1 INTRODUCTION

This chapter presents an overview of the thesis: the motivations behind our work, our objectives, and a brief description of our contributions. We target a performance improvement in superscalar processors by increasing the rate at which instructions can be provided to the processor execution engine. That is, we target the fetch engine of superscalar processors.

In order to increase fetch performance we have used a combined software/hardware approach: we first try to optimize existing applications to take full advantage of the present hardware; next we make minor modifications to the hardware to avoid repeating at run-time what was done at compile-time; finally, we analyze in detail the characteristics of optimized applications, and design a fetch engine which fully exploits these characteristics.

1.1 Motivation

1.1.1 Superscalar processor architecture

Superscalar processors represent the major trend in high-performance processors in the last several years [79]. These processors naturally evolve from pipelined architectures, and try to obtain higher performance in two ways: first, by simultaneously executing several independent instructions in parallel; second, by increasing the clock rate to speed up instruction execution.

When designing a high-performance processor, it is important to keep all parts of the processor balanced, avoiding bottlenecks whenever possible. For example, Figure 1.1 shows the typical processor pipeline with 5 stages. If we design a high-performance processor capable of executing five ALU operations at once, it is also important to ensure that we can feed the ALU stage, and retire those instructions without stalling the pipeline. This means fetching and decoding at least five instructions per cycle, to keep the ALU stage busy, and writing results and graduating instructions at a fast enough rate.

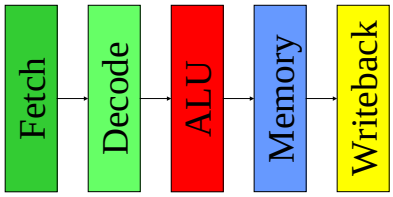

Figure 1.1. Example stages of instruction execution.

But the fetch stage does not behave like other pipeline stages in the sense that it can not be widened by simply replicating it, or adding more functional units. Furthermore, it has to follow the control path defined by branch instructions which have not been executed yet. The fetch stage quickly evolved to include branch prediction, and used it to fetch instructions from speculative execution paths.

This ability to follow speculative paths independently of the execution stages leads to a decoupled view of the processor, as shown in Figure 1.2. The fetch engine reads instructions from memory, and places them in an instruction buffer following an speculative path indicated by the branch prediction mechanism. Then, an execution engine reads instructions from the buffer and generates the required results, providing feedback to the fetch engine regarding the actual outcome of branch instructions.

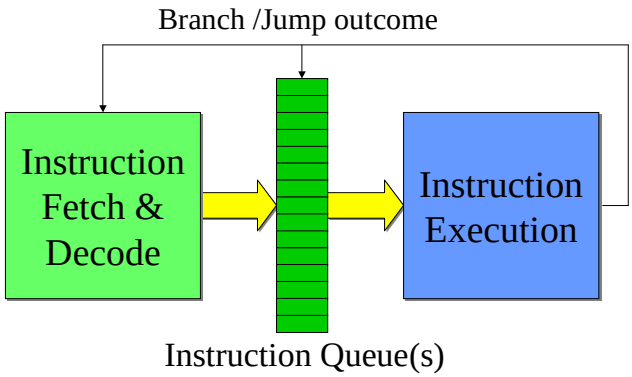

Figure 1.2. Decoupled view of the processor: a fetch engine produces instructions, and an execution engine consumes them.

An analysis of the decoupled view of a superscalar processor reveals that there are three main factors in fetch performance, as shown in Figure 1.3: (1) **memory latency**: how long it takes to read the instructions from memory, (2) **fetch width**: how many instructions we can transfer each cycle, and (3) **branch prediction accuracy**: how many transferred instructions belong to the wrong execution path.

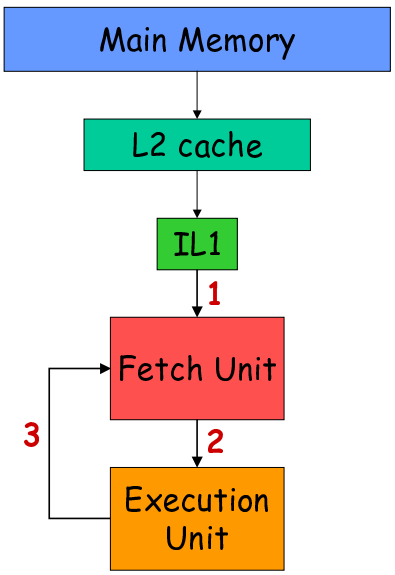

Figure 1.3. Fetch performance factors: memory latency, fetch width, and instruction quality.

The time it takes to load the required instructions from memory is computed together with the time it takes to execute the instructions. If the memory latency is large, it can quickly become the major component in the execution time. The main approach to reducing the memory latency is the use of cache memories and prefetching schemes. Given the popularity of this approach, instead of measuring instruction memory latency, we will measure the instruction cache miss rate.

As we mentioned earlier, the fetch engine can not be widened by simply replicating its functional units. Fetching more than one instruction per cycle requires a completely new fetch architecture, capable of selecting which instructions are to be fetched. This fetch architecture also determines how many instructions can be fetched simultaneously. The ability to fetch multiple instructions in a single cycle becomes a more important fetch performance factor as the issue width of the processor increases.

Finally, we must consider the presence of branch instructions which disrupt the flow of instructions through the pipeline. The problem arises when the outcome of a branch is not known until several cycles after it has been fetched, but we need to continue fetching instructions from a speculative path. By the time the branch has been resolved, several wrong path instructions may have entered the pipeline, and may need to be squashed. The squashing of wrong path instructions represents a wasted amount of fetch cycles, and directly affects fetch performance. The frequency of this event mainly depends on the accuracy of the branch prediction mechanism.

This thesis does not discuss the fetch engine of VLIW and CISC processors. Although they present interesting and hard to solve problems of their own, we concentrate instead on the fetch engine of superscalar processors. Also, we concentrate on the description of the fetch architectures only. Other elements found in the front-end engine of superscalar processors such as the instruction decode and the rename logic stages have not been treated here.

1.1.2 Objectives

Given the importance of fetch performance in superscalar processors, we target an increase in the rate at which useful instructions can be provided to the execution core.

However, we consider approaching fetch performance from a dual software/hardware perspective. We fist consider the use of compiler optimizations to adapt the existing applications to the underlying fetch architecture. The software approach is attractive for two reasons: first, it has a null hardware cost, it does not require additional transistors, and does not require additional power; second, it provides performance improvements on already existing architectures.

Once we have obtained an optimized code which better exploits the underlying hardware, we propose modifications to the different fetch architectures so that the work done by the software optimized is not repeated again during run-time, saving resources, and further increasing performance.

At the same time, we analyze in detail the characteristics of optimized codes, in order to gain a clear understanding of how they improve performance. Using the results of this analysis, we develop a new fetch architecture which fully exploits the unique characteristics of optimized applications to obtain high fetch performance, at a minimum cost.

Thesis objective: we try to adapt the existing software to the underlying architecture, then we re-adapt the architecture to the new software, and finally design a new fetch architecture taking the best from both the software and the hardware approaches.

1.2 Thesis overview

In this section we provide a brief description of the topic we deal with in this thesis. We present the problems we are trying to solve, the approach we take to solving the problem, and the novel contributions of our work.

1.2.1 Compiler optimizations for improved fetch performance

In this topic we analyze in detail the instruction behavior of several applications, and examine the source code of a database management application to understand its internal structure. Based on the results of this analysis we conclude that these applications have characteristics which could be exploited by the underlying fetch architecture, However, we find that they are oddly matched, and that performance is not what could be expected.

The performance of an application with regard to the fetch engine is mainly determined by its dynamic behavior, which we can not control, and by the mapping of instructions, which is determined by the compiler. It is possible to optimized the performance of an application by reordering the code so that it better suits the characteristics of the underlying fetch architecture. Such is the target of code layout optimizations.

There has been much work done on code layout optimizations. However, the proposed optimizations targeted only specific elements in the fetch engine like the instruction cache [24, 32, 36, 59, 87], or the branch prediction mechanism [8]. They do not consider the fetch width as a performance factor, because they targeted single issue pipelined processors, or narrow issue superscalars (less that 4 instructions per cycle), where a single basic block of instructions is usually wide enough. On wide issue superscalar processors (8 or more instructions per cycle), fetching instructions from a single basic block is not enough, and the fetch width becomes a critical factor.

For this reason, we propose a novel code layout algorithm which targets all three performance factors at the same time: instruction cache performance, branch prediction accuracy, and the fetch width. We call out code layout algorithm the *Software Trace Cache* (STC).

We analyze in detail the impact of the STC and other layout optimizations on the three main factors of fetch performance: instruction cache, fetch width, and branch prediction accuracy. Not only we measure the performance improvements in each factor, but we analyze the reasons for those improvements and provide insights on how optimized codes exercise the fetch architecture.

1.2.2 Hardware modifications to exploit software characteristics

After a detailed analysis of the interaction of the software trace cache with the hardware trace cache we find that there is significant redundancy between both mechanisms, as they all do the same task. The software trace cache joins basic blocks in traces at compile time, based on profile data. The hardware trace cache joins basic blocks in traces at run time, based on the dynamic execution paths encountered.

In order to avoid doing the same work both in the compiler and in the hardware, we propose a simple filtering mechanism that completely avoids using hardware resources for those traces that were built by the compiler. The use of this filter allows a more efficient implementation of the trace cache, which reaches the same performance level at a lower implementation cost.

Also, we explore the possibilities of using the profile information that was collected to guide the code layout optimization to improve the branch prediction mechanism. We analyze in detail how the compiler can improve the branch prediction mechanism for some of the best proposed branch predictors, and propose a novel branch predictor organization which requires extensive compiler support, but provided higher prediction accuracy than any other branch predictor examined.

1.2.3 Exploiting layout optimized codes

From the analysis of the behavior of layout optimized codes we have gained a clear idea of the unique characteristics of these codes: they contain a large proportion of not taken branches, and a vast amounts of spatial locality.

Based on these characteristics, we define a new code structure, which we call *stream*: an instruction stream is a sequence of sequential instructions going from the target of a taken branch, to the next taken branch. The instruction stream is a much larger entity than a basic block, and has a length comparable to that of an instruction trace is layout optimized codes. We design our fetch engine around the concept of a stream, which unveils the fetch potential of optimized applications.

Our fetch architecture is based on the fact that there is little performance advantage in increasing fetch performance beyond what the processor back-end can execute. For this reason, it is not always necessary to use the highest performing architecture, if there is an alternative which provides high enough performance, but has a lower implementation cost, or uses less energy. Such is the target of our stream front-end engine: to provide a high enough fetch performance at the minimum cost.

1.3 Document structure

Figure 1.4 shows an overview of the thesis, from the problem we observed, to the different solutions we proposed, leading to new problems or observations, which in turn opened the possibility of new proposals.

The initial problem statement is superscalar processor performance, and in particular the fetch performance of wide superscalar processors (from 8 to 16 issue pipelines). Our first approach to increasing fetch performance is the use of code layout optimizations. In Chapter 4 we propose the Software Trace Cache, a novel code placement algorithm which targets not only the instruction cache performance, but also an increase in the effective fetch width.

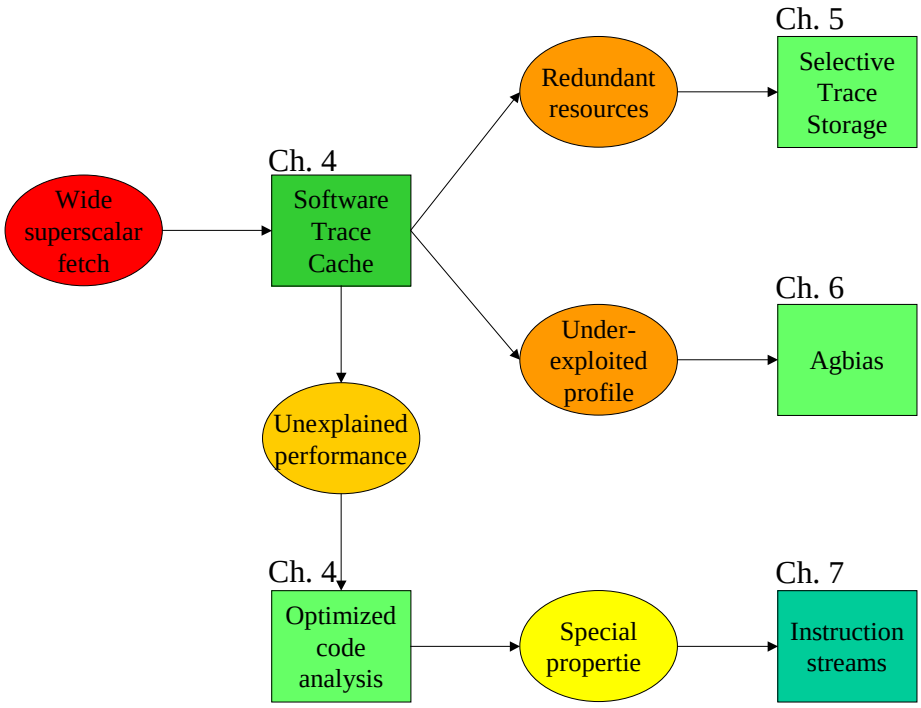

Figure 1.4. Thesis overview: from the problems observed to the proposed solutions.

In addition to measuring the performance improvements obtained, we perform a detailed analysis of the behavior of layout optimized applications with regard to the three fetch performance factors: memory latency, fetch width, and branch prediction accuracy. The detailed performance analysis is also presented in Chapter 4.

Analyzing the joint performance of the software and the hardware trace cache we find significant redundancy between both approaches. In Chapter 5 we propose Selective Trace Storage, a trace cache modification which eliminates this redundancy.

The STC uses profile data to optimize the layout of instructions in memory, but this does not exhaust the possible uses of the data obtained. In Chapter 6 we explore the possible uses of profile data to improve the branch prediction accuracy, and propose a novel branch predictor organization which relies extensively on the compiler.

Finally, based in the special characteristics of optimized applications, in Chapter 7 we introduce a new fetch architecture designed to fetch larger sized instruction sequences, which we call *instruction streams.*

In Chapter 8 we present our conclusions for this work, and present guidelines for future work, and new lines of research opened by our results.