FSEL - Selective Predicated Execution for a Configurable DSP Core

C. Panis
Carinthian Tech Institute
c.panis@cti.ac.at

U.Hirnschrott, A.Krall
Vienna University of Technology
uli@complang.tuwien.ac.at
andi@complang.tuwien.ac.at

G.Laure, W.Lazian
Infineon Technologies Austria
gunnar2@sbox.tu-graz.ac.at
lazian@sbox.tu-graz.ac.at

J. Nurmi
Tampere University of Technology
jari.nurmi@tut.fi

Abstract

Increasing system complexity of SOC applications leads to an increased need of powerful embedded DSP processors. To fulfill the required computational bandwidth, state-of-the-art DSP processors allow executing several instructions in parallel and for reaching higher clock frequencies they increase the number of pipeline stages. However, deeply pipelined processors have drawbacks in the execution of branch instructions: branch delays. In average not more than two branch delay slots can be used, additional ones keep unused and decrease the overall system performance. Instead of compensating the drawback of branch delays (e.g. branch prediction circuits) it is possible to reduce the number of branch delays by reducing the number of branch instructions. Predicated execution (also guarded execution or conditional execution) can be used for implementing if-then-else constructs without using branch instructions. The drawback of traditional predicated execution is decreased code density. This paper introduces selective predicated execution based on FSEL which allows reducing the number of branch instructions without decreasing code density. Selective predicated execution based on FSEL is part of a project for a configurable DSP core.

1. Introduction

Increasing system complexity of SOC applications leads to an increasing demand on computational power of embedded processors. Deep pipelined processors are used for reaching higher clock frequencies. But deep pipelined processors have obstacles when executing branch instructions: *branch delays* [1].

Branch delays are caused by taken branch instructions which cause a break in the linear program flow. *Branch delays* can lead to significant performance lack of the processor sub-system. To overcome this drawback branch prediction circuits have been introduced [2][3][4][5]. Especially in the area of DSP algorithms deterministic behavior is required, which contradicts prediction approaches. During system definition the worst case execution time has to be considered and the prediction assumed as not to be taken. Therefore prediction has no added value for system performance. Another approach for reducing the number of branch delays is reducing the number of branch instructions. Predicated execution can be efficiently used to remove conditional branch instructions caused by *if-then-else* constructs. Predicated or conditional execution has already been introduced in the 80's. The main drawback of predicated execution is an increased program code space. This paper introduces *selective predicated execution* based on FSEL enabling a reduced number of branch instructions without the drawback of increased code space. Only code sections which can make use of the advantage of selective predicated execution need additional instruction space. The chosen orthogonal implementation of FSEL can be efficiently used by a C-Compiler.

The first part of the paper is used for illustrating the motivation of using predicated execution. The second part introduces two implementation examples of predicated execution (Texas Instruments C'62xx, Starcore SC140). The third section is used for introducing selective predicated execution based on FSEL. The result section contains some benchmark results comparing algorithm implementations using selective predicated execution.

2. Motivation

This section describes the branch delay problem caused by branch instructions in deeply pipelined processors. The number of branch delays depends on the number of pipeline stages located between the instruction fetch stage and the branch condition evaluation. Two possible solution approaches are

Proceedings of the IEEE Computer Society Annual Symposium on VLSI Emerging Trends in VLSI Systems Design (ISVLSI'04)
0-7695-2097-9/04 $20.00 © 2004 **IEEE**

shortly explained in this section: Branch prediction and predicated execution.

Today's VLIW (Very Long Instruction Word) DSP processors provide additional computational bandwidth by supporting the execution of several instructions in parallel and by increasing the possible clock frequency due to deep pipeline structures. Whether an application can make use of the provided parallelism is mainly influenced by data dependence between instructions and by the branch instruction frequency. Less branch instructions lead to longer basic code blocks and therefore to a higher possibility to schedule instructions in parallel (increased instruction level parallelism). In [6][7][8] the branch frequency of benchmark examples is analyzed. The ratio is different for scientific code and general-purpose programs (GP). In average general purpose programs have a branch ratio between 20-30%, for scientific code it is still 5-10%. Even for scientific programs (which will be more significant for programs running on an embedded DSP core) each 10^{th} to 20^{th} instruction is a branch instruction. The ratio between conditional and unconditional branches is about 75% conditional branch instructions. Assuming a processor providing several parallel units, the distance between branch instructions is getting quite low. Therefore branch delay slots will consume a significant number of cycles.

One way to reduce the penalty of branch delays is the usage of branch prediction. Grohoski [9] divides conditional branch instructions into loop closing branch instructions (e.g. caused by while loops) and other conditional branches. Loop closing conditional branches will be taken for n-1 times. Assuming that the remaining conditional branches will be taken with 50% probability, this leads to a ratio of 5/6 to 1/6 between taken and not taken branch instructions. Other literature sources [6][10] estimate a ratio of ¾ to ¼, which still justifies the emphasis on the effective implementation of the taken branch instructions.

There are several branch prediction implementations available, getting trickier as the number of pipeline stages is increasing [2][3][4][5]. However, this is not in the focus of this paper.

Another possibility to overcome the problem of unusable branch delays is predicated or guarded execution. It can be used to eliminate conditional branch instructions e.g. generated by *if-then-else* constructs, which are common in control code. It consists of a condition part and an operation part:

(condition) operation

Already in the HP Precision Architecture (1985) conditional execution has been introduced. To quantify the advantage of this architectural feature Pnevmatikatos and Sohi [11] have analyzed benchmark programs (including Espresso, Gcc and Yacc). About 20% of the instructions have been conditional and 5% unconditional branch instructions.

In their study they distinguish between fully guarding which assumes that all instructions can be executed conditionally, and restricted guarding which enables only to execute arithmetic instructions under certain conditions. Detailed results can be found in [11]. For these benchmark examples about one-third of the conditional and unconditional branches can be replaced using full guarding. For restricted guarding the numbers are lower: about 15% of the conditional and 2% of the unconditional branch instructions can be replaced.

The drawback of guarded execution is the growth of the basic block size. In the above discussed benchmark examples the size of the basic blocks increases from 4.8 to 7.3 instructions for full guarding. Using restricted guarding the enlargement is quite less.

Today most of the VLIW architectures support a mechanism of guarded execution. This is mainly influenced due to the aspect that VLIW architectures support a high ILP (instruction-level parallelism), which requires effective branch handling to prevent severe performance limitation.

3. Predicated Execution

This section is used to illustrate implementation examples of predicated execution. The Texas Instruments C62x family and the Starcore SC140 have been chosen. Both VLIW DSP architectures provide the possibility to execute several instructions in parallel, and therefore predicated execution is mandatory to prevent a performance lack in code sequences with high branch frequency.

3.1. TI C62x

The C62x architecture of Texas Instruments supports each instruction to be executed conditionally (full guarding) [12]. To obtain full guarding, 3 bits of each instruction word are used to decode the register whose status is needed to generate the condition. The possible registers are B0, B1, B2, A1 and A2. Under certain conditions A0 can also be used. The remaining coding space (with 3 bits it is possible to encode 8 states) is used to encode unconditional execution and one code combination keeps reserved.

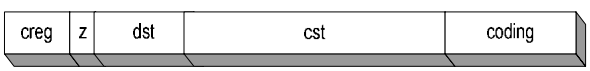

Figure 1: TI C62x instruction example (addk)

The instruction example in Figure 1 shows the leading three bits labeled *creg* used to code one of the registers. The *z* bit following the *creg* is used to decode, whether the test takes place for *equal to zero* (z=1) or *not equal to zero* (z=0).
Each instruction consumes 4 bits to code the condition for the predicated execution, which has influence on the code density. The implementation to encode static registers is useful for the scheduler of the C-Compiler which has a certain freedom of reordering instructions, which is especially necessary for a VLIW architecture supporting the execution of several instructions in parallel. The limitation on a few registers of the register-file supporting predicated execution leads to a restricted use of these registers.

3.2. Starcore SC140

The architecture of the SC140 supports full guarding [13]. Instead of spending the code to each instruction the prefix (already used to build up the execution bundle) is used to code the condition. Execution bundle are those instructions executed during the same clock cycle. There are two subsets per execution bundle possible (even and odd). In the assembly syntax, three instructions are available. IFF is used for instructions of the current set which will be executed, if the flag T is equal to zero. If T is one, the instructions are handled as NOP. The IFT instruction is used for the inverse function. If T is equal to one, the instructions will be executed, if T is equal to zero the instructions will be treated as NOP instructions. The IFA is used for instructions of the same execution bundle, which are executed unconditionally.

The predicated execution implementation of the Starcore SC140 consumes less code space for implementing predicated execution, but the limitation on the status of T leads to a significant limitation for efficient instruction scheduling.

4. Selective Predicated Execution

Selective predicated execution based on FSEL is implemented as separate instruction, which enables to execute the instructions of the same execution bundle conditionally (as illustrated in Figure 2). Therefore the disadvantage of additional coding space (as pointed out in section 2) is restricted to sections, where predicated execution provides added value.

4.1. Architecture

Referring to section 2, the proposed concept is supporting partial guarding, which means that not all of the instructions can be executed conditionally. Different to the definition of partial guarding by Pnevmatikatos and Sohi [11], all instructions with exception of the program flow instructions can be conditionally executed. The FSEL instruction is part of the program flow execution slot and therefore no program flow instruction can be part of the execution bundle. To enable conditional branch instructions the condition is coded in the instruction word itself.

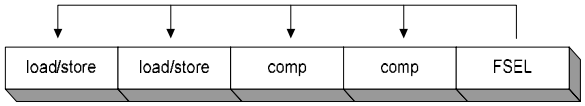

Figure 2: Influence of FSEL on instruction slots

In Figure 2 the influence of the FSEL instruction is illustrated. The FSEL instruction contains the execution condition for the instructions of the same execution bundle. However, not all of the instructions of the execution bundle have to be executed conditionally. Therefore the FSEL instruction supports coding space to enable unconditional execution of instructions in parallel (don't care section).

4.2. Code example

The code example in Figure 3 illustrates the feature of the FSEL instruction. An *if-then-else* construct is well suited for this purpose. If the condition is true, the first instruction shall be executed, if not then the second one. On the right side in Figure 3 the related assembly code for the chosen DSP concept can be seen. Assuming a five stage pipeline, two branch delays will decrease the system performance. In the example of Figure 3 the worst case scenario has been pointed out: none of the available branch delays caused by branch instructions can be filled with useful instructions (NOP instructions are inserted). Assuming the *if-then-else* construct in Figure 3 as part of a longer code section, some of the branch delays get filled with instructions executed anyway. In this example the available resources of the DSP core cannot be used. Therefore the short program sequence has to be executed sequentially.

```
If (d14=0) then                breq d14, true_cond
    adds a0,a2,a2              nop ;branch delay
  else                         nop ;branch delay
    adds a1,a2,a2              adds a1,a2,a2
                               br next_steps
                               nop ;branch delay
                               nop ;branch delay
                          true_cond: adds a0,a2,a2
                          next  step: ...
```

Figure 3: Code example

Using FSEL the if-then-else construct can be coded within one assembly line and executed within one clock

cycle, as illustrated in Figure 4. The dc (don't care) section is not used for this example but can be used for instructions executed unconditionally.

```
fsel d14=0, true: adds a0,a2,a2, false: adds a1,a2,a2, dc:
```

Figure 4: Code example using FSEL

Besides increasing code density (no NOP instructions are inserted), the number of execution cycles can be reduced. Both aspects have influence on the power dissipation of the DSP subsystem. Fetching fewer instructions reduces the switching activity at the program memory port. Less cycles for executing a program reduces the required clock frequency.

5. Results

In Table 1 some benchmark examples are illustrated. The first column contains the name of the chosen algorithm. The remaining columns contain relative numbers in %. The benchmark results are generated once without using predicated execution and once making use of selective predicted execution.

algorithm	Nr.of bundle (%)	Nr. of branch delay NOPS (%)	Code size (%)
Blowfish	98,4	80,8	99,2
Dspstone	98,8	94,6	100,1
Efr	91,0	76,6	98,3
Huffmann	88,6	79,9	99,3
Serpent	95,6	88,9	100,9

Table 1: Benchmark results

The results in the table indicate a reduction of execution bundles and a reduction of necessary branch delay NOPs by using selective predicated execution. The influence on code density is neglectable. Thus, the use of selective predicated execution based on FSEL allows increasing system performance by reducing the number of branch delays without decreasing code density.

6. Acknowledgement

The work has been supported by European Commission with the project SOC-Mobinet (IST-2000-30094) and the CDG Gesellschaft.

7. Conclusion

Predicated execution can be used to reduce the number of branch delays by reducing the number of branch instructions. The number of execution cycles can be decreased (less branch delays) which reduces the required clock frequency for executing an algorithm. A reduced number of branch delay NOPs leads to reduced switching activity at the program memory port. Lower clock frequency and less switching activity at the program memory port decrease the power dissipation of the DSP subsystem.

Traditional implementations of predicated execution feature poor code density. Selective predicated execution as introduced in this paper provides the advantages of predicated execution by reducing the number of unused branch delays, without decreasing code density. Selective predicated execution based on FSEL is part of a project for a configurable DSP core.

8. References

[1] P.Lapsley, J.Bier, A.Shoham and E.A.Lee, *DSP Processor Fundamentals, Architectures and Features*, IEEE Press, New York, 1997.

[2] Smith J.E., *A study of branch prediction strategies*, in Proc. 8th ISCA, pp.135-48, 1981.

[3] Albert D. and Avnon D., "Architecture of the Pentium Microprocessors", IEEE Micro, June 1993.

[4] Heinrich J., *MIPS1000 Microprocessor Users Manual Alpha Draft 11.Oct*, MIPS Technologies Inc., Mountain View. Ca, 1994

[5] Motorola Inc., *Power PC620 RISC Microprocessor Technical Summary*, MPC 620/D, Motorola Inc., 1994

[6] Lee J.K.F. and Smith A.J., "Branch prediction strategies and branch target buffer design", *Computer 17(1)*, pp.6-22, 1984.

[7] Stephens C.,Cogswell B. Heinlein J., Palmer G. and Shen J.P., *Instruction level profiling and evaluation of the IBM RS/6000*. In Proc. 18th ISCA, pp.137-46, 1991.

[8] Yeh T.-Y. and Patt Y.N., *Alternative implementations of two-level adaptive branch predictions.* In Proc. 19th ISCA, pp.124-34, 1992.

[9] Grohoski G.F., "Machine organization of the IBM RISC System/6000 processor", *IBM J.Res. Develop.*, 34(1), Jan., 37-58, 1990.

[10] Edenfield R.W., Gallup M.G., Ledbetter Jr., W.B., Mc.Garity R.C., Quintana E.E. and Reininger R.A., "The 68040 processor", *IEEE Micro*, pp. 66-78, Feb. 1990.

[11] Pnevmatikos D.N. and Soshi G.S., "Guarded Execution and branch prediction in dynamic ILP processors", In Proc. 21st ISCA, pp. 120-9, 1994.

[12] Texas Instruments, *CPU and Instruction Set Reference Guide*, SPRU189B, Texas Instruments, July 1997.

[13] Motorola Inc. and Lucent Technologies Inc. *SC140 DSP Core Reference Manual*, MNSC140CORE/D, Rev.0, 12.1999.