2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing

Trading Conditional Execution for More Registers on ARM Processors

Huang-Jia Cheng*, Yuan-Shin Hwang*, Rong-Guey Chang†, and Cheng-Wei Chen‡
* *Department of Computer Science and Information Engineering*
National Taiwan University of Science and Technology
Taipei 106, Taiwan
†*Department of Computer Science*
National Chung Cheng University
Chia-Yi 621, Taiwan
‡*Processor R&D*
Marvell Technology Group Ltd.
Hsinchu 300, Taiwan

***Abstract*—Conditional execution is an important ISA feature of the ARM series of processors. Every instruction can be made to execute conditionally, that is, it is treated as a NOP if the condition is not met. The advantage of conditional execution is that it can maintain high performance while reducing hardware complexity since it can avoid introducing pipeline bubbles even when no branch prediction units are needed. However, conditional execution takes up precious instruction space as conditions are encoded into a 4-bit condition code selector on every 32-bit ARM instruction. Besides, only small percentages of instructions are actually conditionalized in modern embedded applications, and conditional execution might not even lead to performance improvement on modern embedded processors. This paper proposes to trade conditional execution for more ISA registers on ARM processors, and the 4-bit condition field will be used to encode the extra registers. GCC has been ported to generate ARM code with the new instruction format and experimental results have shown that performance can be improved by 6% on average for MediaBench II benchmarks when the number of ISA registers is extended from 16 to 32.**

***Keywords*-Conditional Execution, ISA, Instruction Encoding, Registers**

I. Introduction

Conditional execution (or predicated execution) is an instruction set architecture (ISA) feature, which predicates individual instructions on processor status flags [5], [9], [13], [15]. The main purpose of predication is to increase the effectiveness of pipelined execution by avoiding branch instructions over very small sections of if- and while-statements. Various forms of predicated instructions have been offered on many processors. Some processors have provided partial predication support by adding limited set of predicated instructions to existing ISA, such as the built-in skip operation for each computational instruction on PA-RISC [16] and the conditional move instructions on Alpha [11] and SPARC [17]. Several processors have even featured full predication support by defining full set of predicated instructions, such as ARM [14], IA-64 [4], and StarCore [6].

Conditional execution is an important feature of the ARM series of processors, as every instruction can be made to execute conditionally. An instruction has only its normal effect if the status satisfies a condition specified in the instruction, and acts as a NOP if otherwise. This feature was chosen because it could maintain high performance while reducing hardware complexity since it could avoid introducing pipeline bubbles and compensate for the lack of a branch predictor.

Conditional execution has one serious drawback that it takes up precious encoding space. As ARM has deployed a straightforward orthogonal instruction coding that all instructions can be coded conditionally on all conditions, every instruction must reserve a bitfield for the predicate specifying whether that instruction should have an effect. Specifically, conditions are encoded into a 4-bit condition code selector on every 32-bit ARM instruction, which translates to one-eighth of the instruction length. Besides, the instruction space for conditions is underutilized since only small percentages of instructions are actually conditionalized in modern embedded applications like interactive multimedia communications, which generally demand massive computations [2]. Experiments of MediaBench II benchmarks [7] have revealed that 0.3%∼15.8% of instructions executed on ARM are conditionalized, and the overall average ratio of conditionalized instructions is about 7%. This result indicates that around 11% of instruction bits fetched to ARM are wasted for the MediaBench II benchmarks.

Although the goal of condition execution is to increase the effectiveness of pipelined execution, its performance gain is usually not as significant as expected. For certain programs, conditional execution might even cause performance degradation. Experimental results of MediaBench II benchmarks on ARM have shown that the performance impact of conditional execution over branch execution ranges from 2.7% slowdown to 8.6% speedup, and average performance improvement is 2.4%. In addition, as architecture trends push more embedded processors to deploy out-of-order execution,

978-0-7695-4322-2/10 $26.00 © 2010 IEEE
DOI 10.1109/EUC.2010.18

branch prediction, and speculation, conditional execution might not be a beneficial ISA feature for advanced embedded processors.

This paper proposes to trade conditional execution for more ISA registers on ARM processors by using the 4-bit condition field as extended register fields to encode the extra registers. As there are at most four register fields in an ARM instruction, each register field can take one bit from the condition field and hence the number of ISA registers will be doubled from 16 to 32. GCC [8] has been retargeted to generate ARM binary code with the new instruction formatted to generate ARM binary code with the new instruction format so that register spills can be significantly reduced by storing more values in the extended register file. Experimental results have shown that performance can be improved by 6% on average for MediaBench II benchmarks when the number of ISA registers is extended from 16 to 32.

The rest of this paper is organized as follows. Section II briefly outlines the condition execution feature on ARM. Section III presents the new instruction encoding, and Section IV describes the retargeting of GCC. Experimental results will be presented in Section V, and Section VI concludes this paper.

II. Conditional Execution

All ARM instructions can be made to execute conditionally only when previous instructions have set a particular condition code [14]. Specifically, an instruction only has its normal effect if a condition specified in the instruction matches the N (Negative), Z (Zero), C (Carry) and V (Overflow) flags in the *Current Program Status Register* (CPSR), as shown in Figure 1. If the flags do not satisfy this condition, the instruction acts as a NOP. This conditional execution feature allows small sections of if- and while-statements to be encoded on ARM without the use of branch instructions, increasing the effectiveness of pipelined execution by avoiding pipeline stalls caused by branch operations.

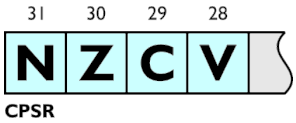


Figure 1. Status Flags

Most ARM assembly instructions can make use of conditional execution by adding two letters to the end of the instruction. The available condition codes and their corresponding status bits are listed in Table I. This feature allows small sections of if- and while-statements to be encoded on ARM without using branch instructions in order to improve code density and execution speed.

The standard example of conditional execution is the subtraction-based Euclid's Greatest Common Divisor (GCD) algorithm in the C programming language [1]:

Table I
ARM Conditional Codes

Code	Suffix	Description	Flags
0000	EQ	Equal / equals zero	Z
0001	NE	Not equal	!Z
0010	CS / HS	Carry set / unsigned higher or same	C
0011	CC / LO	Carry clear / unsigned lower	!C
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	!N
0110	VS	Overflow	V
0111	VC	No overflow	!V
1000	HI	Unsigned higher	C && !Z
1001	LS	Unsigned lower or same	!C || Z
1010	GE	Signed greater than or equal	N == V
1011	LT	Signed less than	N != V
1100	GT	Signed greater than	!Z && (N == V)
1101	LE	Signed less than or equal	Z || (N != V)
1110	AL	Always (default)	any

```
while (i != j)
    {
       if (i > j)
           i -= j;
       else
           j -= i;
    }
```

The GCD loop can be implemented with conditional execution of branches only:

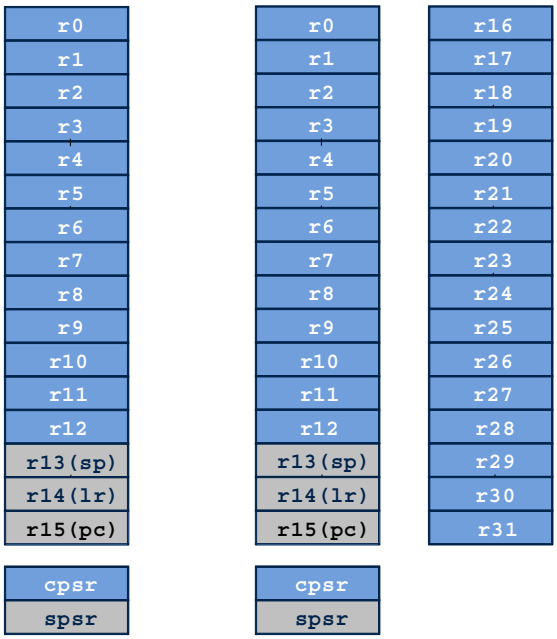
```
gcd   CMP  r0, r1
      BEQ  end
      BLT  less
      SUB  r0, r0, r1
      B    gcd
less  SUB  r1, r1, r0
      B    gcd
end
```

which takes seven instructions, including two conditional branch instructions and two unconditional jump instructions. By using the conditional execution feature of the ARM instruction set, it can be implemented in only four instructions:

```
gcd   CMP   r0, r1
      SUBGT r0, r0, r1
      SUBLT r1, r1, r0
      BNE   gcd
```

The implementation using conditional execution generates shorter code and increases execution speed.

Figure 2. Condition Encoding



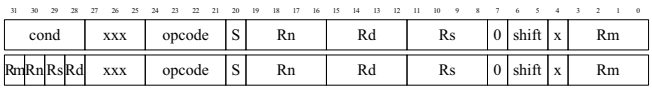
(a) Original (b) Extended Register File

Figure 3. Registers

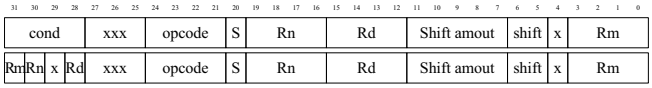
III. Instruction Encoding

The conditional execution feature is implemented with a 4-bit condition code field (i.e. bits[31:28]) on every instruction, as shown in Figure 2. This cuts down significantly on the encoding bits available for displacements in memory access instructions or registers in data processing instructions. This paper presents an instruction encoding scheme that uses this 4-bit condition field to represent registers higher than r15. The original register file containing 16 general-purpose register shown in Figure 3(a) will be expanded to 32 registers by the new instruction encoding, as depicted in Figure 3(b). The first 16 registers of the extended register file remain exactly the same as the original registers.

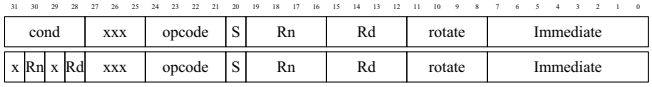
As there are at most four register fields in an ARM instruction, such as the data processing register shift instruction `ADD r5, r5, r3, LSL r2`, each register field can take one bit from the condition field and that is why the number of ISA registers can only be doubled from 16 to 32. Figure 4(a) shows the instruction formats with four register fields. The top format is the original instruction encoding format that represents an ARM instruction with the form of `op Rd, Rn, Rm, shift Rs`. The bottom format in Figure 4(a) depicts that each bit of the 4-bit condition field is



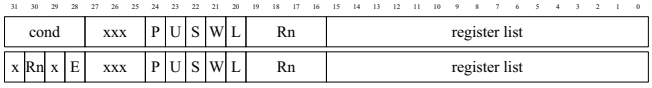
(a) 4 Register Fields



(b) 3 Register Fields



(c) 2 Register Fields



(d) 1 Register Field

Figure 4. Instruction Formats

assigned to one of the four register fields in the instruction. As a result, each register field in the new instruction format has 5 bits and hence can address all the 32 registers in the extended register file.

Figure 4(b) and Figure 4(c) illustrate that the instruction formats with three and two register fields can be easily modified to accommodate the extra bits for the extended registers. However, special attention is needed when changing the last instruction format with only one register field displayed in Figure 4(d). The reason is that this instruction format is used to encode the load and store multiple instructions, which can load and store a subset, or possibly all, of the general-purpose registers from and to memory. Unfortunately, it is impossible to represent all the 32 registers in the extended register file in the 32-bit instruction format. Therefore, the extended register file will be divided into two halves: the lower half (r0-r15) and the upper half (r16-r31), and only half of the registers can be accessed by a load/store multiple instruction. Specifically, when the "E" bit is cleared, a load/store multiple instruction can only reference the lower half of registers. When the "E" bit is set, a load/store multiple instruction can only access the upper half of registers.

IV. GCC Retargeting

GCC 4.4.1 has been retargeted to generate ARM binary code with the new instruction encoding, which supports 32 ISA registers. The retargeting process of GCC backend for ARM consists of the following three steps.

Disabling Conditional Execution: In order to disable the conditional execution feature, all the related transformations, such as if-conversion, in the machine description (MD) file

arm.md must be turned off. In addition, some instruction patterns for machine-dependent optimizations in the MD file must be switched off as well. As a result, GCC backend will search the MD file and look for appropriate patterns that will generate ARM object code with branch instructions.

Instruction patterns can be disabled by placing an off flag in their RTL statements. Consider the following pattern for `Max(a,b)`, which is part of machine-dependent optimizations in *arm.md*:

```
(define_insn "*arm_smax_insn"
  [(set
     (match_operand:SI 0
       "s_register_operand" "=r,r")
     (smax:SI (match_operand:SI 1
       "s_register_operand" "%0,?r")
     (match_operand:SI 2
       "arm_rhs_operand" "rI,rI")))
     (clobber (reg:CC CC_REGNUM))]
  "TARGET_ARM"
  "@
   cmp\\t%1, %2\;movlt\\t%0, %2
   cmp\\t%1, %2\;movge\\t%0, %1\;
   movlt\\t%0, %2"
  [(set_attr "conds" "clob")
   (set_attr "length" "8,12")]
)
```

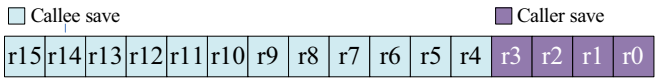
This pattern can be turned off by placing the off flag `TARGET_NO_COND_EXEC` after `TARGET_ARM` and changing the statement to

```
"TARGET_ARM && !TARGET_NO_COND_EXEC"
```

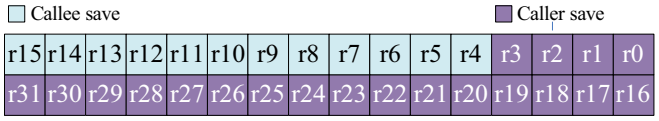
where `TARGET_NO_COND_EXEC` is a new flag introduced by this paper and defined in the *arm.h* file under the *gcc/config/arm/* directory.

Doubling Registers: GCC must be informed that the number of registers has been expanded to 32, which requires several parameters in the files *arm.h* and *aout.h* to be updated. In addition, the sets of caller save registers and callee save registers have to be modified as well. Figure 5(a) shows that r4-r15 are callee save registers while r0-r3 are caller save registers in the original GCC setting. In order to keep things simple and avoid pushing too many register values onto the stack by procedure invocation, the set of callee save registers remains the same and all the extended registers r16-r31 are assigned as caller save registers.

Recompiling C Libraries: The C library used by GCC must be recompiled for the new instruction encoding. It is a straightforward process as the GCC has been retargeted, except for the assembly code embedded in the source of the library. In order to translate these embedded assembly code to the new target, GNU assembler has been modified to identify the assembly instructions that are conditionalized and then convert them to branch instructions.



(a) 16 Registers



(b) 32 Registers

Figure 5. Caller and Callee Save Registers

Table II
BENCHMARK DESCRIPTION

Name	Description
cjpeg	JPEG encoder
djpeg	JPEG decoder
jpeg2000dec	JPEG-2000 decoder
jpeg2000enc	JPEG-2000 encoder
h263dec	H.263 decoder
h263enc	H.263 encoder
h264dec	H.264/MPEG-4 AVC decoder
h264enc	H.264/MPEG-4 AVC encoder
mpeg4dec	MPEG-4 ASP decoder
mpeg4enc	MPEG-4 ASP encoder
mpeg2dec	MPEG-2 decoder
mpeg2enc	MPEG-2 encoder

V. EXPERIMENTAL RESULTS

A. Setup

Experiments are conducted by executing the MediaBench II benchmarks [7] on SimpleScalar/ARM, a port of SimpleScalar [3] to ARM available from the University of Michigan. It simulates the five stage pipeline of Intel StrongARM processors, such as SA-1110 processor [10], which implements the ARM V4 architecture. Instructions are issued in-order by invoking the `sim-outorder` command with the option `-issue:inorder`. The configurations of L1 D-Cache and I-Cache for this processor are: 4KB cache size, 128B line size, 4-way associativity, and miss penalty of 1 cycle.

In addition to the original ARM configuration, SimpleScalar/ARM has been modified to support the following two configurations: ARM with 16 registers and no conditional execution, and ARM with 32 registers.

Table II lists the video and image benchmarks of the MediaBench II suite, which is the successor of MediaBench benchmarks [12]. They are the most advanced and most widely-used multimedia applications.

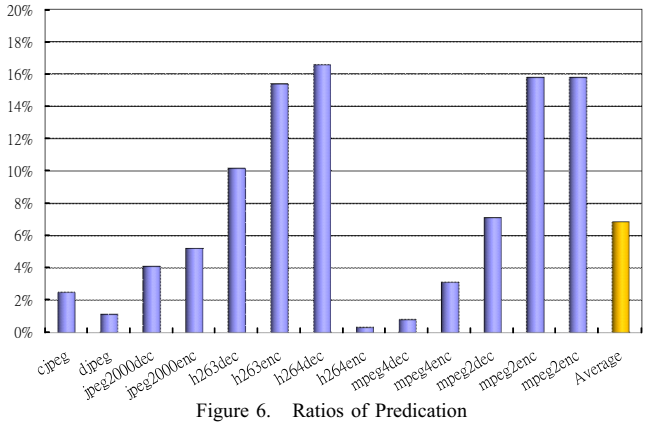


Figure 6. Ratios of Predication

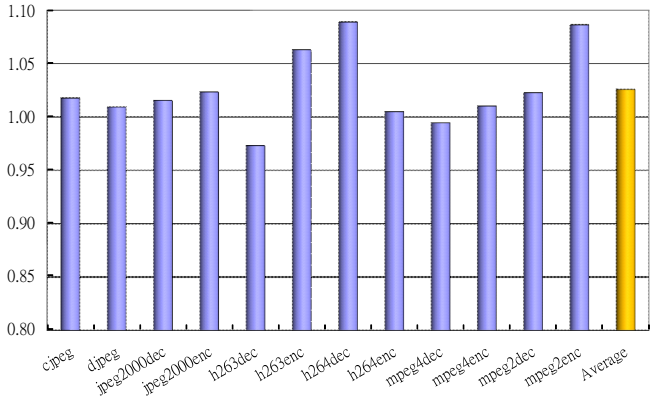


Figure 7. Speedup of Conditional Execution

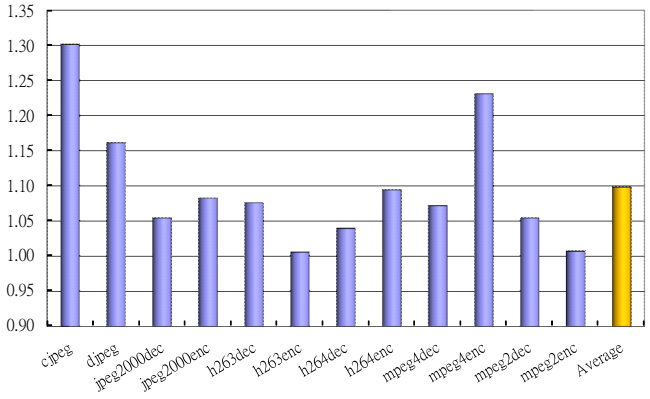


Figure 8. Speedup of 32 Registers

Table III
SPILLS IN OBJECT FILES

Program	16 Registers	32 Registers	Spill Reduction
cjpeg	3285	502	84.7%
djpeg	3282	502	84.7%
jpeg2000	4902	520	89.4%
h263dec	9624	1492	84.5%
h263enc	1406	1060	24.6%
h264dec	104876	12757	87.8%
h264enc	49136	19473	60.4%
mpeg4	964	748	22.4%
mpeg2dec	571	18	96.8%
mpeg2enc	1753	144	91.8%

B. Ratios of conditionalized Instructions

Since a condition field occupies a 4-bit space on every 32-bit ARM instruction, it is important to know if the condition fields of ARM instructions are underutilized. The more instructions that are conditionalized means the condition space is better utilized. Therefore, the ratio of conditionalized instructions executed by every MediaBench II benchmark has been measured by counting the number of instructions that are actually predicated, and the result has been depicted in Figure 6. It has revealed that only 0.3%∼15.8% of instructions executed on ARM are conditionalized, and the overall average ratio of conditionalized instructions is about 6.9%. This result indicates that the condition fields of most instruction are wasted for the MediaBench II benchmarks.

C. Conditional Execution vs. No Conditional Execution

As condition execution takes up one-eighth of the instruction encoding space on ARM for better pipeline efficiency, it is desirable that the performance improvement would be significant. Two versions of ARM binary programs have been generated by GCC from the MediaBench II benchmarks: one with conditional execution and one without, and their execution times have been measured to determine the speedup of conditional execution version over the non-conditional execution version. Figure 7 shows that 10 of 12 benchmarks achieve speedup ranging from 0.5% to 8.6%, while two benchmarks suffer slowdown of 0.5% and 2.7%. The average performance improvement is merely 2.4%.

D. 32 Registers vs. 16 Registers

In order to determine the effect of doubling the number of registers from 16 to 32, no conditional execution will be used in this subsection. Different sets of object programs of the MediaBench II benchmarks have been generated by GCC, and have been executed on 16-register ARM and 32-register ARM respectively. Performance improvement of doubling registers range from less than 1% to over 30%, and the average speedup is 9.8%.

The performance improvement comes from the fact that register spills have been significantly reduced as more values have been stored in the extended register file. Table III tabulates the numbers of register spills that have been introduced by GCC into the target programs of the MediaBench II benchmarks. Note that the JPEG 2000 encoder and

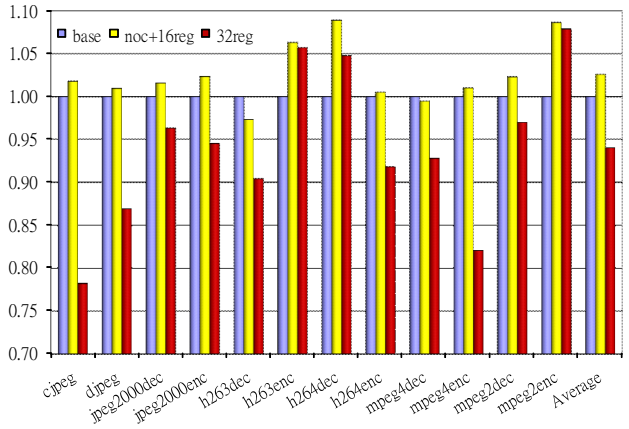


Figure 9. Normalized Execution Times

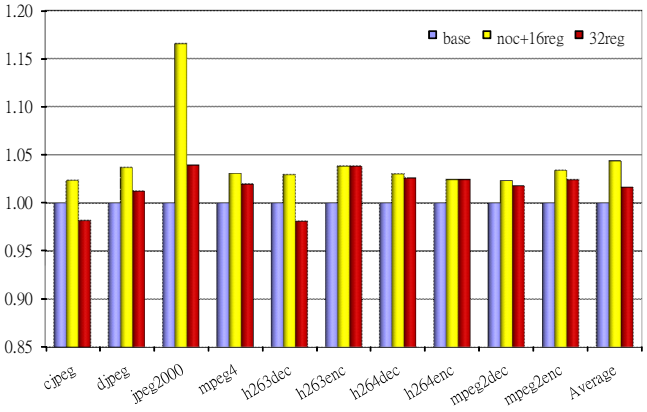


Figure 10. Normalized Code Sizes

decoder are stored in the same object program *jpeg2000*, and the MPEG 4 encoder and decoder are combined in the same file *mpeg4*. As the number of registers is extended from 16 to 32, the numbers of register spills are reduced drastically. More than half of benchmark programs have observed reduction rates of over 80%. The average ratio of register spill reduction is 72.7%.

E. Normalized Execution Times

Figure 9 illustrates the normalized execution times of the MediaBench II benchmarks for different ARM configurations. The "base" configuration is the original ARM, which has 16 registers and the conditional execution feature. The "noc+16reg" configuration indicates that the condition execution feature has been turned off on the 16-register ARM, while "32reg" configuration denotes the 32-register ARM. The measured cycle counts of the MediaBench II benchmarks on these configurations have been normalized according to the base configuration.

The figure shows that performance would be slightly worse than the original ARM when conditional execution feature is turned off for the most of MediaBench II benchmarks, with the average slowdown of 2.6%. However, speedup can be observed for the most of MediaBench II benchmarks when the number of registers is doubled. The only three exceptions are *h263enc*, *h264dec*, and *mpeg2enc*, whose slowdown ratios are around 5%. The best speedup is achieved by *cjpeg*, which is 21.8%. The average performance improvement is 6.0% for MediaBench II benchmarks when the number of ISA registers is extended from 16 to 32.

F. Normalized Code Sizes

As the length of every instruction remains the same, the code sizes of generated object files will not change much. The difference is caused by replacing conditional instructions by branch instructions when the conditional execution feature is turned off, and by reducing register spill instructions when the register file is doubled. Figure 10 depicts the normalized code sizes of the MediaBench II benchmarks for different ARM configurations. The average codesize expansion is 4.4% when the conditional execution feature is turned off, and 1.6% when the register file is doubled.

VI. Conclusions

This paper proposed to free up the underutilized condition field in the ARM instruction encoding by disabling conditional execution feature, and then use the space to accommodate the extra bits that were need for register fields when the number of registers was doubled. Experimental results showed that the performance could be improved by 6% on average for MediaBench II benchmarks when the conditional execution feature was traded for 16 extra ISA registers on ARM processors.

Acknowledgment

This research was supported in part by MOEA project 98-EC-17-A-01-S1-034 and NSC grant NSC98-2221-E-011-065-MY3.

References

[1] ARM Limited. *ARM Developer Suite Assembler Guide*, 2001.

[2] Todd Austin, David Blaauw, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Wayne Wolf. Mobile supercomputers. *IEEE Computer*, 37(5):81–83, May 2004.

[3] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, 2002.

[4] Carole Dulong. The IA-64 architecture at work. *IEEE Computer*, 1998(7):24–32, July 31.

[5] Edil S. T. Fernandes, Anna Dolejsi Santos, and Claudio L. de Amorim. Conditional execution: An approach for eliminating the basic block barriers. *Microprocessing and Microprogrammin*, 40:689–692, 1994.

[6] Freescale Semiconductor Inc. *SC140 DSP Core Reference Manual*, 2005.

[7] Jason Fritts and Bill Mangione-Smith. MediaBench II - technology, status, and cooperation. In *Proceedings of the Workshop on Media and Stream Processors*, 2002.

[8] GCC. The GNU compiler collection. http://gcc.gnu.org/.

[9] Wen-Mei Hwu. Technology outlook: Introduction to predicated execution. *IEEE Computer*, 31(1):49–50, January 1998.

[10] Intel Corporation. *Intel StrongARM SA-1110 Microprocessor Developers Manual*, June 2000.

[11] R.E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[12] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'97)*, pages 330–335, 1997.

[13] Joseph C. H. Park and Mike Schlansker. On predicated execution. Technical Report HPL-91-58, HP Labs, 1991.

[14] David Seal, editor. *ARM Architecture Reference Manual*. Addison-Wesley Professional, 2nd edition, 2001.

[15] Bernd Ullman. Instruction looping, an extension to conditional execution. *ACM SIGARCH Computer Architecture News*, 26(1):3–4, March 1998.

[16] Jonathan P. Vogel and Bruce K. Holmer. Analysis of the conditional skip instructions of the hp precision architecture. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 207–216, 1994.

[17] David L. Weaver and Tom Germond, editors. *The SPARC Architecture Manual, Version 9*. SPARC International, Inc., Santa Clara, California, 2000.