Locality-Aware Code Generation using EPIC Extensions

Kristof Beyls* Erik H. D'Hollander
Electronics and Information Systems
Ghent University
Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium
kristof.beyls@elis.UGent.be

Abstract

The memory-processor speed gap has grown so large that in modern systems accessing the main memory requires hundreds of processor cycles. Traditionally, a cache hierarchy is inserted between processor and memory to narrow the speed gap. However, since a cache has no knowledge about future references, data is stored at all cache levels, even if it exhibits no locality. Recently, EPIC architectures introduced cache hints which allow to specify the cache level where data is stored. In this way it is possible to adapt the allocation and replacement strategy based on the locality of the instruction.

In order to exploit cache hints, a compiler algorithm is proposed which calculates the locality of memory accesses. When there is little locality for a given cache level, the data is not stored at this level, which reduces cache pollution. The goal is to store the data at the lowest cache level where it will stay at least until the next access. As a result of the locality-aware code generation, speedups of up to 20% are measured on a number of pointer-intensive and numerical benchmarks. Furthermore, the results of the locality analysis are also used in the instruction scheduling phase in the compiler, so that the scheduler has a more accurate idea of the true latency of a load operation at run time. In a set of benchmarks, this leads to a speedup of up to 57%.

1 Introduction

Typically, for half of the execution time of a program, the processor is stalled because of cache misses. Despite the large number of researchers which worked on improving cache behavior in the past, speeding up programs clearly requires further work on bridging the memory-processor speed gap. In this work, we investigate how EPIC cache hints can help diminish the effect of memory accesses with poor locality. EPIC cache hints are annotations to memory instructions, and come in two flavors: *target* and *source* cache hints. The target cache hint indicates at which cache level data should be retained, and the source hint encodes

*This research is supported by a grant from the Flemish Institute for the promotion of Scientific Technological Research in the Industry (IWT)

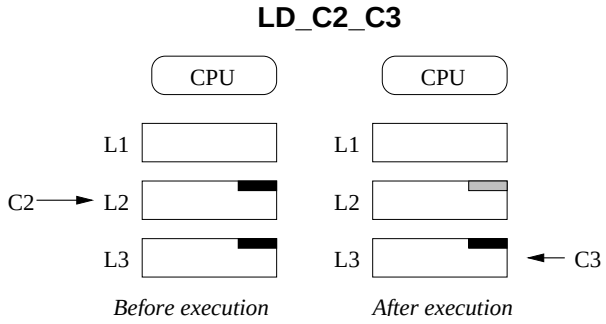

Figure 1: A load instruction with source cache hint C2 and target cache hint C3. Source hint C2 indicates that the data is expected at cache level 2, while target hint C3 requires the data only to be kept at cache level 3. As a consequence, the data becomes the first candidate for replacement in the L2-cache.

the highest cache level where the data is expected to be found. An example of such cache hints on a load instruction is shown in figure 1. We aim to at least partially answer two questions:

1. How can a compiler use cache hints to improve cache behavior?
2. An instruction has a fixed target and a fixed source cache hint. However, different executions of the instruction might request different cache hints. Is it a serious limitation that all executions of an instruction share the same cache hint?

2 Reuse Distance as a Locality Metric for Generating Cache Hints

The selection of appropriate cache hints must be based on the data locality of the instruction it applies to. We employ *reuse distance* as a metric for measuring the locality. The *backward reuse distance* of a memory access is the number of memory locations referenced between the current access and the previous access to the same location. Similarly, the *forward reuse distance* is the number of memory locations accessed between the current and the next access to the same location. The backward reuse distance is used to estimate at which cache level data is found, and the forward reuse distance enables us to decide at which cache level data should be kept. In our experiments, cache hints are chosen so that they indicate the smallest cache level that is larger than the reuse distance of the access. Further details about the reuse distance metric and its use in cache hint selection can be found in [2, 1].

3 Reuse Distance Measurement: Profiling versus Analysis

Before cache hints can be selected, the reuse distances exhibited by the instruction needs to be measured. We use two alternative methods: profiling and an

profiling	analysis
• RD distribution per instruction • measured by instrumenting program • only measure RD for one particular input • applicable to all sequential programs	• RD for each access • measured by compiler analysis • computed polynomial describes RD for all possible inputs • only applicable to programs in the polyhedral model

Table 1: Comparison between profiling and analysis for reuse distance calculation. RD = reuse distance

analytical calculation. During profiling, for each instruction, a distribution of its reuse distance is measured. Based on this distribution, a single cache hint is selected for all the executions of that instruction. In contrast, the analytical method calculates the reuse distance from the source code. The result of the analysis is a polynomial containing the induction variables of the iteration space of the instruction, representing the reuse distance of each execution of the instruction. This enables to generate code in which the cache hint is dynamically tailored for each execution of the memory instruction. The precise working of profiling and analytical calculation of reuse distances is discussed in [1].

Experiments on a number of programs have shown speedups on an Intel Itanium processor of 20% after target cache hint generation, and 57% after source cache hint generation. Furthermore, on average, fixed target hints based on profiling reduce cache misses by 6.7%, while dynamic cache hints resulting from analytical calculation of reuse distance remove 11.7% of the cache misses.

4 Conclusion

The exponentially growing speed gap between processor and memory requires ever more powerful techniques to improve cache behavior. One new method is using EPIC cache hints, which makes the cache visible to the compiler. By generating cache hints in the compiler, after doing a locality analysis, programs were speeded up by 57% maximum, with an average of 7%. Currently, cache hints in EPIC architectures are defined to be static, i.e. the cache hint associated with a memory instruction is fixed for all executions of that instruction, even if its locality varies over time. These static hints resulted in 6% less cache misses. If the ISA was adapted to also allow dynamic cache hints, the cache misses could be diminished by 12%.

References

[1] K. Beyls and E. D'Hollander. Compile-time cache hint generation for EPIC architectures. In *Proceedings of the 2nd Workshop on Explicitly Parallel Instruction Computing Architecture and Compilers (EPIC-2)*, 2002.

[2] K. Beyls and E. H. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of PDCS'01*, pages 617–662, Aug 2001.