Quantitative approach to ISA design and compilation for code size reduction

K. M. Lo, Lin Ma

SimpLight Nanoelectronics, Ltd., Beijing China
100088
kevin.lo@simplnano.com, lin.ma@simplnano.com

Abstract

In this paper, an efficient code size optimization instruction set architecture targeting embedded telecommunication applications is introduced. Nowadays, mixed 16-bit and 32-bit size instruction set approaches are commonly used to achieve code size reduction while minimizing performance loss. They are usually designed with some restrictions such as reducing the number of accessible registers, mode switching, or special hardware logic handling.
The approach starts with a common, basic RISC ISA [6] and a re-targetable high performance compiler. The Open64 compiler was chosen for its machine independent optimization so that once retargeted, the generated code will be of high performance quality. Once retargeted, we start our ISA compression design based on statistics collected from the code generated. By judicious selection from actual instructions generated, a high code compression rate is achieved without adding restrictions to the number of registers used and hardware implementation. Furthermore, this approach does not introduce any noticeable performance degradation due to the mixed 32/16-bit ISA compared to the full 32-bit ISA.

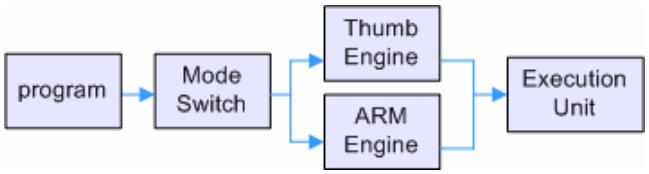
Keywords *mixed instruction code generation, code size, ISA design, instruction scheduling*

1. Introduction

Due to technological advances, people now enjoy high speed computing and large amounts of memory at relatively low prices in desktop computers. However, in the world of embedded systems, the situation is quite different. On the embedded systems, size and power consumption is a big concern during the development process. Therefore, code size is always a critical issue on embedded systems while on desktop platforms it is insignificant.
Because of the importance of code size, many system developers place much effort on improving the related issue. Common methods to reduce code size are mainly based on mode switching, pre-processing decoder, 1-to-n instruction mapping, 16/32-bit instructions mixing, or reducing the number of accessible registers. Some methodologies from well-known developers are discussed below:

1.1 ARM - Thumb [7, 12]

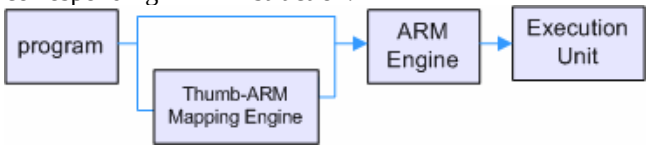
The code compression is done by using Thumb ISA with about extra 36 16-bit instructions. A mode-switch instruction is needed to differentiate between the modes, and the program can only be executed either in 16-bit mode or 32-bit mode.



Thumb ISA cannot handle interrupts and only eight registers out of sixteen can be accessed. Code size can be reduced up to 20-30% while the performance is reduced about 15% due to mode-switch and other overheads [8].

1.2 ARM - Thumb-2[11]

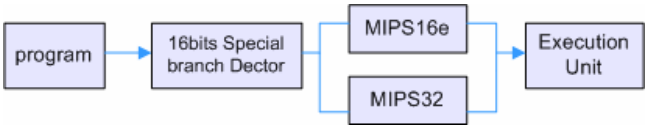
Thumb-2 is an individual ISA with mixed 16-bit and 32-bit instructions. Unlike Thumb, Thumb-2 does not require any mode-switch and is a complete, functional ISA. A special unit is added to map the Thumb-2 instruction to the corresponding ARM instruction.



Possible code size reduction with Thumb-2 is less than the reduction possible with Thumb, with a performance penalty of about 15-25% [8].

1.3 MIPS - MIPS16e [9]

MIPS16e is a 16-bit ISA extension in some MIPS processors. And MIPS16e is used similarly to Thumb. A mode-switch with a special branch instruction is needed to switch between 16-bit and 32-bit modes.



Code size can be reduced up to 20-30% while the performance is reduced about 15% due to the extraction of data structure and mode-switch handlings.

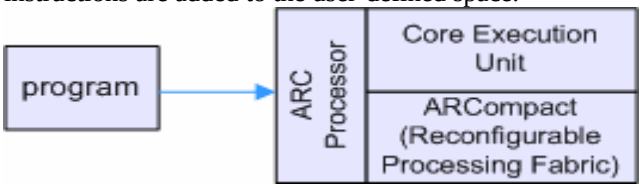
1.4 IBM PowerPC – CodePack [5, 10]

CodePack uses an approach similar to application compression/decompression. The executable is compressed by a program based on several compression algorithms. The executable will be decompressed on the fly during execution by the CodePack-equipped processor.

CodePack delivers up to 20-30% compression rate with a negligible performance penalty [8]. However, a complex hardware de-compressor support is required.

1.5 ARC International – ARCompact [1, 2]

ARCompact allows users to define their instructions. In order to improve the code size optimization, some 16-bit instructions are added to the user-defined space.



By using intermix of 16/32-bit instructions, up to 40% compression rate with a negligible performance penalty [8]. However, complex hardware design is required.

From the methodologies listed above, developers place much effort on hardware design to improve the code size. However, hardware support is always accompanied by higher costs and less flexibility compared to software. By making use of statistical data from fully optimized binaries and software-hardware co-design methodology, an ISA that supports mixed 16/32-bit instructions was designed, with very limited hardware costs. The code size compression ratio is competitive, and the performance penalty is negligible. Mode changing is not needed. The ISA supports thirty-two registers for both the 32-bit and 16-bit sized instructions.

2. Instruction Analysis

Most RISC ISA support a maximum of three operands. However, it is not essential that every instruction occupy all operand space. Provided that thirty-two registers (ie. GPR field = 5bits) and 6-bit opcode length are supported, several groups have been defined as follows to generalize the instruction format:

Combination	Bits used	Example
No operands	0 bits	Jr, Ret
Index24	24bits	Jp
2 GPRs/	10 bits	Mvtc, Mvfc
3 GPRs	15 bits	Add, muls
2GPRs+ imm16	10bits+16bits	Load/Store
2GPRs+ imm5	10bits + 5bits	Shift imm

In order to compress a 32-bit instruction into a 16-bit instruction, the following methods are commonly used:

- Reducing the number of available registers (the number of bits representing GPR fields). As the number of useable registers is reduced, the number of register swapping will be increased. Performance penalty is introduced and the register usability is reduced.
- Reducing the value range of immediate operand. The expressible offset/value range is reduced.
- Breaking a 3-operand instruction to a 2-operand (or less) instruction. Extra instructions are needed for the same function, and therefore, the number of execution cycles is usually increased [6].

By compressing 32-bit instructions to 16-instructions, these solutions can reduce the code size. However, the overhead that accompanies the solutions is also significant and impacts the overall performance. In order to select the right instruction candidates for optimizing code size, a series of analyses have been done. These analyses are based on some data from several commonly used applications of embedded systems.

- Uclibc Open source library package for embedded applications
- G729a Voice codec program used in mobile phones
- Mpeg4 MPEG-4/ASP decoder program
- Nucleus A popular RTOS for embedded processors (ported to the Mips-like architecture)
- libmad MPEG audio decoder library
- ucLinux Linux kernel release 2.6.xx for embedded processors

- Lay 2/3 Layer 2 and Layer 3 of the GSM wireless communication protocol stack

2.1 Register Reducing Method

At first, a typical method is used to test the influence to performance. The compiler is modified to reduce the number of available registers from thirty-two to sixteen. Applications are compiled by both original compiler (32 available registers) and modified compiler (16 available registers). The numbers of instruction issuing and execution cycles are counted for both versions of applications.

The performance penalty is significantly larger and violates the original criteria, namely, achieving a high compression ratio without significant performance impact. From this experiment, the number of instructions count was increased by 1.6%-13.1% (Figure 1) and the number of execution cycles was increased by 2%-12.4% (Figure 2). The performance penalty will be higher in proportion to the complexity of the applications. Because of the significant performance penalty, this mechanism is not the ideal method to design compressed instructions.

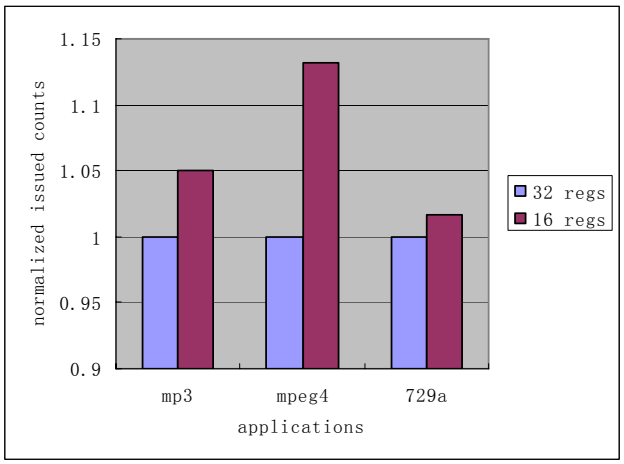


Figure 1: Normalized execution instruction counts (32 registers vs. 16 registers)

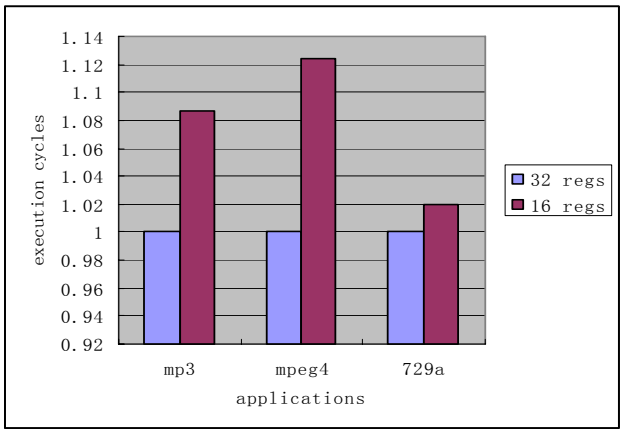


Figure 2: Normalized execution cycles (32 registers vs. 16 registers)

2.2 Instruction characteristics

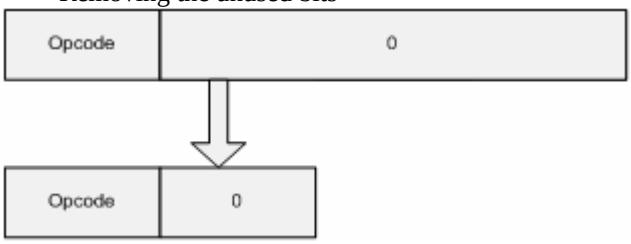
To find a better solution to reduce the code size without a large impact on performance, a series of detailed experiments were completed. From the results, some interesting characteristics have been found. Around 50-65% of the instruction distribution belongs to one of the following groups (Table 1):

- Instruction using $0 (hardwired register value 0) accounts for a large percentage of static instruction counts. The reason being that many register copy operations are involved (add $rd, $rs, $0). Another typical case is branch instructions with branch condition involved $0 (eg. x>0).
- $sp(stack pointer) with small offset, which is mainly due to memory spill/fill operations. Offset of this type is always word aligned and positive, and we found approximately 80.44% could be represented by 7-bit immediate operand (Fig. 3).
- Destination register is same as one of the sources, that is, 2-operand logically. The reason is that much of the calculation is going to apply the result to itself (eg. i++, a=a<<4).
- Offset is zero. As mentioned, most load/store operations have zero offset.
- Instruction without any operand.
- Calculation with a small immediate or offset. This is due to most of ALU calculations being small number or pointer increments (eg. a=b+2, ptr++). Another reason is that quite a few of branches are within a small offset. Around 78% of the offsets could be represented by 5-bit unsigned immediate operand (Figure 4).

2.3 Compressible Instructions

From the above observations, related instructions can either be reduced to 16-bit or converted to a 2-operand instruction by following ways:

- Removing the unused bits



application	$0	$sp	$rs==$rd	imme0	No operand	total
uclibc	26.83%	23.99%	4.88%	5.93%	3.00%	64.64%
mp3	18.46%	21.07%	8.51%	3.71%	1.79%	53.53%
mpeg4	12.92%	11.55%	19.10%	4.61%	0.85%	49.02%
729a	27.24%	22.55%	8.38%	4.60%	1.53%	64.29%
nucleus-demo	18.44%	21.00%	5.70%	4.52%	3.72%	53.38%
L2/L3	22.89%	21.21%	3.62%	3.55%	3.55%	54.82%
linux	19.60%	27.44%	6.85%	3.56%	2.42%	59.86%

Table 1: Distribution for some compressible instructions.

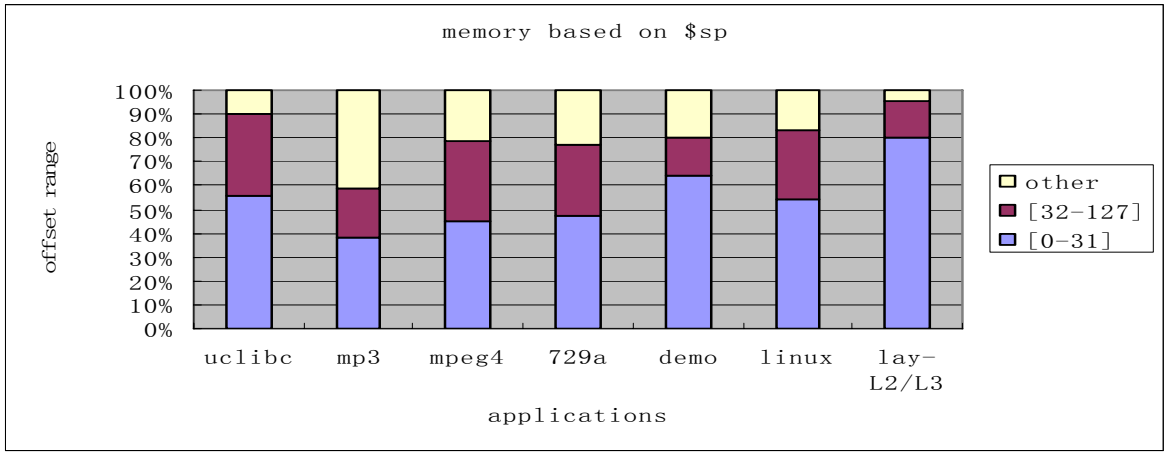


Figure 3: Range distribution of memory operations based on stack pointer.

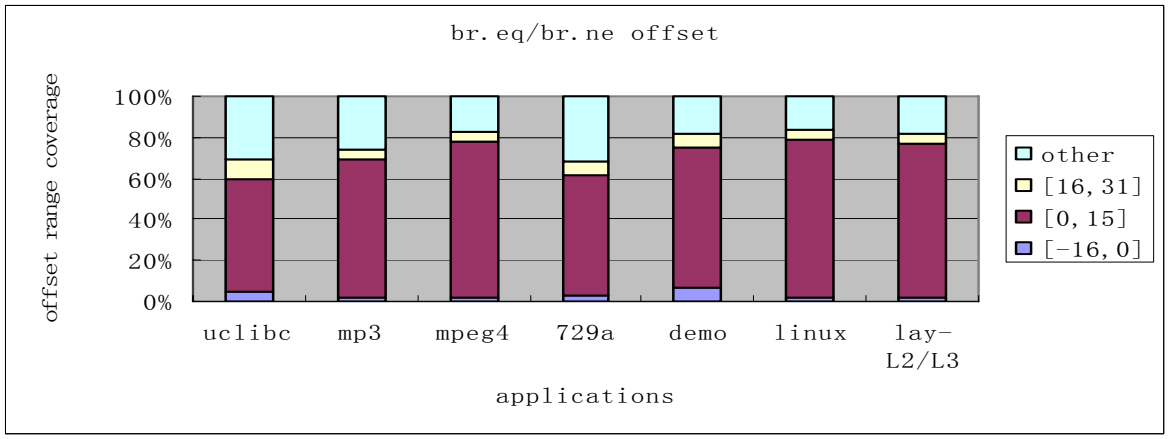
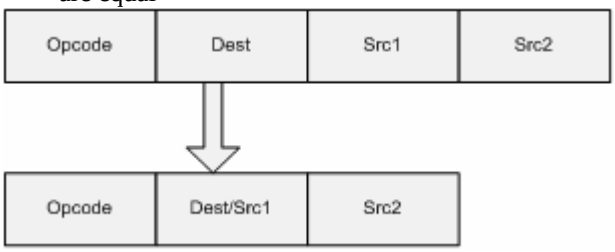


Figure 4: Range distribution of br.eq/br.ne offset

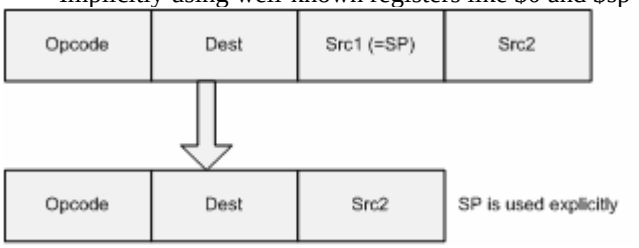
application	uclibc	mp3	mpeg4	729a	Nucleus-demo	linux	L2/L3
coverage	51.62%	42.46%	45.07%	48.89%	45.64%	49.59%	49.04%

Table 2*:* Coverage of designed compressed instructions.

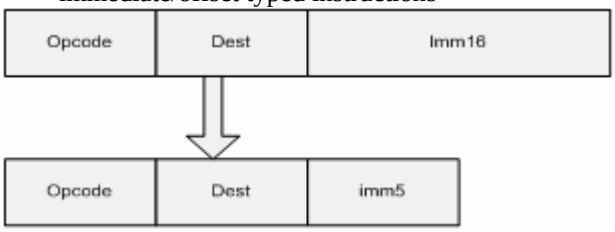
- Combining destination operand to source when they are equal



- Implicitly using well-known registers like $0 and $sp



- Reducing the immediate size to 5bits for some immediate/offset typed instructions



By selecting the compressible and frequently used instructions, a special 16-bit instruction set is designed. The 16-bit instruction set can be covered up to approximately 40% to 50% of the testing program instruction distribution (Table 2).

3. Compiler Support

After designing the 16-bit instruction subset, an enhanced compiler is essential for effectively generating those16-bit instructions. Since the average coverage is around 47.47%, the maximum code reduction goal should be 23.7%. In this section, the modifications of the retargeted Open64 compiler and the code generating process will be described. According 2.1 described above, the compressible characteristics can be detected after the register allocation phase in the compiler. There are three steps to generate optimized mixed instructions in the compiler as shown in Figure 5.

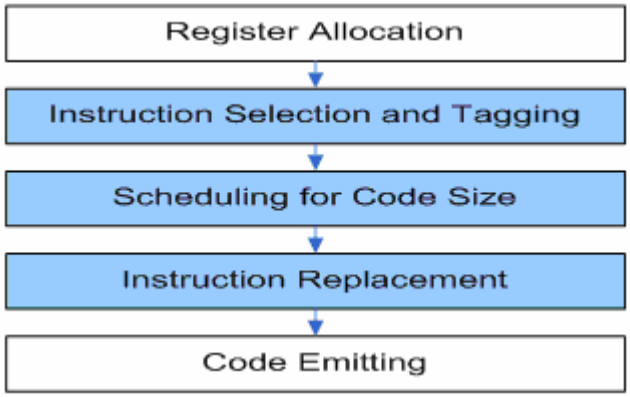


Figure 5: Flow of mixed instruction set generation.

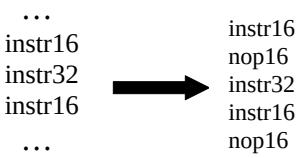
3.1 First step: marking convertible instructions

According to the rules described in Section 2, the 32-bit instruction set is divided into different types (see Table 3). If an instruction has a corresponding compressed opcode and satisfies any of following conditions, it is marked as a candidate for 16-bit instruction.

Instruction type	Satisfied condition
Opcode rd, rs1, rs2	rd==rs1 or rd==rs2 or rs1==$0 or rs2 ==0
Opcode rd, rs1, imm	((rd==rs1 or rs1 == $sp) and restricted immediate value) or imm == 0
Opcode rd, rs1	no condition
Opcode	no condition

Table3: Instruction selection pattern.

In this step, instruction candidates will be selected and tagged but instruction replacement will not be done. In our architecture, 16-bit instructions must be half-word aligned, and 32-bit instructions must be word aligned. Therefore, 16-bit instructions should come in pairs. Also, a 16-bit NOP is required to fill up the slot of the unaligned 16-bit instructions should the next instruction be a 32-bit one. As such, the compression rate will be reduced if the code replacement is completed in this step. For example, if the assembler code sequence is as below, 16-bit NOP instruction will be inserted between instr16 and instr32 for fetching alignment, thus the code size cannot be reduced. Moreover, a nop16 instruction occupies an issue slot and instruction buffer, which is extra overhead.



After this step, the average code size reduction of 14%-21% can be achieved (Table 4).

3.2 Second step: optimization for code size reduction

In this step, 16-bit instruction will be grouped together by the compiler scheduler. This operation differs from three-operand instruction with limited registers which need a complex cost model in the register allocation phase to achieve a balance between the register spill cost and code size[3,4,13]. Originally, instruction scheduling in Open64 is designed to minimize the number of performance stalls. By making use of the original local scheduler, a heuristic process to group suitable 16-bit instructions together is added. For each basic block, a list of candidate instructions is computed. The instructions without data hazard at each time step *t* are selected and sorted by latency in decreasing order. A compressed instruction is selected at time step *t* as best candidate if and only if one of following conditions is satisfied:

- At time step *t*, the compressed instruction is the only candidate.
- At time step *t*, the number of compressed instructions of the candidate list is larger than 1.
- The committed scheduled instruction at time step *t-1* is compressed.

By this method, compressed instructions can be scheduled together and paired up. Based on this heuristic instruction scheduling, the code size reduction is achieved by 17%-24% (Table 4).

Meanwhile, in order to minimize performance degradation, not all basic blocks are treated in the same manner. For example, if the basic block is a loop body, performance has higher preference than code size. In order to have a balance between performance and code size, the code scheduling will favor performance over code size. Thus the code size compression rate is achieved by 16%-23% (Table 4).

3.3 Third step: instruction replacement

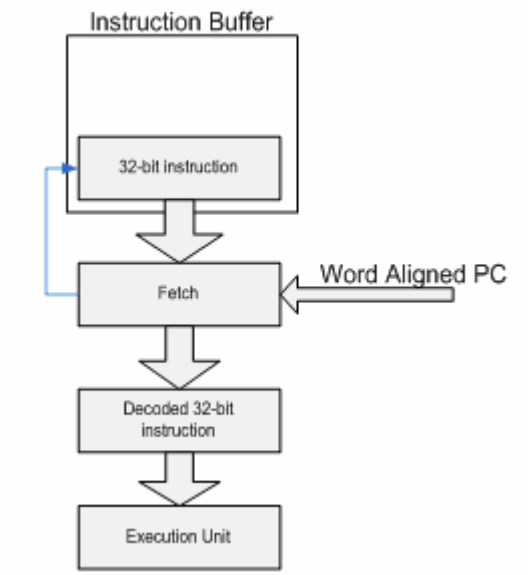
In this step, the instruction replacement will finally be carried out. All tagged 16-bit instructions will be checked. If paired instructions are found, they will be substituted by equivalent 16-bit instructions. Otherwise, the tag will be removed, and the original 32-bit instruction will be used. As a result, there may still be several percents of 32-bit instructions that cannot be replaced for lack of paired 16-bit instructions.

4. Hardware support

The 16-bit instruction set will actually be expanded to a corresponding equivalent 32-bit instruction so no extra execution unit is required. The decoding is transparent to the program. The 16-bit instruction will be supported and handled by the corresponding 32-bit instruction execution unit and the instruction will be expanded in the normal decoding phase. In the instruction decoding phase, a special instruction fetching handler with limited hardware logic is provided.

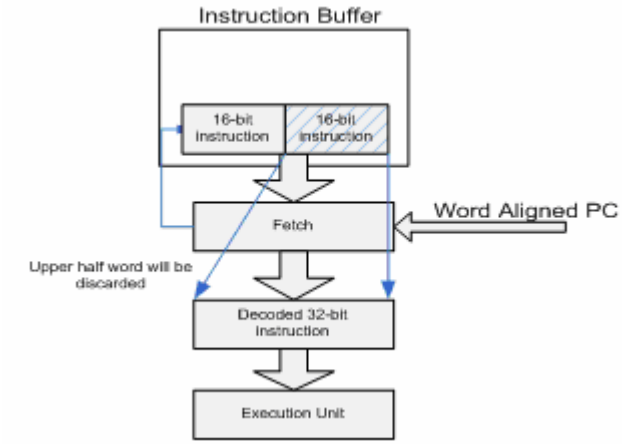
4.1 Original 32-bit instructions

The 32-bit instruction will be fetched and decoded as usual. Only word aligned PC is supported.



4.2 Word aligned 16-bit instructions

For a 16-bit instruction with word aligned PC, the 16-bit instruction opcode will be decoded so that only the first sixteen bits will be taken by the decoder to form a corresponding word-aligned 32-bit instruction.

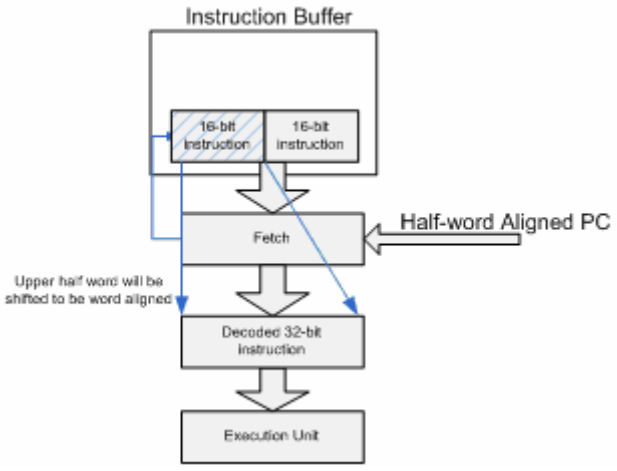


File size(byte)	instr32	Mixed-perf	Mixed-perf reduction	Mixed-size	Mixed-size reduction	Mixed-p&s	Mixed-p&s reduction
Uclibc	126400	105664	19.62%	103664	21.93%	103952	21.59%
mp3	55004	48092	14.37%	46764	17.62%	47148	16.66%
mpeg4	113220	97940	15.60%	95892	18.07%	96836	16.92%
729aori	53244	44812	18.82%	43820	21.51%	44140	20.63%
nucleus-demo	37779	32451	16.42%	31763	18.94%	31827	18.70%
L2/L3	600795	506971	18.51%	494011	21.62%	495435	21.27%
Linux	1016364	843244	20.53%	823792	23.38%	826400	22.99%

Table 4: Code size reduction comparison using 32-bit instructions and mixed instruction set.
*for uclibc, the size is text segment
* mixed-*: mixed 32/16 instructions with different scheduling policies
* mixed-perf: for performance only
* mixed-size: for code size
* mixed-p&s: for code size except loop body

4.3 Half-word aligned 16-bit instructions

For a 16-bit instruction with half-word aligned PC, the 16-bit instruction raw bits will be shifted to form a corresponding word-aligned 32-bit instruction. Then the opcode will be fetched and decoded as a world-aligned 16-bit instruction.



By using this simple fetching and decoding scheme, 16-bit and 32-bit instructions can be mixed transparently with negligible performance impact and hardware implementation cost.

5. Performance Analysis

In order to determine the impact of our ISA compaction on performance, the execution cycles of applications have been measured. A performance simulator was used to measure pure execution cycles. For the compressed instruction replacement algorithm, the total instruction numbers of different mixed 16/32-bit versions are same as the full 32-bit instructions. For the heuristic instruction scheduling on code size purpose, the mixed instructions scheme has slight but negligible degradation to performance compared to the full 32-bit instructions without memory effects such as I-cache miss and D-cache misses (Figure 6). Furthermore I-cache miss counts are decreased (approximately 26% ~ 46%) due to code size reduction (Figure 7). If considering the penalty of I-cache miss, the mixed instructions scheme has slight improvement to performance from 0.6% to 4.6% (Figure 8).

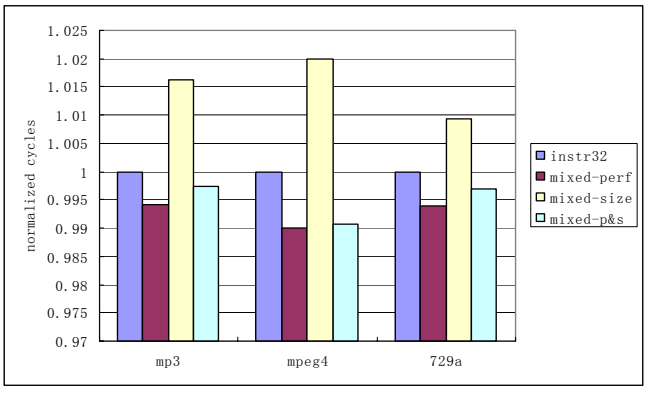


Figure 6: Normalized cycle counts without I-cache miss penalty.

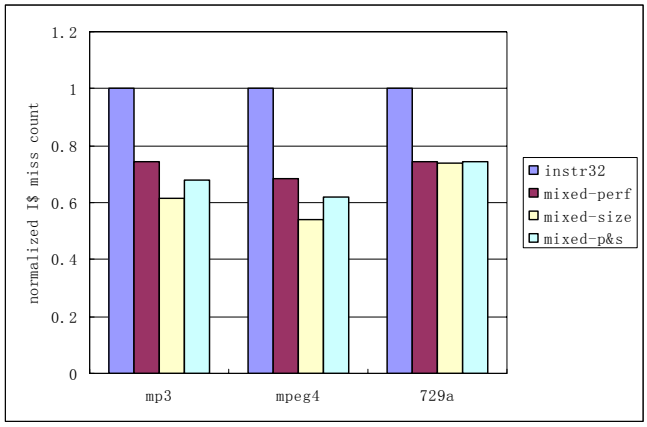


Figure 7: Normalized I-cache miss counts

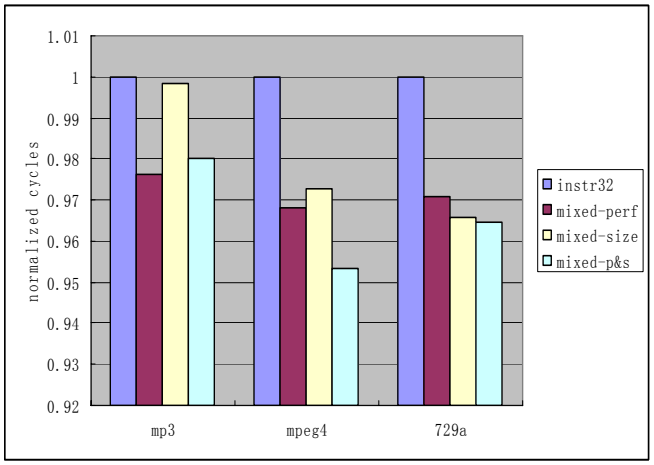


Figure 8: Normalized cycle counts with I-cache miss penalty.

6. Summary

See Table 5.

7. Conclusion

A methodology of compressed instruction selection and generation is introduced in this article. By using a native support 16/32-bit mixed ISA, an advanced compiler, and limited hardware logics, a significant 17-23% code reduction ratio is achieved without performance penalty. The instruction scheduling heuristic on code size purpose helps to improve about 2.7% of the code reduction ratio. With the scheduling policy, the average code reduction ratio is 20.44%, which achieved 86.2% of the peak code reduction ratio (23.7%) of our mixed instruction. Other than instruction alignment, there are practically no restrictions on the code generated. All thirty-two registers are usable in any of the instructions used. There is no need for mode changing during execution.

8. Reference

[1] ARC Cores, ARC Tangent Programmer's Reference", 2001.

[2] ARC Cores, ARCompact Technical Backgrounder, http://www.arc.com [ONLINE], 2001.

[3] A. Krishnaswamy and R. Gupta, "Profile guided selection of arm and thumb instructions", ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'02), Jun 2002.

[4] Halambi,A, etc., "An efficient compiler technique for code size reduction usingreduced bit-width ISAs", Design, Automation and Test in Europe Conference and Exhibition, Proceedings Volume, Issue 2002, pp.402-408, 2002.

[5] IBM, "CodePack PowerPC Code Compression Utility User's Manual Version 3.0", IBM, 1998.

[6] J.L. Hennessy, D. A. Patterson, "Computer architecture: a quantitative approach", Morgan Kaufmann Publishers, Inc., 2002.

[7] J.L. Turley, "Thumb Squeezes Arm Code Size", Microprocessor Report, 9, 4., 1995.

[8]Jim Turley, "Code compression under the microscope", http://embedded.com/columns/significantbits/17701289?_requestid=204208 [ONLINE], Feb 2004.

[9] K.D. Kissell, "MIPS16: High-Density MIPS for the Embedded Market," Proc. Real Time System '97 (RTS97), 1997.

[10] Mark Game and Alan Booker, "Codepack: Code Compression for PowerPC Processors", International Business Machines (IBM) Corporation, 1998.

[11] Richard Phelan, "Improving ARM Code Density and Performance: New Thumb Extensions to the ARM Architecture", ARM Limited, June 2003.

[12] S. Segars, K. Clarke, and L. Goude, "Embedded Control Problems, Thumb and the ARM7TDMI," IEEE Micro, vol.16, no.6, pp.22-30, 1995.

[13]Young-Jun Kwon, etc. "PARE: instruction set architecture for efficient code size reduction". Electronics Letters, Volume 35, Issue 24, 25 Nov 1999 Page(s):2098 – 2099.

Scheme	Methodology	Decoding	Compression ratio	Performance Penalty	Hardware Cost	Compiler complexity
ARM -Thumb	Extended ISA + mode Switching	Instruction mapping	20-30%	Very High	Thumb Engine	Low
ARM - Thumb-2	Separated ISA with Mapping Engine	Instruction mapping	15-25%	High	Thumb-2 instruction mapping Engine	High
MIPS - MIPS16e	Extended ISA + mode Switching	Native support	20-30%	Very High	Special branch detection engine	Low
IBM-CodePack	Binary Compression via software engine	Build-in de-compressor Engine	20-30%	Negligible	Hardware de-compressor	No effort
ARC-ARCompact	16-bit instruction support via User defined interface	Native support	20-40%	Negligible	Complex reconfigurable processor	Low
Proposed Heuristic Scheme	Native 16/32-bit mixed ISA	Native support	17-23%	Negligible to positive gain	Simple fetch handler	Low

Table 5: summary of different schemes