Improving TriMedia Cache Performance by Profile Guided Code Reordering

Norbert Esser¹, Renga Sundararajan¹, and Joachim Trescher²

¹ NXP Semiconductors, San Jose, CA, USA
² NXP Research, Eindhoven, The Netherlands
{norbert.c.esser, renga.sundararajan, joachim.trescher}@nxp.com

Abstract. There is an ever-increasing gap between memory and processor performance. As a result, exploiting the cache becomes increasingly important, especially for embedded systems where cache sizes are much smaller than that of general purpose processors. The fine-tuning of an application with respect to cache behavior is now largely dependent on the skill of the application programmer. Given the difficulty of predicting cache behavior, this is, even when great skill is applied, a cumbersome task. A wide range of approaches, in hardware as well as in software, can be used to relieve the programmer’s burden. On the hardware side, we can experiment, for example, with cache sizes, line sizes, replacement policies, and cache organization. On the software side, we can use various optimization techniques like software pipelining, branch prediction, and code reordering. The research described in this paper focused on improving performance by using code reordering techniques.

This paper reports on the work that we have done to reduce the number of line-fetches in the instruction cache. We have extended the functionality of the linker in the TriMedia compiler chain, such that the number of fetches during program execution is reduced. By reordering the code, we ensure that hot code stays in the cache and the cache is not polluted with cold code. Because fewer fetches are needed we expect a performance increase. By analyzing and profiling code, we obtain execution statistics that can help us find better code-allocations.

Keywords: cache, code layout, profiling.

1 Introduction

Like other processors, the TriMedia uses an instruction cache to speed up program execution. Code that is executed frequently is termed “hot code” and code that is rarely executed is “cold code”. Since cache lines are a finite resource, lines may be victimized and later fetched again. Two aspects affect the number of fetches during program execution. A victimized line may contain hot code and a fetched line may contain cold code, both of which need to be avoided for better I-cache performance.

These aspects bear similarity to register-allocation in compilation. One big difference between register-allocation and cache-line allocation is that cache-line allocation and victimization is performed by a fixed hardware algorithm and can only be influenced indirectly via the layout of the code.
By extending the TriMedia linker with the functionality to reorder code according to linker maps, we are able to investigate various code-reordering algorithms. The linker maps are constructed by using algorithms based on those described by Friedman [1] and Pettis and Hansen [2]. For reference, we also implemented a more basic algorithm that places the most frequently used code fragments consecutively in memory.

There are some unique differentiating aspects of our work compared to previous work. One is that we study code reordering for a VLIW machine. The “code blocks” of VLIW machines are larger than basic blocks of traditional machines which may have influence on the chosen algorithms. We investigate code reordering of decision trees (dtrees) [3], aka, treegions, [4][5]. Our research extends prior work by investigating reordering of both functions and decision trees.

Further, the results for the TriMedia TM3271 [6] show the usefulness of code reordering techniques in an embedded, multi-media context.

2 Previous Work

McFarling [7] proposes one of the earliest approaches to improve performance by remapping machine instructions. The approach specifically targets direct mapped instruction caches. Using profiling information, a tree is built with labels that correspond to basic blocks. Labeled blocks should be added to the cache, unlabeled blocks can be excluded from the cache. All instructions with the same label and all instructions with descendant labels are positioned so that they will not interfere in the cache.

Hwu and Chang [8] propose a similar approach to improve the efficiency of caching in the instruction memory hierarchy. They aim at maximizing the sequential and spatial localities by grouping, for each function, the basic blocks that tend to be executed in sequence. Functions are then placed in a sequential order, where each time the most important descendent function is placed after its ancestor.

Pettis and Hansen [2] proposed constructing an undirected edge-weighted call graph, in which nodes correspond to either procedures or basic blocks and the edges respectively correspond to calls between the procedures or to the blocks following each other directly in sequential execution. The edges are weighted by the number of times the call or execution takes place. Nodes joined by an edge with a large weight are merged using a “closest is best” strategy. By minimizing the overlap in cache lines between nodes with a high edge weight, they were able to gain performance improvements of 8 to 10 percent on average. They also conclude that the gain is predominantly due to repositioning basic blocks rather than on reordering procedures.

Friedman [1] proposes an approach similar to that of Pettis and Hansen. He uses a sequence graph instead of a call graph. His algorithm does not use a “closest is best” strategy when merging nodes. Friedman first generates a function call trace. The sequence graph is built up by sectioning of a “window” of this trace and increasing the weight on the arcs between all functions that are together in the window. The complete sequence graph is constructed by then sliding the window over the trace, until the end is reached.

Hashemi et al. [9] improve upon the work of Pettis and Hansen by applying graph-colouring techniques to map cache lines to procedures. Procedures are placed such that