Abstract

Cache misses represent a major bottleneck in embedded systems performance. Traditionally, compilers optimistically treated all memory accesses as cache hits, relying on the memory controller to account for longer miss delays. However, the memory controller has only a local view of the program, and is not able to efficiently hide the latency of these memory operations. Our compiler technique actively manages cache misses, and performs global miss traffic optimizations, to better hide the latency of the memory operations. Our memory-aware compiler scheduled several benchmarks on the TIC6211 processor architecture with a direct mapped cache, and generated an average of 61.6% improvement over the best schedule of the traditional (memory-transparent) optimizing compiler, demonstrating the utility of our miss traffic optimization approach.

1 Introduction

In recent SOC and processor architectures, memory is identified as a key performance and power bottleneck [23]. With the widening gap between processor and memory latencies, hiding the latency of the memory operations becomes increasingly important. In particular, cache misses are the most time-consuming operations (orders of magnitude longer than cache hits), being responsible for considerable performance penalties. This work proposes an approach to intelligently manage the cache miss traffic, and better hide the latency of the off-chip memory transfers.

Traditional optimizing compilers focused mainly on cache hit accesses (e.g., cache hit ratio optimizations), but did not actively manage the cache miss traffic. They optimistically treated all the memory accesses as cache hits, relying on the memory controller to hide the latency of the longer cache misses. However, not all cache misses can be avoided (e.g., compulsory misses). Moreover, since the memory controller has only access to a local view of the program, it cannot efficiently hide the latency of these memory accesses. This optimistic approach can be significantly improved by integrating an accurate timing model into the compiler, and hiding the latency of the longer misses. By providing the compiler with cache hit/miss traffic information and an accurate timing model for the hit and miss accesses, it is possible to perform global optimizations and obtain significant performance improvements. To our knowledge, no previous work has addressed memory-aware compilation tools that focus on the cache miss traffic, and specifically target the memory bandwidth, by using cache behavior analysis and accurate timing to aggressively overlap cache miss transfers.

The main contribution of this paper is the idea of explicitly managing the cache misses from the code to better utilize the memory bandwidth, and better overlap them with the cache hits and CPU operations. Since the misses are the most time-consuming operations, and represent the bottleneck in memory-intensive applications, by focusing on them first, we can reduce the memory stalls and improve performance.

In Section 2 we present previous work addressing cache and memory optimizations. In Section 3 we show a simple motivating example to illustrate the capabilities of our technique, and in Section 4 we describe our miss traffic management algorithm. In Section 5 we present a set of experiments that demonstrate the explicit management of main memory transfers for the TIC6211 architecture with a cache-based memory architecture, and present the performance improvements obtained by our memory-aware compiler. We conclude with a summary in Section 6.

2 Related Work

We have not seen any prior work that directly addresses compiler management of cache misses using cache analysis and accurate timing to aggressively schedule the cache miss traffic, and target it to the memory subsystem bandwidth. However, related work exists in 3 areas: (I) cache optimizations, which improve the cache hit ratio through program transformations and memory allocation, (II) cache behavior analysis, and cache hit prediction, and (III) memory timing extraction and exploitation.

I. Cache optimizations improving the cache hit ratio have been extensively addressed by both the embedded systems community ([3], [21]) and the main-stream compilers community ([25]). Loop transformations (e.g., loop interchange, blocking) have been used to improve both the temporal and spatial locality of the memory accesses. Similarly, memory allocation techniques (e.g., array padding, tiling) have been used in tandem with the loop transformations to provide further hit ratio improvement. Harmsze et. al. [12] present an approach to allocate and lock the cache lines for stream based accesses, to reduce the interference between different streams and random CPU accesses, and improve the predictability of the run-time cache behavior. However, often cache misses cannot be avoided due to large data sizes, or simply the presence of data in the main memory (compulsory misses).
efficiently use the available memory bandwidth and minimize the CPU stalls, it is crucial to aggressively schedule the cache misses.

Pai and Adve [19] present a technique to move cache misses closer together, allowing an out-of-order superscalar processor to better overlap these misses (assuming the memory system tolerates a large number of outstanding misses). Our technique is orthogonal, since we overlap cache misses with cache hits to a different cache line. That is, while they cluster the cache misses to fit into the same superscalar instruction window, we perform static scheduling to hide the latencies.

Another approach to improve the cache hit ratio used in general purpose processors is data prefetching. Software prefetching [2], [7], [17], inserts prefetch instructions into the code, to bring data into the cache early, and improve the probability it will result in a hit. Hardware prefetching [14], [20] uses hardware stream buffers to feed the cache with data from the main memory. On a cache miss, the prefetch buffers provide the required cache line to the cache faster than the main memory, but comparatively slower than the cache hit access. While software prefetching improves the cache hit ratio, it does not aggressively hide the latency of the remaining cache misses. Similarly, hardware prefetching improves the cache miss servicing time, but does not attempt to hide the latency of the stream buffer hits, and the stream buffer misses. We complement this work by managing the cache miss traffic to better hide the latency of such operations. Moreover, prefetching produces additional memory traffic [20], generating a substantial strain on the main memory bandwidth, due to redundant main memory accesses (e.g., by reading from the main memory data which is already in the cache), and often polluting the cache with useless data. While this may be acceptable for general purpose processors, due to the large power consumption of main memory accesses, in embedded applications the increased power consumption due to the extra memory accesses is often prohibitive. Furthermore, the main memory bandwidth is often limited, and the extra traffic along with the additional prefetch instructions and address calculation may generate a performance overhead which offsets the gains due to prefetching. We avoid these drawbacks by not inserting extra accesses to the memory subsystem, but rather scheduling the existing ones to better utilize the available memory bandwidth. We thus avoid the additional power consumption and performance overhead, by specifically targeting the main memory bandwidth, through an accurate timing, pipelining and parallelism model of the memory subsystem.

II. Cache behavior analysis predicts the number/moment of cache hits and misses, to estimate the performance of processor-memory systems [1], to guide cache optimization decisions [25], to guide compiler directed prefetching [17] or more recently, to drive dynamic memory sub-system reconfiguration in reconfigurable architectures [13], [24]. We use the cache locality analysis techniques presented in [17], [25] to recognize and isolate the cache misses in the compiler, and then schedule them to better hide the latency of the misses.

III. Additional related work addresses extraction and utilization of accurate memory timing in the context of interface synthesis [4], [5], hardware synthesis [16], [22], and memory-aware compilation [9]. Ly et. al. [16] use behavioral templates to model complex operations in a CDFG. Panda et. al. [22] present a pre-synthesis approach to exploit efficient access DRAM modes. [9] presents an approach to marry the timing of memory modules from a memory library with the processor pipeline timing, to generate the timings needed by the compiler, and exploit efficient memory access modes, such as page-mode and burst-mode accesses to improve performance. However, none of these approaches addresses intelligent management of cache miss traffic, and related optimizations. We complement this work by using the accurate timing information to better schedule the cache miss operations.

Our miss traffic optimization technique uses such an accurate timing model to manage the miss traffic and aggressively overlap the memory accesses, efficiently utilizing the memory bandwidth. Our approach works also in the presence of already optimized code (for instance cache hit ratio optimizations [21]), by using accurate timing, pipelining and parallelism information to better manage the memory accesses beyond cache hits, and further improve the system performance. We first predict and isolate the cache misses in the application. We then use the exact timing and pipelining information in the form of operation timings [9] and reservation tables [10] to schedule these cache misses. By allocating a higher priority to the transfers between the main memory and the cache, we ensure that the main memory bandwidth, which usually represents the bottleneck in memory-intensive applications, is effectively used, and the latencies are efficiently hidden, generating significant performance improvements.

3 Illustrative example

We use the simple illustrative example in Figure 1 (a) to demonstrate the performance improvement of our technique. In this example we assume a cache hit latency of 2 cycles, cache miss latency of 20 cycles, and non-blocking writes which take 2 cycles. We use a 16KB direct mapped cache with lines of 4 words, each word 4 bytes. For clarity of the explanation we assume no parallelism or pipelining available between cache misses, but cache hits can be serviced at the same time as cache misses.

We present the example code in three cases: (I) the traditional approach, where the compiler uses optimistic timing to schedule the memory operations, assuming they are all hits, and relies on the memory controller to account for misses, (II) the first phase of the miss traffic optimization, with cache miss traffic analysis and accurate miss timing, and (III) the second phase of the miss traffic optimization, with cache analysis and accurate timing, as well as aggressively optimized miss traffic schedule.

The primitive operation nodes used in Figure 1 represent the cache miss, cache hit, addition, and memory write operations. The shaded nodes grouping several such primitive nodes represent loop iterations.
I. In the absence of accurate timing information, the best the compiler can do is to treat all the memory accesses as hits, and schedule them optimistically, relying on the memory controller to account for longer delays. Figure 1 (b) presents the static schedule and the dynamic behavior for the traditional optimistic scheduling approach. In the example from Figure 1 (a) every fourth access to the arrays a and b results in a cache miss. As a result, in every fourth iteration the memory controller has to insert stalls to account for the latency of the cache misses, generating a total dynamic cycle count of 256 cycles.

II. By analyzing the cache behavior, and providing accurate timing information for all the accesses, the compiler can better hide the latency of the cache misses, and improve overall performance. Since we know that every fourth access to arrays a and b results in a miss, we unroll the loop 4 times (shown in Figure 1 (c)), and isolate the cache misses, to allow the compiler to attach accurate timing information to the memory accesses. The first 2 accesses in the unrolled loop body (a[i] and b[i]) result in cache misses, while all the other accesses result in hits. The compiler can attach accurate timing to the hit and miss accesses, to schedule the operations and hide some of the latencies associated with the memory operations. Figure 1 (c) shows the dynamic behavior for the code in Figure 1 (c) scheduled using accurate timing and cache behavior information. The dynamic cycle count in this case is 220 cycles, resulting in a 16% improvement over the traditional approach.

III. After performing cache analysis and attaching accurate timing information to the memory accesses, we aggressively schedule the miss traffic to create more opportunity for memory access overlap. By recognizing that often (especially after cache locality optimizations have been applied), accesses to the same cache line are close together in the code, the performance can be further improved. Since the first access to a cache line generates a miss and the subsequent accesses to that line have to wait for that miss to complete, the compiler is limited in its capability to efficiently schedule these operations (we will introduce in Section 4 the notion of "cache dependency", which captures such information). However, by overlapping the cache miss to one cache line with cache hits to a different cache line, it is possible to increase the potential parallelism between memory operations, generating further performance improvements. For instance, the cache miss a[i] from Figure 1 (c) accesses the same cache line as the hits in a[i+1], a[i+2], and a[i+3], and requires them to wait for the completion of the transfer from the main memory. By shifting the array accesses a[i] and b[i] from iteration i to the previous iteration, i-1, we allow for more parallelism opportunities. Figure 1 (e) shows the unrolled and shifted code, which further optimizes the cache miss traffic by overlapping the cache misses to one cache line with the cache hits from another cache line.

Figure 1 (f) shows the dynamic behavior for the unrolled and shifted code, optimized using accurate timing and cache behavior information to aggressively overlap the cache miss traffic (the figure does not depict the loop prologue and epilogue). The dynamic cycle count is 187 cycles, generating a further 17.6% performance improvement over the already optimized version from Figure 1 (d).

Thus, a more accurate timing model and cache behavior information allows the compiler to more aggressively overlap the cache hit/miss operations, and create significant performance improvement over the traditional optimistically scheduled approach.

---

2 The static schedule and the dynamic execution in this case are the same.
Algorithm: MIST  
Input: Memory hierarchy timing and pipelining model and Application code.
Output: Application code optimized for memory miss traffic

Begin MIST
1. Perform reuse analysis and determine the miss accesses.
2. Isolate cache misses.
3. Attach accurate timing to the cache hit and miss accesses.
4. Perform cache dependence analysis.
5. Perform loop shifting to reduce the dependence chains in the loop body.
6. Perform Trailblazing Percolation Scheduling (TiPS).
End MIST

Figure 2. The MIST Miss Traffic optimization algorithm.

4 Miss Traffic Optimization Algorithm

We present in the following the Miss Traffic Optimization algorithm (called MIST), which performs aggressive scheduling of cache miss traffic to better target the memory subsystem architecture, and generate significant performance improvements. The MIST algorithm receives as input the application code, along with the memory hierarchy timing model in the form of operation timings [9], and the memory subsystem parallelism and pipelining model captured as Reservation Tables [10], both generated from an EXPRESSION [11] description of the processor/memory system architecture. MIST produces as output the application code optimized for cache miss traffic.

Figure 2 presents the Miss Traffic Optimization algorithm. The first step performs reuse analysis and determines the array accesses resulting in misses. The second step isolates the misses in the code through loop transformations, and the third step attaches accurate timings to these isolated accesses. The fourth step determines the dependences between the hits to a cache line and the misses which fetch the data from the main memory for that cache line, while the fifth step performs loop shifting to transform these dependences from intra-iteration into loop-carried dependences. The last step performs Instruction Level Parallelism (ILP) extraction, to aggressively overlap the memory operations.

Whereas the main contribution of this paper is the idea of optimizing the cache miss traffic to better utilize the memory bandwidth, the loop shifting algorithm is only an instance of such a miss transfer optimization we propose. Once the information regarding miss accesses along with the accurate timing, pipelining and parallelism models are known, other parallelizing optimizations such as software pipelining can be successfully applied.

We will use the example code from Figure 1 (a) to illustrate the MIST algorithm. While this example has been kept simple for clarity of the explanations (single-dimensional arrays, simple array index functions and no nested loops), our algorithm can handle multi-dimensional arrays, with complex index functions, accessed from multiple nested loops.

Step 1 performs cache behavior analysis, predicting the cache misses, while Step 2 isolates these misses in the code through loop transformations. We use a standard cache behavior analysis and miss isolation technique presented in [17] and [25]. For instance, the array reference a[i] from Figure 1 (a) results in a miss access every 4'th iteration of the loop (assuming a cache line size 4), and the set of predicted cache misses is:

$$Miss\ (a[i]) = \{ i \mid 0 \leq i \leq 15 \text{ and } i \mod 4 = 0 \}$$

Figure 1 (c) shows the isolation of the cache misses of the array access a[i] in the original example through loop transformations. After unrolling the loop 4 times, the first access to array a in the body of the unrolled loop (a[i]) generates always a miss, while other 3 accesses (a[i+1], a[i+2], a[i+3]) generate hits.

The third step of the MIST algorithm attaches the accurate timing model for each of the array accesses, depending on whether it represents hits or misses. The timing information is received as an input by the MIST algorithm, and specifies for each operation the moment when the operands are read/written and the latency of the operation [9]. Additionally, a model of the pipelining and parallelism, represented as Reservation Tables [10] is attached to each operation. Together, the timing and the pipelining/parallelism models allow the compiler to aggressively schedule these operations.

For instance, for the array access a[i] in Figure 1 (c) which represents cache misses, we attach the latency of 20 cycles, along with the reservation table representing the resources used during execution (e.g., fetch, decode, ld/st unit, cache controller, memory controller, main memory, etc.). Due to space limitations we do not discuss the operation timing and reservation table models. For details please refer to [10] and [9].

The miss access to a cache line is responsible for bringing the data into the cache from the main memory. Hit accesses to that cache line have to wait on the access which generated the miss on that line to complete, making the data available in

---

3 We could also use alternative techniques such as those presented in [6].
the cache. In order to aggressively schedule such instructions, the compiler must be aware of this dependence between the hit access, and the miss access to the same cache line. In the following we will call this a “cache dependence”.

The fourth step determines cache dependences between different memory accesses. The cache dependence analysis algorithm is presented in Figure 3. There is a cache dependence between two accesses if they refer to the same cache line, one of them represents a miss, and the other a hit. The two accesses refer to the same cache line if there is group spatial locality between them (as determined in Step 1). For instance, in the example code from Figure 1 (c), a[i+1] depends on a[i], since a[i+1] refers to the same cache line, and a[i] generates a miss. The cache dependences are used in the following to guide loop shifting and parallelization, and facilitate the overlap between memory accesses which do not depend on each other.

In Step 5 of the Memory Traffic Optimization algorithm we perform loop shifting to increase the parallelism opportunity between cache miss and hit accesses. The loop shifting algorithm is presented in Figure 4.

Often memory accesses which address the same cache line are close together in the code (especially after performing cache optimizations such as tiling [21]). As a result, the cache hits and the miss to the same line create long dependence chains, which prohibit the compiler from aggressively overlapping these memory accesses (even if the compiler would optimistically overlap them, the memory controller would insert stalls, resulting in performance penalties). Step 5 performs loop shifting to transform the cache dependences from intra-iteration dependences into loop-carried dependences. By reducing the intra-iteration dependence chains, we increase the potential parallelism in the loop body, and allow the compiler to perform more aggressive scheduling of the memory operations. In the example from Figure 1 (c), the miss accesses a[i] and b[i] create a cache dependence on the hits from a[i+1], b[i+1], etc.. To reduce the dependence chains in the loop body (by transforming the intra-iteration dependences into loop-carried dependences), we shift the miss accesses a[i] and b[i] to the previous iteration, as shown in Figure 1 (e). As a result an increased parallelism is exposed in the loop body, and the compiler can better overlap the memory operations. Of course this loop shifting technique results in increased code size but yields better performance. Therefore, for space-critical embedded applications, the designer will need to tradeoff increase in code size for improved performance.

In Step 6 we use an Instruction Level Parallelism (ILP) scheduling approach to parallelize the operations in the loops, based on the accurate timing models derived in the Step 3. While other ILP scheduling technique could be used as well to parallelize the code, we use Trailblazing Percolation Scheduling (TiPS) [18], a powerful ILP extraction technique which allows parallelization across basic-block boundaries.

Due to the accurate timing information, and the loop shifting which increases the potential parallelism between memory accesses, the ILP scheduling algorithm generates significantly more parallelism than in the traditional version, with optimistic timing for the memory accesses. The resulting code presents a high degree of parallelism in the memory miss traffic, efficiently utilizing the main memory bandwidth, and creating significant performance improvements. For more details on the MiST algorithm please refer to [8].

5 Experiments

We present a set of experiments demonstrating the performance gains obtained by aggressively optimizing the memory miss traffic on a set of multimedia and DSP benchmarks. We perform the optimization in two phases: first we isolate the cache misses and attach accurate hit and miss timing to the memory accesses, to allow the scheduler to better target the memory subsystem architecture. We then further optimize the cache miss traffic, by loop shifting to reduce the intra-iteration dependence chains due to accesses to the same cache line, and allow more overlap between memory accesses. We compare both these approaches to the traditional approach, where the scheduler uses an optimistic timing to parallelize the operations, the best alternative available to the compiler, short of accurate timing and cache behavior information.

5.1 Experimental setup

In our experiments we use an architecture based on the Texas Instruments TMS320C6211 VLIW DSP, with a 16k direct mapped cache. The TIC6211 is an integer 8-way VLIW processor, with 2 load/store units. The latency of a hit is 2 cycles, and of a miss is 20 cycles.

The applications have been compiled using the EXPRESS retargetable optimizing ILP compiler, and the Trailblazing Percolation Scheduling (TiPS) [18] algorithm, a powerful Instruction Level Parallelism (ILP) scheduling technique. The cycle counts have been computed using our cycle-accurate structural simulator SIMPRESS [15]. The timing models have been generated using the TIMGEN and the pipelining/parallelism models in the form of Reservation Tables have been generated using the RTGEN Reservation Tables Generation algorithm [10] from an EXPRESSION description of the architecture. The timing and reservation tables generated from EXPRESSION model both the memory subsystem architecture and the processor pipeline. To clearly separate out the performance improvement obtained by the miss traffic optimization algorithm, we performed the best schedule available in the traditional version, using TiPS, and accurate timing, pipelining and parallelism information for both the processor, and the cache hit operations, while the cache miss operations are handled by the memory controller.

5.2 Results

The first column in Table 1 shows the benchmarks we executed (from multimedia and DSP domains). The second column represents the dynamic cycle count for the traditional approach, using an optimistic timing for the memory accesses, assuming that all accesses are cache hits, and no loop optimizations performed. The third column represents the dynamic cycle count for the first phase of our optimization, using accurate timing and cache behavior information, while
the fourth column shows the corresponding percentile performance improvement over the traditional approach. The fifth column represents the dynamic cycle count for the second phase of our optimization, using both accurate timing and cache behavior information and optimized for cache miss traffic through loop shifting. The sixth column represents the percent performance improvement over the first phase of our optimization, while the last column shows the improvement of the second phase over the base-line traditional approach.

The performance improvement due to providing the compiler with information on the memory accesses which result in hits and misses, and attaching accurate timing information for each of these (shown in columns 3 and 4) varies between 15.2% (for beam, where the cache miss isolation was conservative, optimizing only the misses in the innermost loop) and 52.8% (for gsr, where the loop body contains multiple misses which can be efficiently overlapped), resulting in an average of 34.1% gain.

The extra performance obtained by further overlapping the misses to one cache line with cache hits to different lines through loop shifting (shown in columns 5 and 6) varies between 0% (for gsr, where there are enough independent accesses in the loop body to parallelize without loop shifting), and 45.7% (for mm, where the intra-iteration dependence chain between the miss and the hits to the same cache line is significantly reduced through loop shifting), generating an average of 21.3% further improvement over the first phase of the optimization.

The overall performance improvement obtained by both phases of the optimization over the traditional approach (shown in column 7), varies between 37.6% for idct (where in order to keep the degree of loop unrolling low we used conservative cache prediction and isolation information, by heuristically considering some of the hits as misses), and 85.2% (in wavelet), with an average of 61.6%. In general, when a hit is wrongly predicted as a cache miss, we schedule it earlier, generating no penalty. When a miss is wrongly predicted as a cache hit, the resulting schedule will be similar to the traditional approach.

The large performance gains obtained by isolating the cache misses and attaching accurate timing, are due to the better opportunities to hide the latency of the lengthy memory operations. Furthermore, by loop shifting we reduce the dependence chains in the loop body, and create further parallelism opportunities between cache hits and misses to different cache lines. This effect is particularly large in multimedia applications showing high spatial locality in the array accesses (present in the original access pattern, or after cache optimizations such as tiling[21]).

6 Summary

We presented an approach for compile-time cache miss traffic optimization, which considerably improves performance through better utilization of the memory bandwidth, and hiding of the lengthy memory latencies. We use cache behavior analysis together with accurate cache hit/miss timing information and loop transformations to better target the processor/memory system architecture.

Traditionally, the compiler used an optimistic timing model for the memory operations, assuming that all accesses result in cache hits, and relying on the memory controller to account for the longer delays due to misses. However, the memory controller only has access to a local view of the program, and cannot efficiently hide the latency of these memory operations. By providing the compiler with cache hit/miss prediction and accurate timing, we perform more global optimizations, to exploit the parallelism available in the memory subsystem, and better hide the latency of the miss accesses.

We presented a set of experiments which show the performance gains obtained by our miss traffic optimization approach. The average performance improvement for using cache hit/miss information to better model the memory access timing was 34.1% over the schedule using the traditional optimistic timing model. By further optimizing the cache miss traffic, an additional 21.3% average improvement was obtained, generating an overall 61.6% average improvement over the traditional approach.

Currently, our work applies to wide issue statically scheduled VLIW processors. We believe that the technique presented in this paper is also applicable to dynamically scheduled processors. Our on-going work evaluates the improvements of our approach for out-of-order issue superscalar processors, and in the presence of other cache optimizations. We also addresses the tradeoff between increase in code size (due to loop unrolling) versus performance improvement.

7 Acknowledgments

We would like to acknowledge and thank Ashok Halambi, Nick Savoiu, Radu Cornea, Prabhat Mishra, Srikanth Srinivasan and Soumitri Pal for their contributions to the EXPRESS/EXPRESSION project.

References


