A PLATFORM-BASED COMPARISON BETWEEN A DIGITAL SIGNAL PROCESSOR AND A GENERAL-PURPOSE PROCESSOR FROM AN EMBEDDED SYSTEMS PERSPECTIVE*

Daniel Lázaro Cuadrado1, Ole Olsen1, Peter Koch1
Allan Frederiksen2, Ole Wolf3, Rasmus Damsgaard Christiansen4

1Embedded Systems Group, CPK, Aalborg University, 9220 Aalborg, Denmark, {kapokasa,oo,pk}@kom.auc.dk
2Nokia Danmark A/S, Allan.Frederiksen@nokia.com
3DSP Assist, wolf@ dspassist.com
4ETI A/S, rasmus@christiansen.dk

ABSTRACT

The number of embedded systems surrounding us increases rapidly. Digital signal processing comprises a significant part of such systems. Digital signal processors have traditionally been used for digital signal processing. On the other hand, General-purpose processors trend to include support for specialized DSP related features in order to improve their performance for specific tasks. In this work, one platform of each kind (TMS320C55x and ARM9E-S) have been selected, analyzed and compared. The platform has been defined as a processor and the associated compiler. Case studies representative of embedded systems applications are compiled and evaluated in a way that allows us to compare them in terms of clock cycles and memory usage. Preliminary results indicate that GPP platforms compare favorably with DSP platforms in their own field.

1. INTRODUCTION

Nowadays, the number of embedded systems having impact on our daily life is increasing quickly. Most of these systems interact with the environment through signals. That requires intensive digital signal processing (DSP) power for such devices. Traditionally, such computations have been performed by a common type of Application-Specific Instruction-set Processors (ASIPs) so called Digital-Signal Processors (DSP) [0]. A DSP is optimized to perform operations on digital signals, usually math-intensive and time-constrained computations.

On the other hand, the amount and complexity of functionality implemented in embedded devices nowadays is increasing. Heterogeneous code-type mixtures must coexist within embedded systems in which DSP code is still a significant part. General-Purpose Processors (GPPs) are suitable for a variety of applications making them appropriate for these heterogeneous mixtures.

Lately, we have witnessed a trend to include different specialized features in GPP processors in order to improve their capabilities for specific tasks which will allow them to compete with DSPs in their own fields (e.g., MAC / Dual MAC support, specialized addressing modes, Viterbi support etc.).

Thus, the objective of this paper is to evaluate the performance of DSPs and GPPs with specific features. In this initial study, we have chosen a DSP and a GPP to which we have applied different case studies in terms of types and sizes, in such a way that we can compare them. We have defined platforms as the processor together with the compiler. Previous studies found a poor compiler-processor interaction for DSP applications [1]. Section 2 of this paper contains a description of the methodology applied. The results obtained are presented in section 3, and the conclusions and directions for future research are discussed in section 4.

2. METHODOLOGY

In this section we present the platforms, the case studies, and the benchmark scenario for the experiments.

2.1. Platforms

The platforms used for the comparison are shown in Table 1. By platform we mean the processor together with an associated compiler.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Texas Instruments TMS320C55x</td>
<td>Version 2.0 released in 2000</td>
</tr>
<tr>
<td>ARM Ltd. ARM9E-S</td>
<td>ARM 1.1 released in 2000</td>
</tr>
<tr>
<td>released in 2000</td>
<td>Thumb 1.1 released in 2000</td>
</tr>
</tbody>
</table>

The TMS320C55x [2] (hereafter C55x) is a multi-issue fixed-point CISC processor which operates primarily on 16 bit data and supports dual (multiply-accumulate) MAC. The C55x processor family is intended for applications that are DSP intensive. It
contains a small and heterogeneous register file and a 7 stage pipeline. The instruction set is large, complex and non-orthogonal, with a variable length instruction word (8-48 bits).

The ARM9E-S [3] is a 32-bit RISC core implementing the ARMv5TE architecture that includes DSP extensions to improve 16-bit fixed-point performance using a single-cycle MAC unit. It is intended for applications requiring a mixture of DSP and micro-controller performance. It contains a large homogeneous register file and a 5 stage pipeline. It supports two instruction sets: ARM and Thumb. ARM is 32-bit while Thumb is a 16-bit subset of ARM on the same 32-bit architecture. They are both highly orthogonal.

The three compilers support ANSI C. The optimization levels achieving the best cycle count performance have been used. Previous experiments [4] were performed in order to investigate the performance of the different compilation options.

2.2. Case Studies

The selection of the case studies is based on different types and sizes of typical functionalities commonly implemented in embedded systems. A new taxonomy for characterizing case studies is presented in Figure 1, which shows that two different orthogonal parameters (type and size) can be considered for every case study. The traditional DSP benchmarking domain and the domain considered for the experiments are depicted as well, showing that different types of code as well as more realistic sizes have been used.

Signal processing applications often use linear algebra and matrix representations. Consequently, two matrix functions have been considered: the calculation of an autocorrelation matrix and a forward substitution. Dhrystone [5] is a synthetic benchmark, written by Reinhold P. Weicker. The version used is 2.1 released in 1988. Dhrystone was written to reflect actual programming practice in systems programming at that time. It contains statements of a high level programming distributed as follows:

- Assignments (51.0%)
- Control statements (32.3%)
- Procedure / function calls (16.7%)

CVSD (Continuous Variable Slope Delta Modulation) [6] is used on the Bluetooth air interface. It is more robust than log PCM for voice over the air interface. We have considered the encoding and decoding of a 64 ksamples/s signal.

The Viterbi algorithm [7] was discovered and analyzed by Andrea Viterbi in 1967. It is mostly used in estimation and detection problems in digital communications and signal processing, mainly to detect signals in communication channels with memory, and to recover coded control sequences with transmission errors. It finds the most probable sequence of symbols based on transitions in a state diagram.

Table 2. Selected case studies including type and size.

<table>
<thead>
<tr>
<th>Case Study</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Functions</td>
<td>DSP</td>
<td>Kernel</td>
</tr>
<tr>
<td>Dhrystone</td>
<td>Control</td>
<td>Kernel</td>
</tr>
<tr>
<td>CVSD</td>
<td>DSP</td>
<td>Application</td>
</tr>
<tr>
<td>Viterbi</td>
<td>Control</td>
<td>Kernel</td>
</tr>
</tbody>
</table>

Regarding types, control code has its origin in finite state machines (FSM). It contains many basic blocks and is branch intensive. It is typically size-constrained. DSP code has its origin in mathematical transformations used in signal processing. It often consists of tight nested loops performing arithmetically intensive functions with high instruction locality. It is typically time-constrained. The term “grey zone” code is used for DSP code which is not time-critical (e.g., various functions with DSP capabilities which are rarely executed).

Signal processing applications (e.g., EFR GSM and CVSD) provide a realistic mixture of different code types. Kernels (e.g., FFT, FIR, LMS, small state machines) refer to size limited code segments. High level language (HLL) constructs are very simple code segments (e.g., pointer addressing, function call and loop constructs). HLL constructs are not representative of real world applications, as the output depends heavily on the context each HLL construct is a part of.

The case studies applied to the platforms are shown in Table 2.
2.3. Benchmark Scenario

The performance of the selected case studies on the selected platforms has been evaluated using the scenario depicted in Figure 2. Cycle count (CC) in cycles and code size (CS) in bytes allow us to compare different platforms independently of their implemented clock speed, instruction length and data types. The reported metrics are shown in Table 3.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC(C55x)</td>
<td>Cycle count for the C55x platform</td>
</tr>
<tr>
<td>CC(ARM)</td>
<td>Cycle count for the ARM platform</td>
</tr>
<tr>
<td>CCF</td>
<td>Cycle count improvement factor</td>
</tr>
<tr>
<td>CS(C55x)</td>
<td>Code size for the C55x platform</td>
</tr>
<tr>
<td>CS(ARM)</td>
<td>Code size for the ARM platform</td>
</tr>
<tr>
<td>CSF</td>
<td>Code size improvement factor</td>
</tr>
</tbody>
</table>

We have chosen to evaluate the differences in performance between the platforms since we do not have an absolute reference point. Therefore, we have chosen to report improvement percentages in order to represent how much the best platform improves over the other. This is calculated as in (1).

\[
Gain = \left( \frac{\text{worst} \times 100}{\text{best}} \right) \times 100 \text{ [\%]} 
\] (1)

This way all gain values are calculated in relation to the worst result. For both cycle count and code size the best result is represented by the smallest number.

3. RESULTS

The improvement factors for the experiments are shown in Table 4. The results for cycle count gain can be seen in Figure 3. The C55x platform performs better than the ARM9E-S in the matrix function experiments with an average improvement percentage around 30%. C55x has support for complex addressing generation and parallel data transfer from memory that makes it potentially superior as compared to the ARM9E-S in this type of experiment.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A C</td>
<td>Auto Correlation matrix calculation</td>
</tr>
<tr>
<td>F S</td>
<td>Forward Substitution vector calculation</td>
</tr>
<tr>
<td>D M</td>
<td>Dhrystone Mark</td>
</tr>
<tr>
<td>C E</td>
<td>CVSD Encode Function</td>
</tr>
<tr>
<td>C D</td>
<td>CVSD Decode Function</td>
</tr>
<tr>
<td>V D</td>
<td>Viterbi Decoding</td>
</tr>
</tbody>
</table>

For the remaining four experiments the ARM9E-S platform shows a gain average of about 25%. The Dhrystone per second marks (following the standard’s requirements) are 390625 for the ARM9E-S and 219482 for the C55x which shows approximately 45% gain of the ARM9E-S over the C55x. For the CVSD encoding and decoding functions the ARM9E-S performs slightly better than the C55x. Finally, the ARM9E-S performs for Viterbi decoding around 45% better. Even though the C55x has hardware support for Viterbi decoding, the compiler is unable to take advantage of it.

The greatest improvements of the ARM over C55x are observed in the control oriented applications.

In Figure 4 code size results are shown. In this case, the C55x achieves smaller code sizes than the ARM9E-S in four experiments out of six with an average improvement percentage of about 40%. For the Dhrystone benchmark and the Viterbi decoding the ARM9E-S performs better with an average around 70%. CISC processors have an advantage over RISC processors in terms of code size [8]. For this reason, it was expected that the C55x platform performed better. In addition, the variable length instruction word potentially enables high code density making the C55x platform potentially superior in this sense. Again, the ARM platform is superior in control oriented applications.

In order to understand the results, we selected two case studies; CVSD and Matrix. CVSD because is the most realistic case study due to its application size and Matrix due to its combination of DSP and control characteristics.

For the remaining four experiments the ARM9E-S platform shows a gain average of about 25%. The Dhrystone per second marks (following the standard’s requirements) are 390625 for the ARM9E-S and 219482 for the C55x which shows approximately 45% gain of the ARM9E-S over the C55x. For the CVSD encoding and decoding functions the ARM9E-S performs slightly better than the C55x. Finally, the ARM9E-S performs for Viterbi decoding around 45% better. Even though the C55x has hardware support for Viterbi decoding, the compiler is unable to take advantage of it.

The greatest improvements of the ARM over C55x are observed in the control oriented applications.

In Figure 4 code size results are shown. In this case, the C55x achieves smaller code sizes than the ARM9E-S in four experiments out of six with an average improvement percentage of about 40%. For the Dhrystone benchmark and the Viterbi decoding the ARM9E-S performs better with an average around 70%. CISC processors have an advantage over RISC processors in terms of code size [8]. For this reason, it was expected that the C55x platform performed better. In addition, the variable length instruction word potentially enables high code density making the C55x platform potentially superior in this sense. Again, the ARM platform is superior in control oriented applications.

In order to understand the results, we selected two case studies; CVSD and Matrix. CVSD because is the most realistic case study due to its application size and Matrix due to its combination of DSP and control characteristics.
advantage of the processors’ conditional instruction execution capabilities. The C55x compiler does not take advantage of the C55x’s parallel instruction execution ability on this benchmark. The ARM9 benefits strongly from its large homogeneous register file characteristic of GPPs, avoiding the C55x’s problem of burning cycles on restoration of its register contents caused by its small, heterogeneous register file. The ARM9 assembly code contains more branches than the C55x, which also takes advantage of its block repeat feature. However, the number of cycles spent on branches on the two processors is about evened out, because the C55x incurs much stiffer branch penalties than the ARM9. In conclusion, the C55x spends more cycles than the ARM9 primarily because the C55x must perform housekeeping tasks restoring register contents. Features that give the C55x potential to outperform the ARM9, such as parallel instruction execution, are not exploited by the compiler.

Regarding Matrix, in the auto correlation matrix calculation, the compiler takes advantage of the ARM’s stack-relative addressing. Also, the compiler utilizes the ARM’s ability to perform a barrel shift of an ALU output before the output is assigned to a register, which helps generate the correct memory alignment of array elements with almost no overhead. The computations involving complex data are performed quite efficiently, taking advantage of the ARM’s support for 16 bit loads and its multiply-add instructions. However, the compiler generates rather inefficient indexing into multi-dimensional arrays. The two-dimensional arrays could be treated as a linear array by the compiler on this benchmark, reducing in this way the time consuming recalculation of multiple indexes. The poor indexing of multi-dimensional arrays is the major contributor to the cycle count of this function. The C code for the forward substitution calculation assumes an underlying processor architecture that supports 16-bit arithmetic only, and includes a significant amount of 32-bit arithmetic emulation. The ARM has native support for 32-bit arithmetic, but the compiler does not recognize that several 16-bit operations in a row in the C code can be combined into a single 32-bit assembly language instruction. As a result, the ARM does a significant amount of unnecessary work emulating 32-bit arithmetic.

The second part of the forward substitution calculation uses support functions for 32-bit multiplications and saturation commonly found in reference code for, e.g., the GSM vocoder. The compiler does not translate these functions into the ARM’s native 32-bit arithmetic instructions, but instead translates them into function calls. The ARM9 incurs significant function overhead on these function calls. For the C55x and the Matrix case study, the compiler generates several unnecessary move instructions for each multiplication.

In the forward substitution calculation, two right shifts of a 40-bit value by an int value (which is 16 bits) are implemented as library function calls, but they could be implemented with a single assembly instruction each. The results for the forward substitution suffer from the fact that most calculations are done in extended precision, both additions (40 bits) and multiplications (32x32 → 32 bits).

Another study was performed in order to compare both instruction sets of the ARM9E-S platform. For that purpose, all case studies were re-compiled and measured again. A comparison of the results can be seen in Figure 5. The Gain formula in (1) has been used again in (2) and (3). The Thumb instruction set is expected to yield a higher cycle count and code density.

\[
Gain \, CC = \left( \frac{CC(Thumb)}{CC(ARM)} \right) \times 100 \% \quad (2)
\]

\[
Gain \, CS = \left( \frac{CS(ARM)}{CS(Thumb)} \right) \times 100 \% \quad (3)
\]

Results show an average code size improvement of about 25% but at the cost of an average increase in cycle count of about 85%. Differences vary a lot depending on the type of experiment we are considering. For the Dhrystone case study, an approximately 30% improvement in code size is achieved by using the Thumb instruction set, but only producing an about 20% increase in cycle count. However, for the forward substitution function, almost a 40% improvement in code size is achieved but at the cost of almost twice the cycle count.

There are three main reasons for the significant cycle count increases when the Thumb instruction set is used. First, branches are more limited in range (particularly conditional branches) and branches to subroutines are unconditional. This mostly affects control oriented code. Second, data processing instructions are fewer (e.g., only one multiplication instruction exists in the Thumb instruction set while 14 exists for ARM). In many cases, the result of the operation must be stored in one of the operand registers. They have limited access to 8 of the 16 registers. This affects mainly arithmetic-intensive code. Third, single and multiple load and store
instructions can only access 8 registers. This produces an increase in memory accesses and also an overhead in move instructions.

It is often the case that a small fraction of a program accounts for most of the running time [9]. In many cases 20% of the code takes 80% of the cycle count, as a rule of thumb. We can consider a partition in which 20% of the code is assumed to be time-constrained and the rest size-constrained. For the ARM platform both instruction sets can be used together in the same program. We can use the ARM instruction set for the time constrained code and the Thumb instruction set for the size constrained code. Using the averaged numbers obtained in the instruction set comparison we could expect a reduction of about 20% in code size and an increase in cycle count of about 15%. For partitions around 10/90 and 20/80, the combination of both instruction sets would produce significant reductions in code size with a small overhead in cycle count.

4. CONCLUSION AND FURTHER RESEARCH

For the selected case studies, the results indicate that the ARM platform in most cases performs better than the C55x platform regarding cycle count. For code size, the C55x performs better (control oriented studies being the exception). For control applications, the ARM platform is superior to C55x platform. The ARM and Thumb instruction sets comparison reveals that the Thumb instruction set improves code density, as was expected. The results indicate that an average reduction in code size of around 30% is likely with the Thumb instruction set but at the cost of an average increase in cycle count of nearly 80% depending on the type of code considered.

Experiments also showed that compilers are not always able to exploit specialized architectural features (such as hardware looping, Viterbi support, MAC, dual MAC, circular buffers etc.) from the source code. A common trend among DSP processors is to use increasingly complex architectures to improve performance. This requires either a very advanced compiler technology or very skilled programmers in order to take advantage of most of the potential architectural power. The more complex the architecture, the more complex the compiler needs to be. Simple RISC-like processors make the compiler’s task easier.

From the results obtained we conclude that GPPs (with application specific extensions) are now a viable alternative to ASIPs. The ARM platform has shown good performance for some DSP applications—even better than the C55x—while for control tasks it was superior.

The growing need for code reuse makes the ability of C-compilers to exploit processor’s features a key issue. Different C extensions (intrinsics) exist for many embedded processors, but in order to retain reusability across platforms, a standardized set of intrinsics would be desirable and this requires more research.

Considering the increasing size of embedded applications, including various heterogeneous mixtures, GPPs [10] become a good choice for future embedded systems.

REFERENCES