On Efficiency of Transport Triggered Architectures in DSP Applications

JARI HEIKKINEN[1], JARMO TAKALA[1], ANDREA CILIO[2], and HENK CORPORAAL[3]
[1]Tampere University of Technology, P.O.B. 553, 33101 Tampere, FINLAND
[2]Delft University of Technology, Mekelweg 4, 2628 CD Delft, THE NETHERLANDS
[3]IMEC, Kapeldreef 75, B-3001 Leuven, BELGIUM
jari.heikkinen@tut.fi

Abstract: - The trend in programmable architectures for digital signal processing (DSP) is to move towards high-level language programming, which sets high requirements for compilers to efficiently exploit the instruction level parallelism in modern processors. In this paper, efficiency of transport triggered architectures (TTA) in DSP applications is discussed. The efficiency of a high-level compiler on a TTA is compared to commercial very long instruction word DSP architecture. The effect of different coding styles in high-level language code is evaluated with a DSP benchmark, fast Fourier transform.

Key-Words: - transport triggered architecture, customizable processor architecture, digital signal processing, fast fourier transform, coding style, performance evaluation

1 Introduction

The current trend in programmable architectures in the field of digital signal processing (DSP) is to move towards high-level language (HLL) programming and customizable architectures [1]. The reason behind this is the gap between the productivity of designers and increased complexity of DSP applications. In order to improve the productivity even further, several design methodologies suggest tool assisted HLL code generation. Since the coding style of generated HLL may vary from tool to tool, insensitivity to coding style variations is a good property in HLL compilers used to generate the machine code for the target architecture.

Due to the real-time requirements of DSP applications, the performance of DSP processors is improved by exploiting the instruction-level parallelism. Currently very long instruction word (VLIW) architectures have gained popularity in DSP applications. In general, the VLIW architectures are intended to be programmed with HLLs, thus they lend themselves to programmable DSP processor trends.

VLIW architectures are modular; the number of function units (FU) can be increased. There are even VLIW architectures, which support customized, application-specific function units. In VLIW, this may, however, restrict the flexibility, e.g., in Trimedia [2], support for multi-operand instructions, i.e., multi-operand FU, reserves several instruction fields from the VLIW instruction. VLIW architectures have been criticized for their requirements for read/write ports in the register file [3] and, in order to alleviate this problem, clustered approach has been suggested, i.e., the register file is partitioned as illustrated in Fig. 1. In addition, the complexity of the bypassing network and the register file is high, since the bypassing network must support bypassing of operands from all FU outputs to their inputs.

An alternative architecture where the drawbacks of VLIW architecture can be avoided is transport triggered architecture (TTA) [4]. In TTA, a program describes only the operand transfers between the computational resources in the architecture. Such a mirrored programming paradigm allows new scheduling and allocation techniques to be employed in HLL compilers. TTA concept supports heterogeneous, even multi-operand FUs without restrictions. Thus, TTA is a promising concept for embedded DSP applications where customization is desired.

In this paper, performance of TTA and VLIW architectures is compared in a DSP application, namely fast Fourier transform (FFT). The application is described with C language and HLL compilers are used to generate parallel code for the architectures. Performance and efficiency is measured in terms of clock cycles and code density. Furthermore, the effect of different HLL coding styles have been analysed in order to estimate the performance in cases where tool assisted HLL code generation is used.

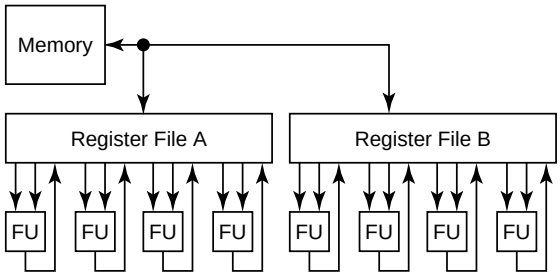

Fig. 1: Principal block diagram of clustered VLIW. FU: Function unit.

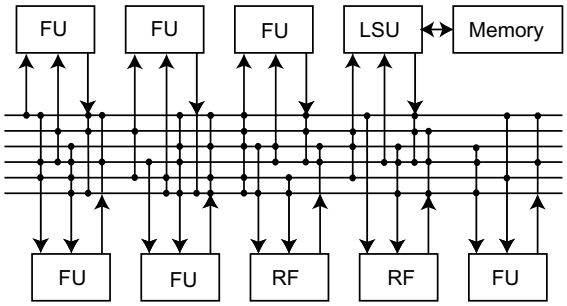

Fig. 2: Principal block diagram of TTA. FU: Function unit. RF: Register file. LSU: Load-store unit. Dots represent socket connections.

2 Transport Triggered Architecture

The bypass complexity can be reduced by making the bypass registers visible at architectural level. This way the spilling of bypass values into register file (RF) is made under program control. The bypass complexity can also be reduced by reducing the number of read and write connections and by reducing the number of bypass buses. This implies that besides the operations also the operand transfers (transports) need to be scheduled at compile-time as is done in the case of RFs. Thus, the bypass transports become visible at the architectural level. This implies that operations can be hidden. In this model, the data transports trigger the FU operations implicitly. Thus, the programming paradigm reminds data flow machines. In principle, the traditional operation triggered programming paradigm is mirrored, hence the name transport triggered architecture [4]. In the TTA programming model, program specifies only the data transports to be performed by the interconnection network. Therefore, only one type of operation is supported: move operation, which performs a data transport from source to destination. The number of move operations per instruction is equal to the number of simultaneous transports supported by the interconnection network.

A TTA processor consists of a set of functional units and register files containing general-purpose registers. These units are connected by interconnection network consisting of buses as illustrated in Fig. 2. Connections to buses are established through input and output sockets. An input socket contains multiplexers feeding operands from the buses into the FUs. An output socket contains de-multiplexers placing the FU results into the correct bus. The number of FUs and RFs nor the input and output connections of FUs and RFs are limited. The TTA concept provides flexibility in form of modularity; functional units with standard interface are used as basic building blocks. Therefore, the architecture can be tailored with special function units without need to change the transport capacity.

MOVE framework is a design environment containing a set of software tools for designing application-specific instruction set processors that utilize the TTA paradigm [5]. MOVE framework provides semi-automatic design process that shortens the design-time. The design flow in MOVE framework consists of three principal components as illustrated in Fig. 3. The design space explorer searches the design space for a processor configuration, which yields the best cost/performance ratio for a given application. The hardware subsystem generates structural hardware description of the selected processor configuration and produces statistics of timing, area, and power consumption based on information in technology libraries. The software subsystem generates instruction-level parallel code for the selected processor configuration and provides statistics, e.g., on cycle count, instruction count, and hardware resource utilization.

The MOVE framework supports all the trends in DSP processors mentioned previously: customization, HLL programming, and instruction-level parallelism. Designer can tailor and optimize the resources of the programmable architecture and the HLL compiler adapts to these modifications. This approach allows processor core to be customized according to the requirements of the application in hand.

3 Benchmark Application

A popular benchmark application for programmable DSP processors is fast Fourier transform (FFT). Here an in-place FFT algorithm has been used as they are often preferred in software implementations since at each recursion the results can be stored into the same memory locations as the input operands. The used algorithm is the decimation-in-place radix-2 Cooley-Tukey FFT [6] and an N-point algorithm, $N = 2^n$, con-

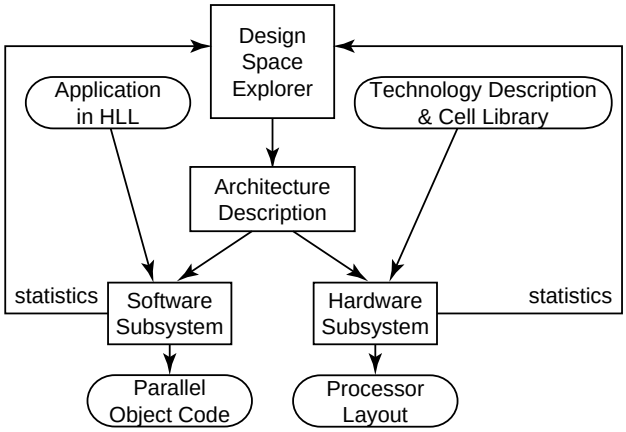

Fig. 3: Principal design flow in MOVE framework.

taining n iterations of radix-2 FFT butterfly operations can be defined as

$$\begin{cases} X_m(p) &= X_{m-1}(p) + X_{m-1}(q) \\ X_m(q) &= (X_{m-1}(p) - X_{m-1}(q))\, W_{2^m}^{\lfloor p \bmod 2^m \rfloor} \end{cases}$$
$$q = p + 2^{m-1};\ p = i + k2^m;$$
$$i = 0, 1, \ldots, 2^{m-1} - 1;\ n = 0, 1, \ldots, 2^{n-m} - 1 \qquad (1)$$

where mod is the modulus operator and $X_m(p)$ is the pth complex-valued operand at iteration m, $0 < m \geq n$. The operands $X_0(p)$ are the input sequence values in bit-reversed order and $X_n(p)$ are the final results in order. W_N^r is the complex-valued twiddle factor defined as

$$W_N^r = e^{-j2\pi r/N} \qquad (2)$$

where j is the imaginary unit. Direct implementation of the complex multiplication requires four real-valued multiplications. However, by utilizing the periodicity of sine and cosine functions, this can be realized with three real-valued multiplications as described in [7].

The FFT algorithm was described in C language using fractional data representation, i.e., fixed-point representation where the number range is normalized. Fractional representation is often used in DSP realizations and it represent also challenges for C compilers due to the fact that ANSI C does not contain predefined data type for fractional representation. Due to the in-place structure of FFT, three nested loops had to be used in the code.

The main objective was to investigate the effect of different loop structures on the performance. Five different C codes of the FFT were programmed. The first version, *code A*, was written without any attention to the coding style. The application code included both `FOR` and `WHILE` loops with loop indices iterating downwards and upwards. Twiddle factors and data elements in the input sequence were accessed with the aid of pointers. In *code B*, only `FOR` loops were used since, in general, compilers perform better with `FOR` loops. In *code C*, the loop indices in all the loop structures, still `FOR` loops, were iterating downwards since some compilers prefer iterations in decreasing order. The nested loops of the original code were removed in *code D*, since they may produce inefficient code in some compilers. In addition, a special experiment was made to investigate the use of pointers. In *code E*, data elements in vectors are accessed with indexing and all the pointers are removed.

4 TTA and VLIW Comparison

For the comparison of VLIW and TTA, the benchmark was compiled with C compiler for a commercial clustered VLIW processor, Texas Instruments TMS320C6203 [8]. This is realized with 0.15 μm technology and the maximum clock frequency is 300 MHz [9]. For TTA, we used the development tools from the MOVE framework. Unfortunately, there is no TTA processor implemented with comparable technology as the VLIW processor. The technology libraries for the TTA design space explorer are based on 0.7 μm standard cell technology. The explorer estimates that the maximum clock frequency for the TTA architectures used in the evaluation are between 25-27 MHz.

Due to the technolgy difference, the evaluation is done by comparing the number of clock cycles. TTA designs are configured to contain the same number of computational resources as in the used VLIW processor; eight functional units from which two are multipliers and six ALUs. Furthermore, the TTA architecture was configured to contain 32 general-purpose registers partitioned into two register files as in the reference VLIW.

The multipliers in the VLIW architecture have an internal pipeline stage and, therefore, pipelined multipliers are used also in TTA. In the VLIW architecture, load from memory to a register takes four cycles. The address computation is performed in an ALU, which is used also for signal processing. This implies that the ALU has different latencies depending on the operation. Variable latency is not allowed in TTA, thus direct modeling of load operation cannot be done. Due to this, a separata unit was added to perform load and store operations.

The performance statistics of VLIW were measured using the instruction set simulator incorporated into the development tools of the VLIW processor. The statistics for TTA processor were obtained from the

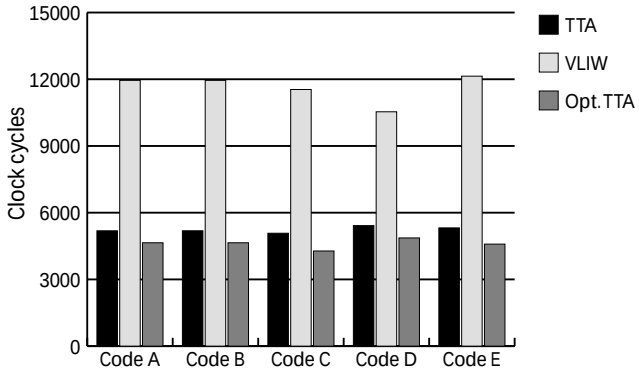

Fig. 4: Cycle count comparison of TTA and VLIW compilers.

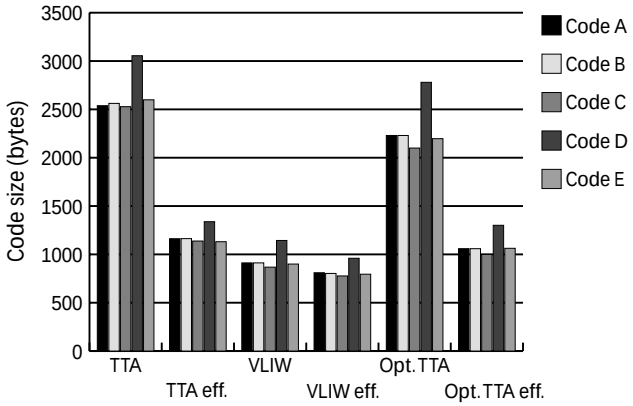

Fig. 5: Code size comparison of TTA and VLIW.

parallel simulator of the MOVE framework invoked in the scheduling phase of the compilation. The number of clock cycles elapsed in the FFT application with different coding styles in VLIW and TTA is shown in Fig. 4. Different coding styles do not have a major effect on the number of clock cycles in either architecture. For TTA, the maximun difference in the number of clock cycles is 7 percents whereas for VLIW it is 14 percents. The best cycle count for TTA is obtained when all the loops are configured to iterate downwards (*code C*) whereas VLIW compiler performs best for code containing few or no nested loops (*code D*).

VLIW processor requires at least twice as many clock cycles to perform the FFT application than TTA. The reason behind this is that the initial application was written without much concern on the coding style thus resembling more general-purpose code than digital signal processing code. Since the compiler of TTA is tuned for general-purpose processing, it was able to compile the application better than the VLIW compiler tuned for digital signal processing. By writing the application carefully, the compiler of VLIW could utilize software pipelining and other advanced code compilation techniques thus resulting in comparable or even better performance, since the compiler of TTA does not yet support these techniques. However, if the objective is to move towards automated code generation, TTA seems to be a good solution, since it can detect the parallelism from even a poorly written code.

In embedded DSP systems, the code density is an important aspect. The effect of coding styles to the resulting code size is illustrated in Fig. 5. Effective code sizes, where NOP-instructions have been removed, are also shown in the figure. The largest difference in code size in TTA is approximately 21% whereas in VLIW it is approximately 32%. The smallest code size in both architectures was obtained with the code where iterations were performed downwards (*code C*). The largest code size was obtained when nested loops were removed (*codeD*). The code sizes for VLIW are about three times smaller than for TTA. This is because the architecture generated by the MOVE tools does not contain any instruction compression, i.e., each parallel move operation is directly stored into the instruction memory. Furthermore, the number of NOP-instructions in TTA code is large thus increasing the code size considerably as can be noted by comparing the total code size with the corresponding effective code size. In VLIW the increase to the code size caused by NOP-instructions is fairly small. The code density of TTA could improved by utilizing code compression techniques such as entropy coding. These methods can result in notable reduction in code size since, in TTA, code compression can be made individually for each application.

Finally, the design space explorer was used to find an area-efficient architecture for each coding style case. In all cases, the optimized architecture obtained from the design space explorer contains one multiplier, two ALUs and two register files with 16 registers each. These architectures are referred to as optimized TTA in Fig. 4 and 5. The cycle counts for the optimized TTA design are in all cases smaller than for the fixed resources TTA. This is due to the optimized resources for each of the coding style cases. In addition, the code sizes for the optimized TTA are smaller than in the fixed resource TTA. This is due to the fact that the less resources, the smaller the instruction word width. Thus, it can be stated that the explorer works efficiently in finding an optimized architecture configuration to fullfill the constraints of the application.

5 Conclusions

In this paper, performance of transport triggered architecture has been compared to clustered VLIW in cases where the DSP application is described with a high-level language. It was noted that the TTA approach is a good candidate future DSP implementations where HLL compilation and customization according to application are needed. It was noted that the bottleneck of used TTA architecture is the code density and, therefore, further investigation is needed to improve this property in TTA.

References:

[1] J. M. Rabaey, W. Gass, R. Brodersen, T. Nishitani, and T. Chen, "VLSI design and implementation fuels the signal-processing revolution," *IEEE Signal Processing Mag.*, vol. 15, no. 1, pp. 22–37, Jan. 1998.

[2] J. T. J. van Eijndhoven, F. W. Sijstermans, K. A. Vissers, E. J. D. Pol, M. I. A. Tromp, P. Struik, R. H. J. Bloks, P. van der Wolf, A. D. Pimentel, and H. P. E. Vranken, "TriMedia CPU64 architecture," in *Proc. IEEE Int. Conf. Computer Design*, Austin, TX, U.S.A., Oct. 10–13 1999, pp. 586–592.

[3] R. P. Colwell, R. P. Nix, J. J. O'Connel, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," *IEEE Trans. Comput.*, vol. 37, no. 8, pp. 967–679, Aug. 1988.

[4] H. Corporaal, *Microprocessor Architectures: From VLIW to TTA*, John Wiley & Sons, Chichester, UK, 1997.

[5] H. Corporaal and M. Arnold, "Using transport triggered architectures for embedded processor design," *Integrated Computer-Aided Engineering*, vol. 5, no. 1, pp. 19–38, 1998.

[6] A. V. Oppenheim and R. Schafer, *Discrete-Time Signal Processing*, Prentice Hall, Inc., Englewood Cliffs, NJ, U.S.A., 1989.

[7] A. Wenzler and E. Lüder, "New structures for complex multipliers and their noise analysis," in *Proc. IEEE Int. Symposium on Circuits and Systems*, Seattle, WA, U.S.A., Apr. 30–May 3 1995, vol. 2, pp. 1432–1435.

[8] N. Seshan, "High velociTI processing," *IEEE Signal Processing Mag.*, vol. 15, no. 2, pp. 86–101, 117, Mar. 1998.

[9] *TMS320C6203 fixed-point digital signal processor*, data sheet, Texas Instruments Inc., Houston, TX, U.S.A., 2000.