Compiler Optimizations for Media Processors

Rainer LEUPERS*

University of Dortmund
Dept. of Computer Science 12
44221 Dortmund, Germany
leupers@ls12.cs.uni-dortmund.de

Abstract

In the design of embedded systems, programmable processors gain more and more importance due to their high flexibility and potential for reuse. As a consequence, compilers for embedded processors are required, capable of generating very fast and dense code. In particular, this concerns the area of computation-intensive multimedia applications. While domain-specific digital signal processors may offer sufficient performance for multimedia, they show comparatively low flexibility and pose difficult problems to compiler and software developers due to their irregular architectures. In fact, meeting the system specification while minimizing the costs frequently requires time-consuming assembly-level programming of embedded processors. Recent media processors cover a larger set of application areas and, due to a more regular architecture, also facilitate the construction of compilers capable of generating high-quality machine code. However, media processors simultaneously introduce new challenges for compiler technology. In this paper, we motivate the use of media processors and we present two new compiler optimizations for such processors. [1]

1. Introduction

Today, many embedded systems are designed on the basis of instruction set processors rather than on purely application specific hardware. Such *embedded processors* are used to achieve higher degrees of reusability and flexibility both of which are key features in system design for increasingly complex applications and short market windows. Flexibility is achieved by programmability of embedded processors, which allows for quickly accommodating late specification changes or new standards in existing designs. Reusability covers both the reuse of software (if it is written in a high-level language) and the reuse of processors. Many embedded processors (RISCs, DSPs, and microcontrollers) are available in the form of *cores*, i.e., complex macro cells which can be instantiated from a component library. Such cores are generally used in *systems-on-a-chip*, a typical floorplan of which is shown in fig. 1.

If processor cores are employed in a design, then the silicon area occupied both by the cores and the program memory contribute to the total chip area and thus to the system manufacturing costs. There are two general approaches to ensure an efficient use of embedded processors.

*The author acknowledges the support by HP EEsof

[1] Publication: EMMSEC '99, Stockholm/Sweden, 1999

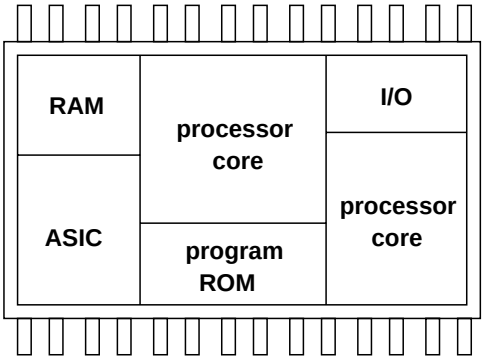

Figure 1: System-on-a-chip with embedded processor cores

- **Application or domain specific processors:**

  The use of standard ”general-purpose” processors in an embedded system for a particular application may lead to a waste of resources. Therefore, embedded system designers mostly make use of domain or even application specific processors. A large variety of such specialized processors, either tuned for high performance or low power consumption are available on the semiconductor market.

- **Generation of compact machine code:**

  When developing the software executed by an embedded processor, much time is spent in code optimization, so as to achieve dense code. In case an embedded system has to meet real-time constraints, the code performance may be of even higher importance. In particular for DSP processors, for which compiler support is still insufficient, programmers frequently have to resort to assembly programming in order to generate sufficiently compact machine code [1]. Since such low-level programming is extremely time-consuming, and both processors and applications get more and more complex, high-level language compilers capable of generating high-quality code will become very important design tools in the near future.

In this paper, we consider *media processors* as a specific class of domain specific embedded processors, and we propose two code optimization techniques for the use in compilers for media processors. The purpose of these techniques is an increase in the quality of compiler-generated code, which in turn helps to avoid assembly programming of embedded processors.

2. Media processors

Embedded systems used in the areas of telecommunication, image processing or consumer electronics involve a large amount of computation-intensive digital signal processing, such as filtering, coding, or compression. Since the performance of general-purpose processors is insufficient for such operations, dedicated digital signal processors (DSPs) are available. These DSPs are equipped with special hardware for fast execution of kernel operations in signal processing, like multiply-accumulate operations.

In spite of their performance advantages, DSPs unfortunately show two drawbacks: Due to the high specialization of DSP architectures, different processors have to be used for different application domains (e.g. audio and video). Furthermore, generation of high-

quality code for DSPs by compilers is very difficult and requires many special and time-consuming code optimization techniques (see [2, 3] for overviews).

In order to overcome these problems, semiconductor vendors are starting to offer so-called *media processors*. These processors provide instructions for efficient computations on different multimedia data types, such as 8-bit 16-bit or 32-bit data. While also some general-purpose processors offer multimedia support (e.g. Intel's Pentium with MMX technology [4]), special media processors like the Texas Instruments 'C6x [5] or the Philips Trimedia [6] in general allow for higher performance due to a large number of functional units. In addition, such processors are relatively regular and are thus much more "compiler-friendly" than traditional DSPs. Mostly, the architectures of media processors follow the VLIW (very long instruction word) paradigm to expose a high degree of instruction-level parallelism to the compiler and they show deep instruction pipelines to achieve high throughput.

While the VLIW-like architectures eliminate some difficulties encountered in code generation for traditional DSPs (like dealing with special-purpose registers and peculiar instruction formats), they also pose new challenges to compiler technology. In this paper we discuss two code generation problems associated with media processors, and we outline solutions for these problems.

Firstly, this concerns the fact, that control structures in application programs (such as if-then-else) cause hazards in the instruction pipeline which may significantly slow down the potential performance. Media processors provide *conditional instructions* as a means of compensating this effect, but effective exploitation of such instructions in a compiler is not a trivial problem.

Secondly, media processors are capable of performing computations on different data types by means of dedicated *multimedia instructions*. These instructions execute identical operations on multiple data in parallel. So far, the use of multimedia instructions in compilers is restricted to inlining of assembly library code or *compiler intrinsics*, which are expanded into specific instructions like a macro. The disadvantage of these approaches is that the resulting code is highly machine-dependent, which prevents the reuse of software for new processors.

3. Exploiting conditional instructions

Control structures in programs, like if-then-else (ITE), are normally compiled into machine code by means of *conditional jump* instructions. Depending on the ITE condition, either the then or the else part of an ITE statement are skipped. However, instruction pipelining implies that the processing of instructions is started already before it is established that these instructions are actually the correct ones w.r.t. the program control flow. In case the pipeline if filled with "wrong" instructions due to an execution of a jump instruction (called a pipeline hazard), the pipeline needs to be stalled for several instruction cycles.

Such hazards may significantly degrade the code performance. In case of the TI 'C6x, for instance, any jump instruction incurs a delay of up to 5 cycles. Since in any cycle up to 8 instructions may be executed, the maximum performance penalty of a jump is equivalent to 40 instructions. This can be avoided by using *conditional instructions*. A conditional instruction is denoted by $[c]I$, where I is any regular machine instruction, and c is a Boolean "guard" (stored in a register) that evaluates to true or false at program runtime. Instruction I is executed only if c = true, and behaves like a no-operation, otherwise. Except for explicit conditional jumps, conditional instructions have no effect on the program control flow and thus do not cause pipeline hazards. On the other hand, using conditional instructions in

general leads to a larger competition of instructions for processor resources and thus may lead to *structural hazards*, which in turn may degrade performance. Therefore, conditional instructions have to be used carefully.

From a compiler viewpoint, the presence of conditional instructions implies two alternative code generation schemes for ITE statements, one of which has to be selected so as to maximize performance. If $S = (cond, B_T, B_E)$ denotes an ITE statement, where $cond$ is the condition, and B_T and B_E are the then and else blocks, respectively, then a "traditional" implementation of S by conditional jumps looks as shown below.

```
            c := evaluate(cond)
        [c] goto then_label
            B_E
            goto join_label
then_label: B_T
join_label: ...
```

If c is true, then the execution time for S is equal to the execution time for B_T plus one jump penalty. Otherwise, if c is false, then the execution time is the time for B_E plus two jump penalties. The worst case execution time is the maximum of both possibilities. When using conditional instructions instead, the implementation scheme would be:

```
            c := evaluate(cond)
        [c]  B_T
        [!c] B_E
```

Here, the blocks B_T and B_E are guarded by their respective execution conditions and are concatenated to a single "large" block. If the instructions in both blocks do not compete for processor resources, then the execution time is equivalent to the time needed for the largest of the two blocks and thus is guaranteed to be lower than for the conditional jump scheme. On the other hand, concatenating two blocks with extensive resource utilization will in general lead to lower performance than conditional jumps.

The best of these two ITE implementation schemes can only be chosen in each case, if the execution times of B_T and B_E are (approximately) known. If B_T and B_E are *basic blocks* (i.e. branch free), then this can be achieved by estimation techniques. However, if both B_T and B_E in turn comprise ITE statements, then the estimation gets difficult. In particular, for nested ITE statements, also a mixture of the two implementation schemes may be required.

In case of a nested ITE statement, the implementations chosen for the inner ITE statements, which determine their execution time, must be known when deciding the scheme for the outer one. On the other hand, the execution time of inner statements also depends on the implementation selected for the outer one. This cyclic dependence is due to the fact, that translating a nested ITE statement into conditional instructions may require the insertion of additional code preserving program correctness.

We solve this problem by means of a *dynamic programming algorithm*. A nested ITE statement is considered as a binary tree, with left and right children of nodes corresponding to then and else blocks. The algorithm makes two passes over the ITE tree. In the first (bottom-up) pass, the execution times for program blocks within ITE statements are estimated for four possible configurations. These four configurations result from the fact, that any ITE statement may be implemented by conditional jumps or conditional instructions,

and for both schemes it may or may not be necessary to insert additional correctness preserving code. The four estimated values are passed upwards in the ITE tree until the root is reached. Then, the second (top-down) pass determines the implementation schemes based on the estimations. For the tree root (i.e. the outermost ITE statement), obviously no additional code is required, so that only two of the possible configurations need to be compared, and the minimum is selected. In turn, this selection generates the information, whether or not the ITE statements inside the outermost one require additional code, so that again only two possible configurations remain. This process is recursively continued down to the tree leaves. Afterwards, the best (i.e. fastest) implementation schemes for all ITE statements have been determined.

This algorithm is efficient, since its runtime is linear in the number of ITE statements. We have performed an experimental evaluation, where the machine code generated by the proposed algorithm was compared to the machine code generated by the TI 'C6x ANSI C compiler for different control-intensive source codes. In most cases, a reduction of the worst case execution time (as much as 30 %) has been observed. The price for this acceleration is an increase in code size (typically 25 %).

4. Exploiting multimedia instructions

The term "multimedia" implies that in applications one has to deal with different media data types, such as 8-bit video or 16-bit audio data. Execution operations on such data on a pure 32-bit processor would imply a huge waste of resources. So-called *multimedia instructions* enable a much more efficient use of resources. In terms of the C programming language, any 32-bit register, for instance, may store one 32-bit "integer", two 16-bit "short", or four 8-bit "char" data. Likewise, a 32-bit functional unit (with appropriate hardware support) may perform operations on two pairs of 16-bit arguments or four pairs of 8-bit arguments at a time. As an example, fig. 2 illustrates the behavior of a TI 'C6x machine instruction performing two parallel 16-bit additions on two pairs of 16-bit "subregisters".

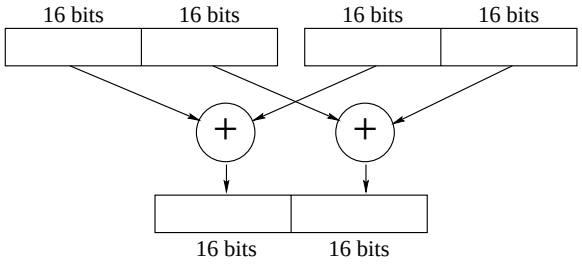

Figure 2: TI C6x multimedia instruction "ADD2"

Such multimedia instructions may significantly accelerate computations on media data. As an example, consider the vector addition source code in fig. 3. Executing the loop body once with 32-bit instructions requires four loads, two additions, and two stores. In contrast, when using the above "ADD2" instruction, the number of required instructions is halved. This is possible, since the memory accesses refer to adjacent locations in memory, which permits to load two 16-bit data simultaneously by a single 32-bit load instruction.

Traditional compiler techniques are inadequate for multimedia instructions due to a limited exploration of the search space during code generation. Most code generation techniques are based on the paradigm of tree pattern matching with dynamic programming [7]. Thus, their scope is restricted to code generation for expression trees, which are a standard form

```
void f(short* A,short* B,short* C)
{ int i;
  for (i = 0; i < N; i += 2)
  { A[i]   = B[i] + C[i];
    A[i+1] = B[i+1] + C[i+1];
  }
}
```

Figure 3: C source code for vector addition

for intermediate program representation. However, exploitation of multimedia instructions requires to perform code generation for multiple expression trees at a time (e.g. two trees, one per statement, in case of the code from fig. 3).

We propose to perform code generation on data flow graphs (DFGs) instead of trees. DFGs are a generalized graphical representation of basic blocks. The DFGs are generated by means of an ANSI C frontend. Any DFG is first decomposed into a set of expression trees by cutting the DFG at its common subexpressions. Each tree is *covered* with a minimum number of machine instructions by means of tree pattern matching. For this purpose, we use the olive code generator generator, a variant of which has been described in [7]. Instead of computing only a single optimal tree cover, we use a modified version of olive, which generates *alternative covers*. These alternatives reflect the fact that some operation, depending on the code generated for other trees, might be covered by a multimedia instruction. If, for instance, code is to be generated for an addition of two 16-bit variables, then both a regular 32-bit add instruction as well as the above "ADD2" instruction would be identified as alternative covers. However, using "ADD2" is only possible, if the DFG (possibly in a different tree) contains a "partner" instruction, such that both instructions eventually can be mapped to a single multimedia instruction.

Once all trees of a DFG have been covered, groups of instructions that can be implemented by a single multimedia instruction are identified globally for the entire DFG. This is possible, whenever there is no scheduling precedence between the group of instructions considered, and when appropriate multimedia instructions are available. In case of instructions accessing memory data it must be additionally ensured that these instructions refer to adjacent memory addresses.

We formulate the generation of multimedia instructions mathematically as an *integer linear programming* (ILP) problem. This formulation contains binary *solution variables*, whose values account for the detailed selection of machine instructions for a DFG. The restrictions on packing separate 32-bit instructions to multimedia instructions, such as available alternative covers and schedulability of instructions are encoded in the form of linear *constraints* on the solution variables. Finally, we specify an *objective function* in such a way, that for a given set of alternative DFG covers that cover is selected which maximizes the use of multimedia instructions.

For each DFG, the corresponding ILP is automatically generated and optimized by means of a public domain ILP solver (lp_solve from TU Eindhoven, available via ftp). Even though ILP is an exponential problem, covering of DFGs with not more than approximately 100 nodes can typically be done within a few seconds of CPU time.

For experimental evaluation, we have implemented code generators for the TI 'C6x and

the Philips Trimedia and compiled a set of DSP kernel routines into machine code for these processors. As can be expected, the use of multimedia instructions resulted in a significantly lower number of instructions needed to execute the DSP routines. This is no surprise, since multimedia instructions are designed for this purpose. However, the most important feature of the proposed code generation technique is that it allows for exploitation of multimedia instructions also when compiling *plain ANSI C source code.* That is, in contrast to other compilers, it does not require hand-optimized assembly libraries or ”compiler intrinsics”, which results in much higher portability of C source code. This is reflected by the fact that we compiled *identical* sets of C source codes to two different media processors.

5. Conclusions and recommendations

Currently, there is a trend towards software based implementation of embedded systems, where a large part of the system functionality is realized by machine code running on programmable embedded processors. As both embedded applications and embedded processors become more and more complex, high-level language compilers should be used for development of embedded software. However, due to the lack of compilers capable of generating efficient code, embedded processors are still mostly programmed in low-level assembly languages, resulting in a significant productivity bottleneck.

In order to overcome this problem, code optimization techniques beyond the scope of classical compiler technology are required, which take the specific architectures of embedded processors into account in order to improve code quality. In this paper, we have outlined two new code optimization techniques for media processors, which exploit two specific architectural features: conditional instructions and multimedia instructions. We expect that these and similar optimizations will help to take the step from assembly level programming of embedded processors to the use of compilers. Eventually, this will enable a more efficient design of processor based embedded systems.

References

[1] V. Zivojnovic, J.M. Velarde, C. Schläger, H. Meyr: *DSPStone – A DSP-oriented Benchmarking Methodology*, Int. Conf. on Signal Processing Applications and Technology (ICSPAT), 1994

[2] P. Marwedel, G. Goossens (eds.): *Code Generation for Embedded Processors*, Kluwer Academic Publishers, 1995

[3] R. Leupers: *Retargetable Code Generation for Digital Signal Processors*, Kluwer Academic Publishers, 1997

[4] A. Peleg, S. Wilkie, U. Weiser: *Intel MMX for Multimedia PCs*, Comm. of the ACM, vol. 40, no. 1, 1997

[5] Texas Instruments: TMS320C62xx CPU and Instruction Set Reference Guide, URL http://www.ti.com/sc/c6x, 1998

[6] Philips: URL http://www.trimedia.philips.com, 1998

[7] A.V. Aho, M. Ganapathi, S.W.K Tjiang: *Code Generation Using Tree Matching and Dynamic Programming*, ACM Trans. on Programming Languages and Systems 11, no. 4, 1989, pp. 491-516