Abstract. Compilers have long used redundancy removal to improve program execution speed. For handheld devices, redundancy removal is particularly attractive because it improves execution speed and energy efficiency at the same time. In a broad view, redundancy exists in many different forms, e.g., redundant computations and redundant branches. We briefly describe our recent efforts to expand the scope of redundancy removal. We attain computation reuse by replacing a code segment by a table look-up. We use IF-merging to merge conditional statements into a single conditional statement. We present part of our preliminary experimental results from an HP/Compaq iPAQ PDA.

1 Introduction

Compilers have long used redundancy removal to improve program execution speed. For handheld devices, which have limited energy resource, redundancy removal is particularly attractive because it improves execution speed and energy efficiency at the same time. In a broad sense, any reuse of a previous result can be viewed as a form of redundancy removal. Recently, our research group has investigated methods to expand the scope of redundancy removal. The investigation has resulted in two forms of operation reuse, namely computation reuse and branch reuse.

Computation reuse can be viewed as an extension of common subexpression elimination (CSE). CSE looks for redundancy among expressions in different places of the program. Each of such expressions computes a single value. In contrast, computation reuse looks for redundancy among different instances of a code segment or several code segments which perform the same sequence of operations. In this paper, we shall discuss computation reuse for a single code segment which exploits value locality [1, 2, 3, 4] via pure software means.

We exploit branch reuse through an IF-merging technique which reduces the number of conditional branches executed at run time. This technique does not require special hardware support and thus, unlike hardware techniques, it does not increase the power rate. The merger candidates include IF statements which have identical or similar IF conditions which nonetheless are separated by other statements. The idea of IF-merging can be implemented with various degrees of
aggressiveness: the basic scheme, a more aggressive scheme to allow nonidentical IF conditions, and lastly, a scheme based on path profiling information. In the next two sections, we discuss these techniques respectively and compare each technique with related work. We make a conclusion in the last section.

2 Computation Reuse

Recent research has shown that programs often exhibit value locality [1, 2, 3, 4], a phenomenon in which a small number of values appear repeatedly in the same register or the same memory location. A number of hardware techniques [5, 6, 7, 1, 2, 8, 9, 4] have been proposed to exploit value locality by recording the inputs and outputs of a code segment in a reuse table implemented in the hardware. The code segment can be as short as a single instruction. A subsequent instance of the code segment can be simplified to a table look-up if the input has appeared before.

The hardware techniques require a nontrivial change to the processor design, typically by adding a special buffer which may contain one to sixteen entries. Each entry records an input (which may consist of several different variables) and its matching output. Such a special buffer increases the hardware design complexity and the hardware cost, and it remains unclear whether the cost is justified for embedded systems and handheld computing devices. Using a software scheme, the table size can be much more flexible, although table look-up will take more time. The benefit and the overhead must be weighed carefully.

In our scheme, we use a series of filtering to identify stateless code segments which are good candidates for computation reuse. Figure 1 shows the main steps of our compiler scheme. For each selected code segment, the scheme creates a hashing table to continuously record the inputs and the matching outputs of the code segment. Based on factors such as value repetition rate, computation granularity estimation, and hashing complexity, we develop a formula to estimate whether the table look-up will cost less than repeating the execution. The hashing complexity depends on the hash function and the input/output size. The hashing table can be as large as the number of different input patterns. This offers opportunities to reuse computation whose inputs and outputs do not fit in a special hardware buffer.

2.1 How to Reuse

Computation reuse is applied to a stateless code segment whose output depends entirely on its input variables, i.e. variables and array elements which have upwardly-exposed reads in the segment. The output variables are identified by liveness analysis. A variable computed by the code segment is an output variable if it remains live at the exit of the code segment. If we create a look-up hash table for the code segment, the input variables will form the hash key. An invariant never needs to be included in the hash key. Therefore, for convenience, we exclude invariants from the set of input variables.