would be ideal if the compiler could generate codes with longer load-use distances. The problem is to compute the right latency for each load *i.e.* to consider the delinquent loads with higher latencies during instruction scheduling. This method

```
L?_3_69:
.
   ldw $r32=28[$r15]
   ;;
   cmpeq $b5=$r32,$r0
   pft 540[$r15]
   ;;
   brf $b5, L?_3_69:
```

Simple Prefetch with a Stride of 540 byte

```
L?_BB37_14:
.
   ldw $r28=16[$r27]
   mov $r62 = $r27
   ;;
   sub $r27=$r27,$r21
   ;;
   pft 32[$r62]
   ;;
   brf $b4, L?_BB37_14
```

Using $r62 Register to Save the Address to Prefetch

Table 1. Examples of Prefetch: Simple Case, Using Extra Register Case

is called pre-loading. Of course, the purpose of pre-loading is not to increase the static load latencies of all load operations, otherwise this would increase the register pressure and no speedup would be obtained. Our pre-loading strategy selects a subset of delinquent loads as candidates. We proceed in two phases, explained below.

The first phase of our pre-loading strategy is the same used for the prefetching, *i.e.* we start with a precise profiling phase. This profiling allows us to detect delinquent loads as well as the code fragments to which they belong (function or loop).

The second phase of our pre-loading strategy defines the right load-use distance to each load. This is a major difficulty in practice: a compile time prediction of the probability of cache misses and hits is difficult (if not impossible) at the back-end level. This is why the initial phase of fine-grain profiling provides useful information. Depending on ratio of hit/miss for each load, we compute a certain probability of dynamic load latencies that we set at compile time. For instance, if a load misses the cache 30% of the times (143 cycles of latency) and hits 70% of the time (3 cycles of latency), then its static latency is set to $0.3 \times 143 + 0.7 \times 3 = 45$. If the register pressure becomes very high because of this long static latency, the compiler cannot extract enough ILP to hide this latency, then we reduce the latency. Currently, our method iterates on different values of static load latencies until reaching a reasonable performance gain. For our case of embedded systems, the compilation time is allowed to last during such iterative process.

Thanks to our pre-loading technique, we can achieve a pretty good performance increase. However, we must take care of the following points:

- Increasing static load latencies renders the compiler more aggressive regarding ILP extraction (deeper loop unrolling, global scheduling, super-block formation, etc.). Consequently, the code size may increase, or the memory layout of the code can be modified. This can have negative effects on instruction cache misses. Furthermore, it is better to skip the pre-loading optimisation for shorter trip count loop. It is especially the case of software pipelined loop with few iterations: increasing the static load latency increases the static II. If the number of loop iterations is not high enough, then the software pipelining would be too deep for reaching the steady state of the kernel.

- For other kinds of code (*i.e.* non-loop code), if the new load latencies are too long, the compiler may not find enough independent instructions to schedule between the load and its costumer. To avoid that, many techniques can be applied in combination with pre-loading such as tail duplication, region scheduling, super-block instruction scheduling, trace scheduling, scheduling non-loop code with prologue/epilogue of loop blocks, etc. And all these aggressive ILP extraction methods usually yield a code size increase.

- The last important point is that when increasing the load latency, the register pressure may increase. This fact can have bad effects if there are not enough free registers and oblige the compiler to introduce spill code to reduce the simultaneously alive variables. If spill code cannot be avoided, pre-loading should not be applied.

The pre-loading technique is efficient and practical because it can be applied on irregular codes with/without irregular data strides. It can also be applied in combination with other high or low level code optimisation techniques. Algorithm 1 details our whole methodology of data prefetching and pre-loading.

6. Experiments

For our experimentation, we used a cycle accurate simulator provided by STmicroelectronics. The astiss simulator offers the possibility to consider non-blocking cache. We fix the number of MSHR (the pending loads queue) to eight. We make the choice of eight MSHR because during experimentation, we observe that the ILP and register pressure reach a limit when MSHR is set to eight; a larger MSHR does not bring more performance.

In our experimental study, we use a precise cycle accurate simulator provided by the vendor because of many reasons:

Algorithm 1 The Prefetching and Pre-loading Algorithm

Require: struct ld <list> Tload;
{ Our methodology is profile guided, it's needs a profiling information about loads and their strides. The profiling information is summarized in a sorted list **Tload** (by the number of misses) of loads and attached to each load a sorted list of strides **Tstride** (sorted by occurrence of each stride). So Tload is a list of list. Each element of the main list is a structure which contain a load identifier (object adress), and a list of strides. Each element of the stride list is a structure which contain a stride and his occurrence.}
Require: float P;
{ **P** is the ratio to define a single strong stride, it's used to discard the noise values}
Require: int N;
{**N** is the maximum number of added pft instructions}
LOAD <list> iterator it1;
{ the it1 iterator is used to browse the Tload list}
for (it1=Tload.begin();it1=!Tload.end();it1++) **do**
  if (occurrence of the most frequent stride $>$ **P**% $\times$ sum of all occurrences) **then**
    {The first stride is used more than **P** % of time, case of strong single regular stride}
    look in assembly source in a load free bundle available
    if (free load bundle available) **then**
      insert prefetch instruction with the right stride
    else
      insert a new bundle with the prefetch instruction
    end if
  else
    {browse the list of strides to get more than **P** % of cumulated frequency stride}
    int b,sum=0;
    while (sum $<$ **P**% $\times$ sum of all occurrences) **do**
      b++;
    end while
    {b is the number minimum of strides to achieve more than (**P**)% of all strides.}
    if (b $> N$) **then**
      {case of phased multi-stride}
      look in assembly source N free loads bundles available
      if (N freel oads bundles are available) **then**
        insert N prefetch instruction with each stride
      else
        insert a N new bundles each one contains one prefetch instruction
      end if
    else
      {fully irregularity of the strides use of the preload solution}
      $ld_use = miss_ratio \times miss_penalty + hit_ration \times hit_penalty$
      for (while available free registers are available and enough ILP) **do**
        increase the load to use distance
      end for
    end if
  end if
end for

- We do not have a physical machine based on a VLIW ST231 processor. These processors are not sold for workstations, and are parts of embedded systems such as mobile phones, DVD recorders, digital TV, etc. Consequently, we do not have a direct access to a workstation for our experiments.

- The ST231 processor has a blocking cache architecture, while we conduct our experimental study on a non-blocking one. Only simulation allows to consider non-blocking cache.

- Our experimental study require precise performance characterisation that is not possible with direct measurement on executions: the hardware performance counters of the ST231 do not allow to characterise processor stalls we are focusing on (stalls due to Dcache misses). Only simulation allows to measure precisely the reasons of the processor stalls.

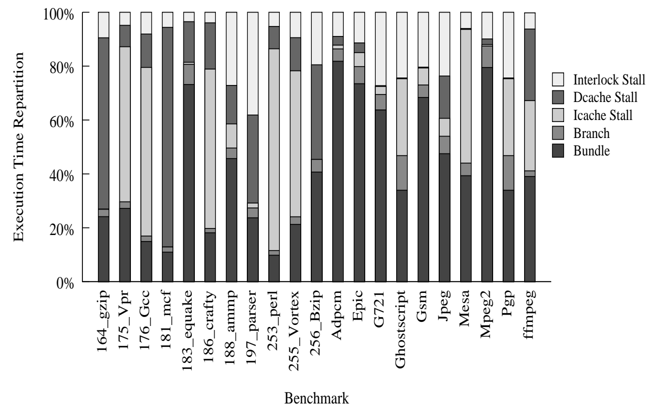

Figure 1. Time Execution Repartition for Spec Benchmark

Concerning the compilation phase, we use the -O3 compilation option for all tested benchmarks with the st200cc compiler. The st200cc is the commercial compiler provided by STmicroelectronics. Pre-loading strategy has been implemented inside this compiler to set the loads latencies at different granularity levels: loops, functions, application. The compiler does not insert prefetch instructions, so we insert them inside the assembly code using our prefetch algorithm.

Concerning the used benchmarks for experimentation, we use well known benchmarks such as Spec2000 and mediabench. Furthermore, we use the vendor benchmark called ffmpeg used for their internal research. At a first time, we made a precise performance characterisation of all these benchmarks. We express the total execution time in

terms of the following formula: $T = Calc + DC + IC + InterS + Br$. 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 branch penalties. Fig. 1 plots the performance characterisation of the used benchmarks. As can be seen for mediabench applications, only small fraction of execution time is lost due to Dcache penalties, except in the case of jpeg. So, most of the mediabench applications will do not take advantage from Dcache optimisation techniques on ST231. The best candidates for our low level cache optimisation method are the benchmarks which contains large Dcache penalty fractions. As shown in Fig. 1, Mcf and Gzip seem to be the best candidates for Dcache improvement. Indeed Mcf has more 76% of Dcache penalty, Gzip has more than 56% of Dcache penalty. Other benchmarks have smaller fractions of Dcache penalties, between 10% and 20% depending on the benchmark. However, these benchmarks have enough Dcache misses to expect some positive results. The benchmarks that have negligible fraction due to Dcache stalls are ignored for our optimisation strategy.

For each optimised benchmark, we made a precise trace analysis to determinate the regularity of the delinquent loads. We apply the prefetching and pre-loading techniques described before and we compare the results to the performance of the generated code with the -03 compiler optimisation level. Fig. 2 illustrates our experimental results (performance gain). As shown, the prefetch technique allows to have positive overall speed-up till 9.12 % (mcf). Thanks to prefetching, some cache misses are eliminated. However, prefetching requires regular data streams to be applied efficiently. If the data stream is not regular (non constant strides), the pre-loading technique is more efficient. While it requires a compilation trade-off between register pressure and load latencies, the produced performance gain is satisfactory in practice: we can get up to 6.83 % overall performance gain for bzip. The pre-loading technique gives good results except in crafty benchmark. After a deep study of crafty, we observed that specifying larger latencies for load instructions has a negative impact on a critical software pipelined loop. This loop causes a slowdown due to instructions cache penalty because the memory layout of the code changes, creating conflict misses. Note that we can obtain higher speed-up when we combine the two techniques conjointly. As shown in Fig. 2, jpeg gains more than 14% of execution time.

Regarding cache size, our prefetching technique does not introduce any extra code in practice; we succeed to schedule all prefetch instructions inside free VLIW slots. However, the pre-loading technique may introduce some additional

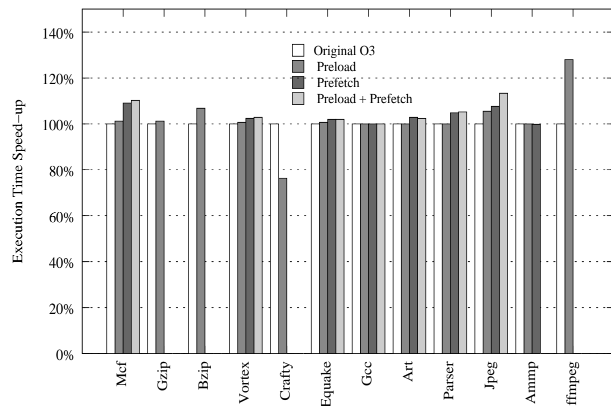

Figure 2. Efficiency of Prefetching and Preloading

negligible code size growth (3.9% in extreme case of mcf), see Fig. 3.

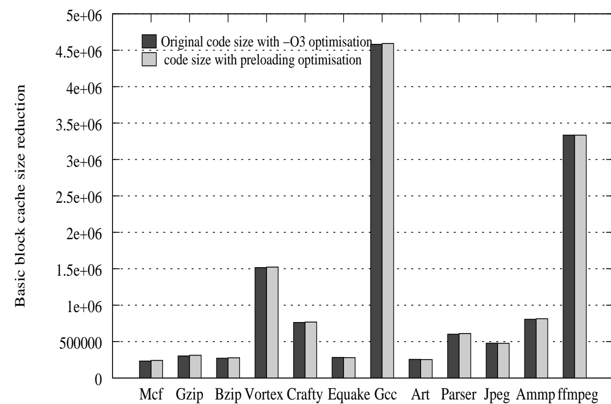

Figure 3. Comparison between Initial and New Codes Sizes

7 Conclusion

Our article presents an assembly level code optimisation method for reducing cache miss penalties. We target embedded VLIW codes executing on an embedded processor with non-blocking cache architecture. For experimental purpose, we used an embedded system based on a VLIW ST231 core. Contrary to high performance or computational intensive programs, the embedded applications that we target do not have regular data access or control flow, and the underlying hardware is cheap and simple. Our code optimisation method is based on a combination of data

prefetching and pre-loading.

Our method of data prefetching selects one or two delinquent loads that access a regular data stream that is not possible to analyse statically. Then, we insert one or two prefetch instructions inside a VLIW bundle for bringing data before time to the prefetch buffer or to cache. This simple method is efficient in case of blocking and non-blocking caches, where we can get a whole application performance gain up to 9 %. The code size doesn't increase in this situation. Our method of pre-loading consists of increasing the static load distance inside a selected loop or a function. This method allows the instruction scheduler to extract more ILP to be exploited in the presence of non-blocking cache. With pre-loading, we can get a minor code size growth (up to 3.9%) with an application performance gain up to 28.28 %. The advantage of pre-loading vs. prefetching is that it is not restricted to regular data streams. When we combine data prefetching with pre-loading in the presence of non-blocking cache, we get a better overall performance gain (up to 13 %) compared to optimised codes with -O3 compilation level. These performances are satisfactory in our case. The results of our study clearly show that the presence of non-blocking caches inside VLIW processors is a viable architectural improvement if the compiler applies some low level code optimisations, as we propose. Our future work will be devoted to improve data prefetching for embedded applications with irregular data accesses. We should be able to make a better usage of available caches and prefetch buffers present in the embedded processor.

Acknowledgements

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

References

[1] *ST231 Core and Instruction Set Architecture Manual*, 2005.

[2] C. D. Al-Sukhni H.F, Holt J.C. Improved stride prefetching using extrinsic stream characteristics. In *Performance Analysis of Systems and Software, 2006 IEEE International Symposium on Volume*, pages 166–176, 2006.

[3] J. C. Beyler and P. Clauss. Performance driven data cache prefetching in a dynamic software optimization system. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 202–209, New York, NY, USA, 2007. ACM.

[4] T.-F. Chen and J.-L. Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, Boston, Massachusetts, United States, 1992.

[5] C. Ding, S. Carr, and P. H. Sweany. Modulo Scheduling with Cache Reuse Information. In *Euro-Par '97: Proceedings of the Third International Euro-Par Conference on Parallel Processing*, pages 1079–1083, London, UK, 1997. Springer-Verlag.

[6] K. I. Farkas and N. P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. *SIGARCH Comput. Archit. News*, 22(2):211–222, 1994.

[7] Jiwei Lu and Howard Chen and Rao Fu and Wei-Chung Hsu and Bobbie Othmer and Pen-Chung Yew and Dong-Yuan Chen. The Performance of Runtime Data Cache Prefetching in a Dynamic Optimization System. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 180, Washington, DC, USA, 2003. IEEE Computer Society.

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

[9] Koray Öner and Michel Dubois. Effects of memory latencies on non-blocking processor/cache architectures. In *ICS '93: Proceedings of the 7th international conference on Supercomputing*, pages 338–347, New York, NY, USA, 1993. ACM.

[10] D. Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture*, pages 81–87, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[11] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. Lx: A Technology Platform for Customizable VLIW Embedded Processing. In *Proceedings of the 27th International Symposium of Computer Architecture (ISCA)*, pages 203–213, June 2000.

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

[13] Samir Ammenouche and Sid-Ahmed-Ali Touati and William Jalby. Practical Precise Evaluation of Cache Effects on Low Level Embedded VLIW Computing. In *HPCS, ECMS proceedings*, Nicosia, Cyprus, June 2008.

[14] Santosh G. Abraham and Rabin A. Sugumar and Daniel Windheiser and B. R. Rau and Rajiv Gupta. Predictability of load/store instruction latencies. In *MICRO 26: Proceedings of the 26th annual international symposium on Microarchitecture*, pages 139–152, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[15] Sid-Ahmed-Ali Touati. Optimal acyclic fine-grain scheduling with cache effects for embedded and real time systems. In *CODES '01: Proceedings of the ninth international symposium on Hardware/software codesign*, pages 159–164, New York, NY, USA, 2001. ACM.

[16] Y. Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. *SIGPLAN Not.*, 37(5):210–221, 2002.