Integrated Instruction Selection and Register Allocation for Compact Code Generation Exploiting Freeform Mixing of 16- and 32-bit Instructions

Tobias J.K. Edler von Koch Igor Böhm Björn Franke

Institute for Computing Systems Architecture
School of Informatics, University of Edinburgh
Informatics Forum, 10 Crichton Street, Edinburgh, EH8 9AB, United Kingdom
T.J.K.Edler-Von-Koch@sms.ed.ac.uk, I.Bohm@sms.ed.ac.uk, bfranke@inf.ed.ac.uk

Abstract

For memory constrained embedded systems code size is at least as important as performance. One way of increasing code density is to exploit compact instruction formats, e.g. ARM Thumb, where the processor either operates in standard or compact instruction mode. The ARCompact ISA considered in this paper is different in that it allows freeform mixing of 16- and 32-bit instructions without a mode switch. Compact 16-bit instructions can be used anywhere in the code given that additional register constraints are satisfied. In this paper we present an integrated instruction selection and register allocation methodology and develop two approaches for mixed-mode code generation: a simple opportunistic scheme and a more advanced feedback-guided instruction selection scheme. We have implemented a code generator targeting the ARCompact ISA and evaluated its effectiveness against the ARC750D embedded processor and the EEMBC benchmark suite. On average, we achieve a code size reduction of 16.7% across all benchmarks whilst at the same time improving performance by on average 17.7%.

Categories and Subject Descriptors D.3 [*Programming Languages*]: Processors—Code Generation; D.3 [*Programming Languages*]: Processors—Compilers

General Terms Algorithms, experimentation, measurement, performance

Keywords Instruction selection, register allocation, code size, dual instruction set architecture, variable-length instructions, ARCompact

CGO'10, April 24–28, 2010, Toronto, Ontario, Canada.

1. Introduction

A large number of embedded processors are deployed in cost-sensitive, but high-volume markets where even modest savings of unit cost can lead to a substantial overall cost reduction. Highly integrated systems-on-chip (SoC) serve these markets and provide embedded processors as part of a more complex system integrated with other components such as memories and various peripheral devices on the same chip. Of all components, memories typically occupy the largest fraction of the chip area and, hence, contribute most to the overall cost. Among embedded memories,f flash storage plays a prominent role as non-volatile memory that needs to be large enough to store the full image of the binary executable. As a consequence, any reduction in code size translates directly to equivalent savings of die area and, eventually, unit cost. For example, 128-Mbit of NOR flash (two bits per cell) fabricated with $0.13\mu m$ design rules using a multilevel-charge (MLC) storage scheme occupy a chip area of $27.3mm^2$ [6] whereas an ARM Cortex-M3 processor (CM3Core core) only occupies $0.43mm^2$ at the same technology node [3].

One popular architectural approach to code size reduction is the provision of a compact instruction set architecture (ISA) alongside the standard, full-width RISC ISA. For example, some ARM processors (e.g. ARM7TDMI) implement the compact Thumb instruction set whereas MIPS has a similar offering called MIPS16e. Common to these compact 16-bit ISAs is that the processor either operates in 16-bit or 32-bit mode and switching between modes of operation is done through mode change operations which add a certain runtime and code size overhead. Furthermore, not all registers available in 32-bit mode are also accessible in 16-bit mode. Compilers seeking to take full advantage of compact instruction formats need to analyse the code and identify regions where the benefits of a compact instruction format outweigh its disadvantages (i.e. mode switching overhead and increased register pressure on the restricted register set). Sig-

nificant care needs to be taken as performance degradation by up to 98% as a result of an overly aggressive use of compact instructions has been reported in e.g. [11].

In this paper we consider the ARCompact [2] ISA implemented by the ARC 600 & 700 series of embedded processors. The ARCompact, ARM Thumb-2 [14] and also the recently introduced microMIPS [13] ISAs are different from other compact ISAs in that they allow freeform mixing of 16- and 32-bit instructions *without a mode switch*. This enables the compiler to generate a fine-grained interleaving of 16- and 32-bit instructions, thus potentially leading to greater code density without performance loss. However, at the same time the compiler needs to decide for *every* instruction that has a compact counterpart whether to emit the full-width or compact version. This decision does not only depend on the availability of an equivalent compact instruction, but additional register constraints need to be obeyed. Due to fewer instruction bits, 16-bit instructions comprise shorter register fields, and can only address eight rather than 32 registers for the standard 32-bit instructions in case of the ARCompact ISA. These register constraints demand an integrated approach where instruction selection and register allocation are considered simultaneously. Failure to do so may lead to missed opportunities or, even worse, to code bloat and performance loss due to excessive register-to-register data movement and, thus, negate the intended code size reduction effect.

1.1 Motivating Example

Before we discuss the proposed code generation approach in detail, we provide a motivating example that highlights some of the issues encountered in the selection of compact instructions.

Consider the small basic block in the first column of table 1. Suppose that in this block only registers *r2..r4* are available for use while all other registers contain values whose live ranges begin before and end after this basic block.

32-Bit Only	**Mixed Mode** (Aggressive)	**Mixed Mode** (Integrated)
`ld r2,[sp,0]`	`ld_s r2,[sp,0]`	`ld_s r2,[sp,0]`
`ld r3,[sp,4]`	`ld_s r3,[sp,4]`	`ld_s r3,[sp,4]`
	`mov r4,r1`	
`ld r4,[sp,8]`	`ld_s r1,[sp,8]`	`ld r4,[sp,8]`
`add r2,r2,r3`	`add_s r2,r2,r3`	`add_s r2,r2,r3`
`asl r2,r2,2`	`asl_s r2,r2,2`	`asl_s r2,r2,2`
`sub r2,r2,r4`	`sub_s r2,r2,r1`	`sub r2,r2,r4`
	`mov r1,r4`	
24 bytes	**20** bytes	**16** bytes

Table 1. Three versions of a sample basic block: 32-bit instructions only; aggressive use of 16-bit instructions; integrated instruction selection and register allocation avoiding register-to-register data movement.

Of these, *r2* and *r3* can be accessed by 16-bit instructions whereas *r4* cannot. We give three different versions of the basic block. The first one uses 32-bit instructions only (left column), this results in a code size of $6 \times 4 = 24$ bytes. For the second version (second column), we instructed the code generator to use 16-bit instructions – denoted by the suffix "_s" – wherever possible. This clearly leads to a problem: since these compact instructions can only operate on a limited set of registers, the register allocator has to insert additional move instructions to "spill" the value currently stored in *r1*, which is 16-bit accessible, to *r4* so it can use three 16-bit accessible registers, *r1..r3*. The aggressive use of compact instructions has increased the overall instruction count and the resulting code size is $6 \times 2 + 2 \times 4 = 20$ bytes. A better solution is presented in the third version of the basic block (third column). In that version, only some of the instructions have been replaced by their 16-bit counterparts. Those instructions operating on register *r4* remain 32-bit instructions. As a result "spilling" from the limited 16-bit accessible registers into the general-purpose registers is avoided and, thus, the instruction count remains constant while the code size is further reduced to $4 \times 2 + 2 \times 4 = 16$ bytes. This example demonstrates that instruction selection and register allocation need to be considered simultaneously to avoid excessive register traffic and to fully exploit the benefits of a compact ISA.

1.2 Contributions

Among the contributions of this paper are:

1. The development of an integrated instruction selection and register allocation framework targeting freely interleavable compact and standard RISC instructions for code size reduction,
2. the investigation of an opportunistic instruction selection procedure that obeys the stricter register constraints of the compact instruction format, but does not actively seek to maximise the usage of compact instructions,
3. the development of a more sophisticated instruction selection scheme that uses feedback information obtained from standard instruction selection and register allocation for improved compact code generation,
4. a demonstration of the practical integration of both schemes in a CoSy-based compiler targeting the ARCompact ISA,
5. an extensive evaluation against the industry standard EEMBC benchmark suite and a cycle-accurate simulator of the ARC750D processor, and
6. a comparison with GCC 4.2.1 targeting the ARCompact ISA on the same platform.

1.3 Overview

The remainder of this paper is structured as follows. In Section 2 we provide background information on the ARCompact

instruction set architecture and its particular ability of freeform mixing of 16- and 32-bit instructions without a mode switch as well as on the CoSy compiler development system [1]. This is followed by an in-depth description of our code generation approach in Section 3 where we develop two schemes for the selection of compact instructions under register constraints. In Section 4 we evaluate the effectiveness of our methodology and present extensive benchmarking results. The context of related work is established in section 5, before we summarise and conclude in section 6.

2. Background

In this section, we give an overview of the ARC750D processor and its instruction set architecture. This is followed by a brief presentation of those features of the CoSy compiler construction framework that are relevant to the work presented in this paper.

2.1 ARC750D and the ARCompact ISA

The ARC750D processor [2] is a configurable implementation of the ARCompact instruction-set architecture with the particular goal of creating an energy-efficient and easily extensible embedded microprocessor. ARCompact is a 32-bit RISC ISA supporting both 32-bit-wide and 16-bit-wide instructions. Both types of instructions can be intermixed freely without additional overhead as no mode-switch or decompression stage is required. A 32-bit memory value may contain either one 32-bit instruction, two 16-bit instructions or even a combination of half a 32-bit instruction and one 16-bit instruction. Thus, the frequent use of 16-bit instructions could reduce code size up to a theoretical maximum of 50%. In practice, this value is lower as not all instructions have a compact counterpart. ARC claim up to 40% code size reduction on their web site [2]. Obviously, compact instructions have certain limitations when compared to their 32-bit counterparts: in most cases, they only have access to a limited subset of the 32 core registers, namely the eight registers r0..r3 (callee-saved) and r12..r15 (caller-saved); the range for immediate operands is generally limited; and flags, such as carry, overflow, zero-result etc. are not set, to name a few. In table 2, we give examples of a few of the 32-bit *add* and the corresponding 16-bit *add_s* instructions to illustrate these limitations.

2.2 CoSy Compiler Construction Framework

Having seen the opportunities for code size reduction provided by 16-bit instructions, we will now describe the CoSy compiler framework [1] that has been used in the implementation of the proposed compact code generator.

CoSy provides a multi-pass compiler architecture where the compiler back-end operates on a *mid-level intermediate representation* (MIR), produced by the compiler's front- and mid-end. The code generator executes three principal passes, namely *Match/Cover*, *Register Allocation* and *Code Emission*. The *Match/Cover* pass computes a cost minimal covering of the MIR given the usual tree-pattern rewrite rules mapping IR nodes onto processor instructions. Each rule is associated with a – possibly dynamic – cost. According to the selected rules the MIR is transformed into a directed acyclic graph (DAG) *low-level intermediate representation* (LIR) that is annotated with pointers to the original MIR nodes. Later passes maintain and extend these annotations such that the user is able to trace back the creation point of any LIR node. We make use of this feature in Section 3.2 to implement feedback-guided instruction selection. A graph colouring-based register allocator maps pseudo-registers to actual registers or memory and is responsible for inserting spill code. The register allocator provided with CoSy can handle several overlapping register classes and we use this feature for the implementation of separate, but non-disjoint registers classes for 16-bit accessible and globally accessible registers, respectively. Finally, assembly code is emitted for each LIR node using code emission rules for each type of LIR node.

Instruction	dest	src1	src2
`add a,b,c`	any reg	any reg	any reg
`add a,b,u6`	any reg	any reg	6-bit unsig. imm.
`add a,b,limm`	any reg	any reg	long immediate
`add_s a,b,c`	16bit reg	16bit reg	16bit reg
`add_s a,b,u3`	16bit reg	16bit reg	3-bit unsig. imm.
`add_s a,a,limm`	16bit reg	(dest)	long immediate

Table 2. Comparison of 32-bit and 16-bit formats for the *add dest,src1,src2* instruction.

2.3 Naïve ARCompact Code Generation using CoSy

The obvious approach to add support for 16-bit instructions to the CoSy compiler framework would be to add new tree-pattern rules for this class of instructions with lower costs than their 32-bit counterparts. The compiler would then select 16-bit instructions wherever possible and we would hope that this measure has the desired effect on code size. Such a naïve scheme, however, has serious shortcomings that are due to register allocation occurring after the *Match/Cover* pass and register access constraints of 16-bit instructions as described above. By using 16-bit instructions almost exclusively, we effectively reduce the size of the available register set to just a small subset of the general-purpose registers available to 32-bit instructions. This inevitably leads to higher register pressure and results in an increased number of register-to-register data moves or even spills to memory. Thus, even though we might be able to reduce code size initially, these additional instructions will negate the benefits from using a compact instruction format. In addition, extra data moves and spills will most certainly degrade performance and are therefore highly undesirable.

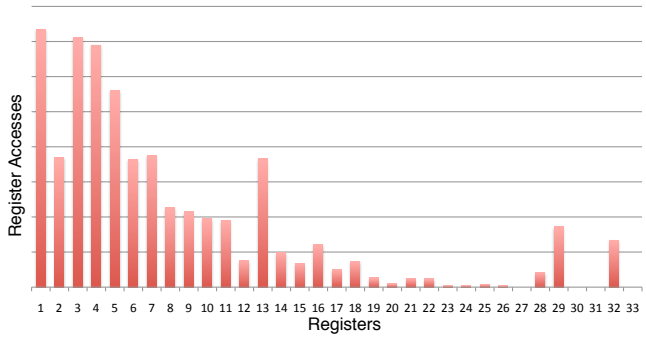

Figure 1. Typical distribution of dynamic register accesses.

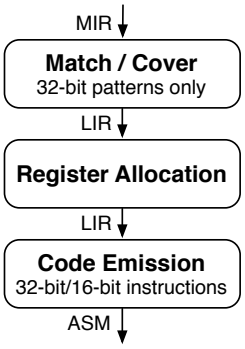

Figure 2. Sequence of stages of the *opportunistic* scheme for mixed-mode instruction selection.

3. Methodology

From our discussion in Section 2.3 it becomes evident that we need to balance the use of 16-bit instructions against the increase in register pressure resulting from their restricted register addressing capabilities. Keeping in mind our goal of reducing code size as much as possible, we therefore want to maximise the use of 16-bit instructions while minimising the amount of move and spill instructions inserted at the register allocation stage solely because of the use of 16-bit instructions. In the following sections, we propose two methods to achieve this. The first one, the opportunistic use of 16-bit instructions, is implemented purely at the code emission stage and requires no further changes to the compiler framework. The second one, the selective use of 16-bit instructions directed by MIR annotations using code generator feedback, is more involved but less dependent on specific traits of the ARCompact ISA and expected to produce better results.

3.1 Opportunistic Instruction Selection

Our first compact instruction selection scheme is motivated by the non-uniform distribution of register accesses that favours registers with lower ID over those with higher ID. An example distribution is shown in Figure 1. It is clearly visible that the lower part of the register set is accessed much more frequently than the upper part. This is partly due to the fact that the standard graph colouring register allocator always selects the register with the lowest ID of the set of possible registers for a value. In addition, calling conventions dictate that arguments are passed in registers r0..r7 and the result is returned in register r0. Finally, immediate values tend to be small and often fall within the limited bounds 16-bit instructions can operate on.

We exploit the fact that 16-bit accessible registers are already frequently used in standard 32-bit code and construct an *opportunistic* instruction selector that does not aim to specifically identify compact instructions in the match/cover stage, but delays this decision until after standard 32-bit instruction selection and register allocation. Only at the code emission stage do we check if an instruction that possesses a 16-bit counterpart *incidentally* satisfies the stricter register constraints of that compact instruction. If this is the case we emit a short 16-bit instruction, otherwise we emit a standard 32-bit instruction. In either case, we leave the register allocation intact as only the type of instruction is affected. Figure 2 summarises this approach schematically. Certainly, this approach will only work because of the coincidental match between frequently used registers and those registers accessible by 16-bit ARCompact instructions. Still, this opportunistic scheme avoids the main drawback of the naïve solution from Section 2.2, namely the introduction of additional register moves and spilling. In addition, this compact code generation scheme has low implementation complexity and only requires local changes to the code emission pass.

3.2 Feedback-Guided Instruction Selection

The opportunistic instruction selection scheme presented in the previous section has a number of shortcomings. First, we rely on the coincidence that standard graph colouring results in code that frequently satisfies the stricter register constraints of the 16-bit ARCompact instructions. Other architectures may embed registers accessible by short instructions elsewhere in the register set and our simple opportunistic approach will become less effective. Second, it seems conceptually unsound that 16-bit instructions are handled outside the cost framework of the match/cover stage and defy standard instruction selection and register allocation. For this reason, it appears likely that we may miss further opportunities for the use of 16-bit instructions.

We therefore propose an improved approach based on code generator feedback, which is illustrated in Figure 3. In summary, we run two iterations over the match/cover and register allocation stages. The purpose of the first iteration is to attempt aggressive 16-bit code generation including instruction selection and register allocation. At the end of this first iteration feedback is gathered and made available to the second iteration of code generation. During the second code generation stage the feedback is then used to determine and annotate register pressure points in the MIR where 16-bit instructions should be avoided. This allows for the feedback-guided, selective use of 16-bit instructions in the second iteration and, thus, avoids the introduction of register pressure related register-to-register move operations.

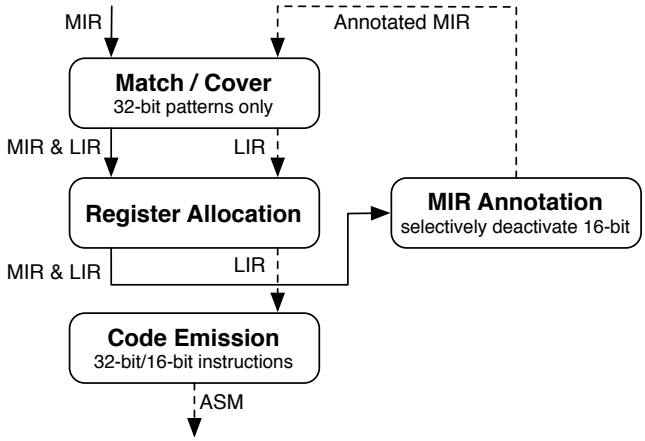

Figure 3. Feedback-based approach for mixed-mode instruction selection.

In the following paragraphs we explain the two phases – IR *Annotation* and *Feedback-guided Code Generation* – in detail:

I. IR Annotation.

1. **Match/Cover.** In contrast to the opportunistic approach, we add all 16-bit instructions as separate tree-pattern matching rules to the match/cover stage with costs lower than those of the corresponding 32-bit instructions, i.e. we use both 16- and 32-bit rules with a preference for the 16-bit ones due to their lower cost.
2. **Register Allocation.** The aggressive use of 16-bit instructions will most likely introduce a large number of additional register moves and spills during register allocation. When such additional instructions are inserted into the LIR, the register allocator inserts backpointers to the MIR nodes that "caused" them. For example, suppose that one of the operands for a 16-bit *add* instruction is originally located in a register that is not 16-bit accessible. In this case, the register allocator will insert an additional register move operation to place the value in a register accessible by short instructions. This may result in more *move* operations to be inserted to free the temporary space in the 16-bit accessible register set. All of these move operations will be inserted before the actual *add* instruction node and links will be set up pointing to the newly created auxiliary *move* nodes. After register allocation, we can then trace back exactly which MIR nodes were at the origin of additional moves or spills.
3. **MIR Annotation.** In this pass we iterate over all LIR nodes and identify those moves and spills that were directly caused by 16-bit instructions based on a number of simple tree patterns. Whenever such a node is found, we use the backpointer inserted during the previous register allocation pass and determine the 16-bit instruction node it relates to. This in turn allows us to locate the MIR node that gave rise to it using the MIR-LIR links inserted at the match/cover stage (see Section 2.2). We can then annotate this MIR node with an additional "no 16-bit" flag indicating that the node should not be covered with a 16-bit instruction rule so as to avoid additional move or spill instructions. Finally, the LIR produced in this iteration is discarded.

II. Feedback-guided Code Generation.

1. **Match/Cover.** In the second iteration, the match/cover stage uses MIR annotations to selectively disable the use of 16-bit instruction rules for nodes flagged in the previous IR annotation stage. This is achieved by specifying special CONDITION clauses available in the code generator description language that should hold for rules to be applied.
2. **Register Allocation.** We expect that the register allocator will insert significantly fewer, if any, additional moves and spills caused by 16-bit instructions in this second iteration. While it is possible that new moves or spills will appear in different places, this is unlikely since most of the register pressure actually results from temporary variables that have short live ranges within a basic block.
3. **Code Emission.** In this stage, we perform standard code emission targeting 16- and 32-bit rules. As before in the opportunistic scheme, we may find that even for some 32-bit instruction nodes marked with the "no 16-bit" flag the stricter register constraints for the use with 16-bit instructions are incidentally satisfied. Thus, we additionally employ the opportunistic scheme and use 16-bit instructions wherever possible as doing so does not incur any extra cost.

For the example shown in Table 1 we would initially generate the code in the second column, however, without actually emitting this code (see Figure 4). Next, we would identify the two inserted `mov` instructions as ones that have been introduced by the register allocator to free space in the 16-bit accessible register range for the `ld_s r1,[sp,8]` and `sub_s r2,r2,r1` instructions, respectively. We mark these two short instructions with the "no 16-bit" flag, before entering the actual feedback-guided code generation stage. When we encounter the two instructions again, we check for the "no 16-bit" flag and decide not to emit compact instructions, but to resort to their standard 32-bit versions. Finally, after the subsequent run of the register allocator the shorter code sequence shown in the third column of Table 1 is emitted.

It is important to note that although we are using feedback to guide instruction selection in our compact code generation methodology, this feedback information is generated statically within the compiler and unlike [10, 11, 12] does not require any profiling.

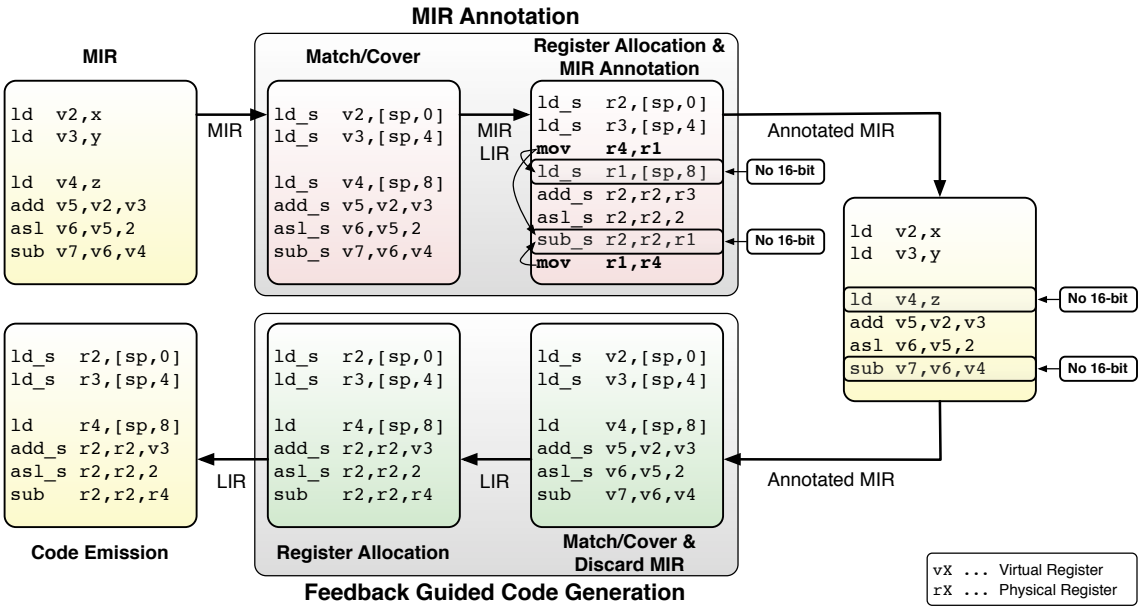

Figure 4. Feedback-guided compact code generation for the example in Table 1.

4. Empirical Evaluation

We have extensively evaluated our integrated code generation approach and in this section we describe our experimental setup and methodology before we present and discuss our results.

4.1 Experimental Setup and Methodology

We have evaluated our compact code generation approach against the EEMBC 1.1 benchmark suite [7] that comprises applications from the automotive, consumer, networking, office and telecom domains. All codes have been built with our highly optimising compiler based on the commercial CoSy compiler development system [1]. Our main interest has been on code size, therefore we have measured code size, i.e. the total `size` of the `.text` segments of all object files forming a single benchmark (except for the benchmark harness), using the UNIX tool size. In addition, we have measured the performance impact resulting from our code generation methodology using a verified, cycle-accurate simulator of the ARC750D processor. Table 3 lists the configuration details of our target processor.

Finally, we have also compiled the full set of benchmarks with the ARC port of the GCC 4.2.1 compiler with full optimisation and mixed code generation enabled (*'-mA7 -O3 -mmixed-code'*) in order to compare the benefits of exploiting the compact instruction format of the ARCompact ISA with GCC to our approach.

Throughout this section, plain 32-bit code (featuring 32-bit instructions exclusively) serves as the baseline. This code is generated by our CoSy-based compiler for the discussion of our compact code generation schemes, and by GCC for the evaluation of GCC's approach so that relative improvements due to mixed-mode code generation become comparable for both compilers.

Core	ARC750D
Pipeline	7-stage (interlocked)
Execution Order	In-Order
Branch Prediction	Yes
ISA	ARCompact
Floating-Point	Hardware
Memory System	
L1 Cache	
Instruction	8k/2-way associative
Data	8k/2-way associative
L2 Cache	None
Bus Width/Latency/Clock Divisor	32-bit/16 cycles/2
Simulation	
Simulator	Full-system, cycle-accurate
Options	Default
I/O & System Calls	Emulated

Table 3. Configuration of the ARC750D simulator.

4.2 Code Size

We initially discuss code size reduction due to the selection of compact instructions as this has been the primary motivation for our work. Our overall code size results are summarised in the diagram shown in Figure 5. For each of the EEMBC benchmarks from the automotive, consumer, networking, office and telecommunication domains we present three results relating to the percentages of code size improvements resulting from (a) our opportunistic scheme, (b) our feedback-guided instruction selection scheme, and (c)

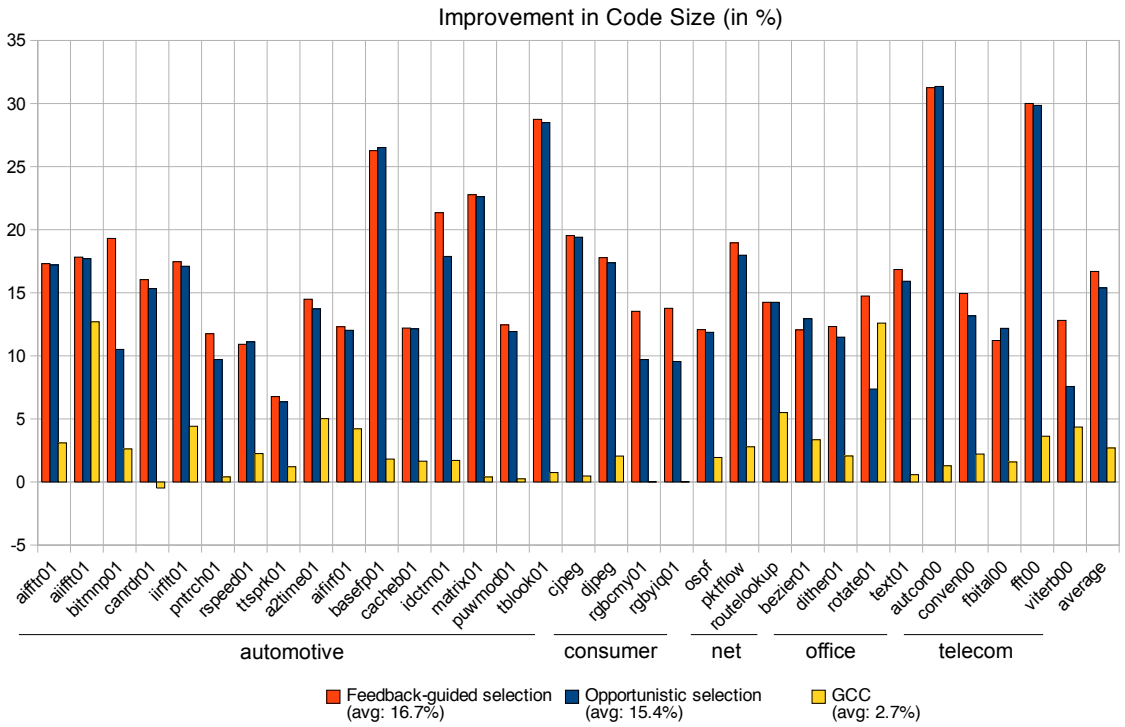

Figure 5. Code size improvements due to compact code generation for the EEMBC 1.1 benchmarks (baseline: 32-bit only).

the ARC port of the GCC 4.2.1 compiler. We will discuss the GCC results separately in Section 4.4.

For the opportunistic scheme code size reductions between 6.4% and 31.4% are achieved, with an average improvement of 15.4%. For individual programs (`fft00` and `autocor00`) code size improvements of 30% are reached (30.0%) or exceeded (31.4%). This result is slightly surprising given that the opportunistic scheme does not actively seek to maximise the use of 16-bit instructions, but only identifies and converts 32-bit instructions that already satisfy the stricter register constraints of the corresponding 16-bit instructions. Only 6 out of 32 applications fail to deliver code size reductions of more than 10% for this simple scheme, while 14 benchmarks are improved by more than 15%. This suggests that our initial assumption, namely that in standard 32-bit code most register traffic is already handled in 16-bit accessible registers, holds in general (see Figure 1). Any scheme trying to improve on this would need to identify further opportunities among the few remaining instructions that operate on non-16-bit accessible registers whilst the 16-bit accessible registers are not fully utilised.

With this information it is not surprising that our feedback-guided instruction selection scheme performs only slightly better than the simpler opportunistic scheme. We observe that although the average improvement (16.7%) is not much higher than for the simple scheme (15.4%) feedback-guided instruction selection provides *more consistent* improvements across the range of benchmarks and reduces the standard deviation from the mean from 6.45% to 5.86%. In particular, our more advanced scheme produces denser code for benchmarks such as `bitmnp01`, `pntrch01`, `rgbcmy01`, `rgbyiq01` and `viterb00` where the simple scheme does not perform too well. Still, both schemes perform similarly on those applications that provide the greatest opportunities for code size reductions (`autocor00`, `fft00`, `tblook01`, `basefp01`).

It is important to note that for both of our compact code generation schemes there is not a single case where the code size has been increased as this is impossible for either of the two algorithms.

ARC claims up to 40% code size reduction for the extensive use of compact instructions [2], however, this represents a theoretical upper bound only achievable if compact instructions are used for virtually all operations and all live data fits into the small 16-bit accessible register subset. In practice, our code generation methodology comes close to this theoretical limit for a number of applications (e.g. `autocor00` and `fft00`) while for the majority of codes we reach about 45% of the hypothetical peak code size savings.

4.3 Performance

Next we evaluate the performance impact of our compact code generation methodology. A summary of our results is shown in the diagram in Figure 6.

Across the range of benchmarks the opportunistic scheme reduces the cycle count by 16.6% on average. For individual applications (`conven00` and `viterb00`) performance im-

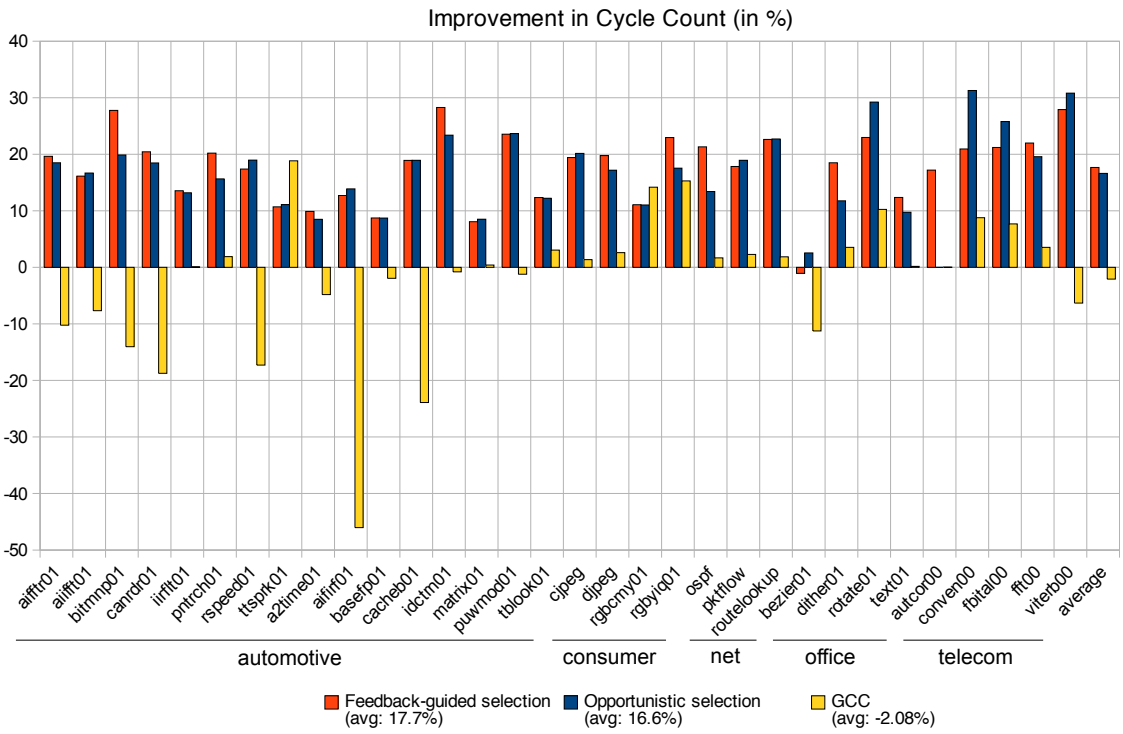

Figure 6. Performance improvements due to compact code generation for the EEMBC 1.1 benchmarks (baseline: 32-bit only).

provements in excess of 30% can be observed. Only 5 out of 32 applications are improved by less than 10%, whereas 7 programs are sped up by more than 20%. Not a single program suffers from performance degradation as a result of opportunistic compact code generation.

Similar to the code size evaluation in the previous Section, feedback-guided instruction selection gives a slight improvement over the simpler opportunistic scheme. On average, performance is improved by 17.7%. Unlike the case with code size, feedback-guided instruction selection does not perform better across the entire set of applications, but there are benchmarks where one of the two schemes clearly outperforms the other. For example, the opportunistic scheme produces faster code for the `rotate01`, `conven00`, `fbittal00` and `viterb00` benchmarks whereas the feedback-guided scheme takes a lead on the `bitmnp01`, `pntrch01`, `idctrn01`, `rgbyiq01`, `ospf` and `autocor00` codes. For a single program (`bezier01`) the feedback-guided scheme results in a negligible performance loss of 1.06%.

We have investigated the origin of the performance gains observed in our experiments and found that these largely result from improved instruction cache behaviour due to increased code density of the mixed-mode code. With 8kB the size of the level-1 instruction cache in our processor is relatively small, but representative for typical low-power and low-cost embedded processor cores. Any code size reduction technique will inevitably lead to higher instruction cache hit rates for any non-trivial application. For the one application where we see a minor performance degradation we have found that this is caused by an increased number of instruction cache conflicts. The conflicts can be avoided if we configure the ARC750D processor to contain a 4-way associative instruction cache rather than the 2-way associative cache that has been used in our experiments.

The reason why most of the prior work in this field [9, 10, 11, 12, 16] has reported performance losses or, at best, the same level of performance for mixed-mode code is that the mode switching overhead associated with the Thumb, Thumb-2 and MIPS16e ISAs exceeds the performance gains from a reduced number of instruction cache misses. Due to the ability to freely interleave compact and standard instructions without mode switch the ARCompact ISA does not suffer this penalty.

4.4 Comparison to GCC

The diagrams in figures 5 and 6 contain additional data points for the official ARC port of the GCC 4.2.1 compiler targeting the ARCompact ISA and ARC750D processor.

The comparison of code size improvements when going from plain 32-bit code to mixed-mode code reveals that GCC performs significantly worse than either of our two schemes. On average, mixed-mode code generation in the GCC compiler results in a modest 2.7% reduction in code size over the 32-bit baseline. For two programs the code size reduction exceeds 10%, however, for the majority of programs code size savings of less than 5% are the norm. The poor performance

Publication	Comment	Platform (Architecture/ISA)	Compiler	Benchmarks (Name/Number)	Code Size Red. (Avg./Range)	Performance Gain (Avg./Range)
[9]	Repeated instruction selection, no caches, 1-cycle mem. access	Custom MIPS4000/ Modified MIPS16	EXPRESS	Livermore Loops/22	38%/ 31%-49%	-6%/ up to -24% (=loss)
[10, 11]	Profile-guided IS, SimpleScalar	StrongARM SA-110/ ARM-Thumb	XScale GCC 2.9	MediaBench/12	28.2%/ 22.8%-31.9%	-2.03%/ up to -14.5% (=loss)
[12]	Profile-guided IS, code size levels	Intel XScale PXA250	Zephyr/VPO	MiBench & MediaBench/4	34.5%/ up to 42.1%	0% to -49% (=loss)
[16]	ISA not fixed, but app.-specific	UniCore (proprietary)	GCC 3.2.1 (Post-pass)	MediaBench/15	16%/ 14%-18%	0%/ up to -2% (=loss)

Table 4. Comparison to other published results for mixed-mode code generation.

of the GCC compiler is due to the non-integrated instruction selection and register allocation stages where 16-bit instructions are generated without taking into consideration the increased register pressure on the restricted, 16-bit accessible register subset.

The performance results for the GCC compiler shown in Figure 6 are even more disappointing. On average, mixed-mode applications generated by GCC perform 2.08% *worse* than their 32-bit counterparts. 13 out of 32 programs suffer from performance degradation with individual benchmarks (`aifirf01`) taking 46.1% *more* cycles to execute. At the same time the possible performance gains that can be observed for the remaining programs are limited and do not come close to those resulting from our proposed techniques. For only two programs (`ttsprk01` and `rgbcmy01`) GCC delivers higher speed improvements than either of our two schemes. Overall, we feel the very modest code size improvements achieved by GCC do not justify the potentially large drop in performance. This may present a serious obstacle to the widespread adoption of mixed-mode GCC code generation by code size and performance aware users.

5. Related Work

Compact instruction sets are supported by a number of commercial embedded processors and code size aware compilation techniques targeting these short instruction formats have found significant interest in the scientific community [9, 10, 11, 12, 16]. All of these papers address mixed mode code generation where there is an overhead associated with switching between compact and standard instruction sets. In this paper, however, we are concerned with an ISA that allows freeform mixing of 16- and 32-bit instructions and to the best of our knowledge this problem has not yet been addressed in the academic literature.

A direct comparison of our results to those published elsewhere is difficult due to different choices of benchmarks, compiler frameworks and not least instruction set architectures. Nonetheless, we juxtapose the available results on code size reduction and performance impact along with relevant details relating to the architectures and ISAs, compilers, and benchmarks in Table 4.

In [9] a modified MIPS16 ISA is targeted and a compiler flow with repeated instruction selection passes is proposed. After the first instruction selection stage targeting a generic 3-address ISA a profitability analysis is performed that guides the second instruction selection stage in its selection of compact instructions. In a final step register allocation takes place. This work is similar to our approach in that compiler feedback is used to improve compact code generation, however, the main difference is that register allocation is not taken into account and, as a consequence, excessive spilling may occur. The single-cycle memory access model, however, almost eliminates the spilling cost. This and the trivial Livermore loops benchmarks may lead to overly optimistic performance results, which still show an overall slowdown. The reported code size reduction, however, is impressive, but requires further evaluation on more representative benchmarks.

Feedback information is used in [10, 11] for the selection of ARM and Thumb instructions. Frequently executed functions are identified via profiling before a heuristic based on expected performance and relative code size is used to choose between 32-bit ARM and 16-bit Thumb instructions. In contrast, our approach does not require this expensive profiling stage and operates on purely static information available in the compiler. Overall, code size reduction is good, but the higher code density does not translate to performance gains resulting from improved instruction cache behaviour.

Instruction set assignment is determined on a per-basic-block basis in [12]. This requires a profile-guided control flow analysis to determine the program points where the processor's execution mode should be switched. An effort level is chosen depending on a code size budget set by the user. In order to avoid excessive mode switching overhead a detailed profitability analysis that estimates the cost and benefit of using different instruction sets for different parts of a given program is required. As a secondary goal the impact of using a compact instruction format on the worst-case execution time is analysed. Again, we do not rely on profiling, but employ static feedback information within the compiler. While the overall code size reduction is good for four small applications and the most aggressive code size

budget, the performance penalty can be drastic. For more relaxed settings, the performance loss can be compensated, but at the same time code size savings drop sharply.

A slightly unconventional approach to compact code generation is taken in [16] where mixed mode code generation is performed as a post pass to the actual compiler. In addition, the semantics of the chosen compact ISA is changed in order to efficiently encode mode changing operations. This results in good code size reductions without incurring any significant performance loss. In contrast, we target a fixed ISA requiring no architectural changes whilst reducing code size and, at the same time, *improving* performance across the entire range of benchmarks.

In summary, those techniques in [9, 10, 11, 12, 16] that achieve higher code size reductions than our approach result in significant performance losses. For [16] the code size gains are comparable, but we generate faster code and do not rely on hardware modifications.

The method in patent [8] describes a compilation technique targeting compact instructions of the ARCompact instruction set where a block of code is compiled for *both* standard 32-bit and compact 16-bit instruction sets. Eventually, whichever version is "better" gets chosen. In this scheme compact code generation is aggressive, however, whenever spilling occurs a "clean up" operation is performed to offload memory references to the general-purpose register set. This scheme is largely identical to the aggressive scheme shown in Table 1 with all its disadvantages.

MIPS and ARM state code size savings up to 35% and 26%, respectively, for their microMIPS [13] and Thumb-2 [14] ISAs while performance drops by 2%. A direct comparison of the Thumb, Thumb-2, MIPS16e and ARCompact ISAs is given in [15]. Design trade-offs for microprocessors with variable length instructions are subject of [5] and general code-size reduction methods are surveyed in [4].

6. Summary and Conclusions

We have developed an integrated instruction selection and register allocation methodology for a compact ISA with freely interleavable 16- and 32-bit instructions. We have presented a simple, yet powerful opportunistic instruction selection scheme and a more general, feedback-guided scheme that is more portable and does not, unlike the first scheme, depend on a particular distribution of register accesses. Experimental results based on our highly optimising CoSy compiler targeting the ARC750D processor demonstrate that an average code size reduction of 16.7% can be achieved across the industry standard EEMBC benchmarks whilst at the same time improving performance by on average 17.7%. This result is encouraging and shows that code size reduction and optimisation for performance are not mutually exclusive, but can be obtained simultaneously.

Future work will include the power/energy evaluation of our compact code generation methodology.

References

[1] ACE Associated Computer Experts bv. CoSy compiler development system. `http://www.ace.nl`, retrieved 12 August 2009.

[2] ARC International. ARC750D Core. `http://www.arc.com`, retrieved 12 August 2009.

[3] ARM Ltd. ARM Cortex-M3. `http://www.arm.com`, retrieved 12 August 2009.

[4] Árpád Beszédes, Rudolf Ferenc, Tibor Gyimóthy, André Dolenc, and Karsisto, Konsta. Survey of code-size reduction methods. ACM Computing Surveys, Vol. 35, No. 3, pp. 223–267, 2003.

[5] John Bunda, Don Fussell, W.C. Athas, and Roy Jenevein. 16-bit vs. 32-bit instructions for pipelined microprocessors. SIGARCH Computer Architecture News, Vol. 21, No. 2, pp. 237–246, 1993.

[6] Dave Bursky. Nonvolatile Memory: More Than A Flash In The Pan. In *electronic design*, `http://electronicdesign.com`, ED Online ID 5267, July 2003.

[7] The Embedded Microprocessor Benchmark Consortium. EEMBC Benchmark Suite. `http://www.eembc.org`, retrieved 12 August 2009.

[8] Richard A. Fuhler, Thomas J. Pennello, Michael Lee Jalkut, and Peter Warnes. Method and Apparatus for Compiling Instructions for a Data Processor. United States Patent US 7278137B1, Oct. 2, 2007.

[9] A. Halambi, A. Shrivastava, P. Biswas, N. Dutt, A. Nicolau. An Efficient Compiler Technique for Code Size Reduction Using Reduced Bit-Width ISAs. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, p. 402, 2002.

[10] Arvind Krishnaswamy and Rajiv Gupta. Profile guided selection of ARM and Thumb instructions. ACM SIGPLAN Notices, Vol. 32, No. 7, pp. 56–64, 2002.

[11] Arvind Krishnaswamy and Rajiv Gupta. Mixed-width instruction sets. Communications of the ACM, Vol. 46, No. 8, pp. 47–52, 2003.

[12] Sheayun Lee, Jaejin Lee, Chang Park, Sang Min. Selective code transformation for dual instruction set processors. ACM Transactions on Embedded Computing Systems, Vol. 6, No. 2, 2007.

[13] MIPS Technologies. microMIPS Instruction Set Architecture. MD00690, Revision 01.00, October 2009.

[14] Richard Phelan. Improving ARM Code Density and Performance – New Thumb Extensions to the ARM Architecture. ARM Thumb-2 Core Technology Whitepaper, June 2003.

[15] Jim Turley. Code compression under the microscope. In *Embedded Systems Design*, `http://www.embedded.com`, retrieved 12 August 2009.

[16] Liu Xianhua, Zhang Jiyu, Cheng Xu. Efficient code size reduction without performance loss. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pp. 666–672, 2007.