Adaptive Reduced Bit-width Instruction Set Architecture (adapt-rISA)

Sandro Neves Soares[1], Ashok Halambi[2], Aviral Shrivastava[3], Flávio Rech Wagner[4] and Nikil Dutt[2]
[1]Universidade de Caxias do Sul - Brazil, [2]University of California at Irvine – United States, [3]Arizona State University - United States, [4]Universidade Federal do Rio Grande do Sul – Brazil
snsoares@ucs.br, halley@ics.uci.edu, Aviral.Shrivastava@asu.edu, flavio@inf.ufrgs.br, dutt@uci.edu

***Abstract* — rISA (reduced bit-width Instruction Set Architecture) is an important architectural feature to reduce code size, which continues to be an extremely important concern for low-end embedded systems. rISA reduces code size by expressing parts of the application in terms of low bit-width instructions. ARM-Thumb, ARCcompact and MIPS16/32 are popular examples. With the intent to exploit the dynamically changing "working instruction set" of today's complex software, ARM 11 now comes with two rISAs, which can be interleaved in the application binary. However, it was demonstrated that the code compression achieved by rISA is extremely sensitive on the selected rISA design. Therefore, it is important to design the optimal rISA for a given embedded application. The *one optimal rISA per application* approach has already been explored by previous works. In this paper, we present a scheme to design a *multiple rISA architecture* for embedded systems. Our experiments on MiBench report an average of 19% code compression and up to 7% power reduction of instruction memory when compared to previous approaches using only one optimal rISA.**

I. Introduction

Code size continues to be an extremely important concern for low-end embedded systems, where the system power and performance is dominated by the RAM size. Although a clear majority of embedded processors manufactured each year fall into this category, they remain invisible to human eyes, but aiding us in our daily lives in the form of controllers in cars, TVs, refrigerators, and music players. While employing RISC processors to design modern embedded systems is preferred, since they provide increased design flexibility using a simpler, low power core, one of their fundamental drawbacks is the poor code density. For such systems, a higher code size can imply the impossibility to execute the functionality, in the worst case, and a significant impact on the system power and cost, in the best case. The problem is becoming complex with the current trend of increasing software content on embedded systems by 10X per decade.

rISA (reduced bit-width Instruction Set Architecture) is an effective and popular solution to this code size problem. Architectures with rISA have two instruction sets, the “normal” set, which is the original 32-bit instruction set, and the “reduced bit-width” instruction set, which encodes the most commonly used instructions in 16-bit narrow instructions. By expressing parts of the application using the "reduced bit-width" ISA, significant code size reduction can be achieved. In addition, since the fetch-width of the processor remains the same, the processor when operating in rISA requires less fetches to the instruction memory, thus saving memory energy [1]. However rISA architectures that have just one "reduced bit-width" ISA are unable to exploit the dynamically changing "working instruction set" of today's embedded applications. This is true not only because of the increasing complexity of software, but also because a "reduced bit-width" ISA can have only a very limited number of instructions due to bit-width constraints. Realizing this, the new ARM 11 architecture [2] comes with two "reduced bit-width" ISAs, which can be interspersed in the program. Furthermore, the code compression and power reduction obtained by this dual instruction set technique is heavily dependent on the application computational requirements and on the narrow instruction set design. Consequently, previous works about rISA suggested techniques to design the best "reduced bit-width" architecture for a given (set of) application(s). However they only solve the problem for single "reduced bit-width" ISA architectures. If the focus is now changed to develop dual "reduced bit-width" ISA for architectures such as ARM 11, the different computational requirements inside a single application should be considered and, in response, the rISA parameters should be adapted accordingly.

As far as we are aware, this is the first effort to automatically design "reduced bit-width" ISAs for multiple rISA architectures. Our approach *adapt-rISA* is aware of the potential different computational requirements inside a single embedded application, adapting the rISA parameters to them at compilation time. At run time, static reconfiguration capabilities are available to correctly decode these reduced instructions, created using different rISA parameters. This way, *adapt-rISA* achieves better results than rISA (as explored by previous works, i.e., with only one optimal rISA per application), both on code compression (19% on average) and on power reduction (up to 7% less fetch requests). In addition, this work also employs a new rISA design that may simplify the translation unit implementation, a block necessary to translate reduced to normal instructions at run time.

II. rISA Architectural Feature

The normal code and the corresponding reduced code of a small section of the CRC32 program of the MiBench benchmark [3] are shown in Figure 1 – MIPS 16/32 architecture. The reduced code in Figure 1b constitutes a block of reduced instructions or a reduced block. A program, compiled using a rISA compiler, is composed by several of these reduced blocks and also by blocks containing only normal instructions (the normal blocks). The change mode

instructions and the reduced nops (`rISA_nop`), present in the figure, are explained in detail later in this text.

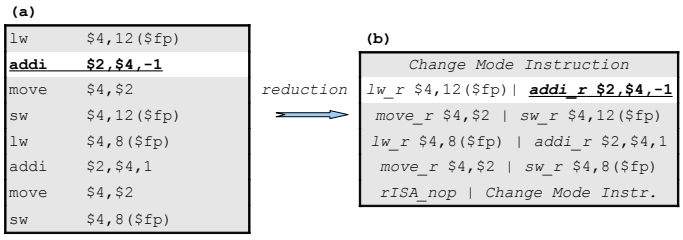

Figure 1. Normal code (a) and Reduced code (b)

If $\mathcal{O}$ is the set of different opcodes in the application code and N is the cardinality of this set, for each of the opcodes in $\mathcal{O}$, there are X_i occurrences of instructions in the code, with $X_i > 0$ and $0 \le i \le N-1$. Since there are, in general, fewer bits in the reduced instructions to specify the opcode, a *conversion-to-rISA* algorithm will employ a set Γ, subset of $\mathcal{O}$. For each of the opcodes in Γ, whose cardinality is n, being $n < N$, all the corresponding x_i occurrences of instructions will be marked to be reduced, with $x_i \ge 0$ and $0 \le i \le n-1$. Actually, not all the x_i occurrences of instructions of the opcodes used, or selected, by the *conversion-to-rISA* algorithm can be reduced; some of them must be discarded from reduction. This way, only x'_i occurrences of a given selected opcode are actually reduced, being $x'_i <= x_i$. The role of a rISA compiler is to find the best **rISA design** and also the best **rISA design configuration** to maximize the number of reduced instructions in the final executable code, i.e., to maximize the $\sum_{0}^{n-1} x'_i$. We call a rISA design the information related to the number of bits reserved to each field of the reduced instruction, which includes the definition of n, the cardinality of Γ, and also the use (or not) of special reduced instructions, employed to support the conversion process, as the instruction `rISA_extend` to complete immediate values. A rISA design configuration specifies the combination of different opcodes from $\mathcal{O}$ in the subset Γ, whose occurrences will be initially marked for reduction in the application code. If, for example, the rISA design (***rd***) employs four bits to specify the opcode, each corresponding rISA design configuration (***rdc***) will have its own set Γ, containing 16 ($n = 16$) different opcodes from $\mathcal{O}$. If the goal is to increase code density, ***rdc*** must include the most frequently encountered instructions in the code, but if the goal is power reduction, the most executed instructions must be selected.

Figure 2 shows the translation of the `addi` instruction, present in the code of Figure 1, from 32 bits (Figure 2a) to 16 bits (Figure 2b), using the rISA design ***rISA_4444***. This design uses four bits to specify, respectively, the opcode, the source register (rs), the target register (rt) and the immediate value (imm). Since the values are not semantically altered, the instruction can be reduced. The reduced opcode is arbitrary, but it must be converted to the original opcode during the decode phase at run time. During this phase, rISA processors dynamically expand the reduced instructions into their corresponding normal instructions. Only normal instructions are actually executed. Usually, each reduced instruction has a corresponding instruction in the normal set. This simple and direct transformation demands only a simple translation unit in the decode logic of the processor. No other hardware module is necessary.

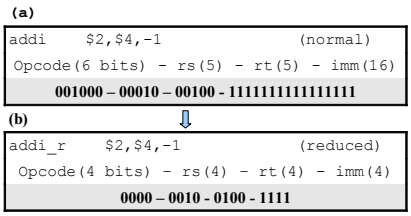

Figure 2. addi instruction: Normal (a) and Reduced (b)

III. Related Work

In the contemporary embedded processors market, important companies, such as ARM [2] and ARC [4], implement rISA features in their products. [5, 6] present a design space exploration framework for rISA design aimed at improving code density. The experiments employ various rISA designs (one design and one configuration per application) working with a minimum of 16 and a maximum of 128 opcodes ($n = 16$ to $n = 128$). Four to seven bits were required to specify the opcode in these designs. The works show that the rISA design ***rISA_4444*** presents the best results when balancing code compression and the complexity of the translation process at run time. If a normal instruction, selected by a ***rISA_4444*** design configuration, cannot fit on a reduced instruction, without losing information, it is simply discarded from reduction. Some other rISA designs solve this problem adding special reduced instructions to the application original code: `rISA_extend` to complete immediate values is an example. Other examples are the special reduced instructions employed to perform spills (either to memory or to non-rISA registers) and reloads of values in registers, due to the limited availability of registers by the reduced instructions.

The focus of [1] is energy reduction using rISA. It is shown that a *conversion-to-rISA* algorithm aimed at improving code density does not achieve the best results in terms of energy reduction, because it does not consider the dynamic aspects of the application execution. An algorithm that is aware of this dynamic behavior is presented and applied on a MIPS processor model. This algorithm selects the most executed instructions, instead of the most frequent in the application code. An average of 26% reduction in the number of fetches to the instruction memory is reported.

The work in [5] details various software and hardware aspects of rISA designs. Some of them are listed here since they are also used by *adapt-rISA*: (a) in order for the normal instructions to adhere to the word boundary, there can be only an even number of contiguous rISA instructions. To achieve this, a rISA instruction that does not change the state of the processor is needed: the `rISA_nop` instruction. The compiler can then pad odd-sized sequences of rISA instructions with rISA nops; (b) in order to dynamically change the execution mode of a processor, there should be a mechanism in software to specify the change in execution mode. For most rISA processors, this is accomplished using explicit mode change instructions. An instruction in the normal instruction set that changes mode from normal to rISA mode is termed the `mx`

instruction, and an instruction in the rISA instruction set that changes mode from rISA to normal is the `rISA_mx` instruction. A piece of code including the mode change instructions was shown in Figure 1; (c) the fetched code is interpreted (decoded) as normal or rISA instruction depending on the operational mode of the processor. When the processor is in rISA mode, the fetched code is assumed to contain two rISA instructions. The first one is translated into a normal instruction, while the second one is latched and kept for the next cycle of execution. The translation can be performed in terms of simple and small table lookups. Since the conversion to normal instructions is done during or before the instruction decode stage, the rest of the processor remains the same - only the decode logic needs to be modified.

Another class of techniques for code size reduction is code compression with dynamic hardware-based decompression. These techniques typically propose a hardware block that dynamically decompresses the instruction stream as it arrives at the processor. This decompression block resides between memory (either the main memory or instruction cache) and the processor. [7] first proposed a Huffman-coding based code compression scheme for the MIPS architecture. [8,9] explored dictionary-based compression techniques. [10] considered an improvement to the standard dictionary-based compression based on vector Hamming distances. They report a code size reduction of 20% to 28% for the TI TMS320C6x processor. [11,12] proposed a bitmask-based code compression technique that significantly improves on the dictionary-based approach. With application-aware bitmask and dictionary selection methods they were able to achieve a code size reduction of upto 42% for the TI TMS320C6x processor. These dynamic code compression techniques are relatively independent from the rISA technique and can be combined with rISA to achieve greater code size reduction.

IV. Adaptive rISA

The central idea supporting the adaptive rISA concept is that a *divide and conquer rISA approach* can be employed to extract better results from a single embedded application in terms of code compression and also of power reduction. Previous works did not consider such granularity: they search for the optimal rISA design, and configuration, for an entire application (only one rISA design configuration per application). The software and hardware aspects behind the *adapt-rISA* solution are the same as those used in [5] and similar to that in ARM's Thumb ISA [2]. They are: (a) use of *rISA_nops* and change mode instructions; (b) conversion from reduced to normal instructions performed during the instruction decode stage; and (c) the actual execution of normal instructions only.

A. Motivation

Even an embedded application of low complexity probably includes distinct sections of code with different computational requirements (based on string manipulation, or on bitwise and logical operations, or on arithmetic operations, and others). In general, only a subset of $\boldsymbol{O}$ will be needed to reference all the different opcodes in a specific section of the application. As a consequence, the same cardinality of $\boldsymbol{\Gamma}$, specified by a given rISA design, can encompass more of the opcodes in that region. This is important not only to increase the number of reduced instructions in the final code, but also because lesser number of bits may be employed to specify the opcode in the reduced instructions. The additional bits may then be employed to specify immediate values, for example. This assertion and also the fact that ***rISA_4444*** presents a solution with a good trade-off between code compression and translation process complexity, when compared to other designs, led us to select such design in our experiments. The smaller number of opcodes, in previous works, was used by the design ***rISA_4444***: sixteen only.

It was stated earlier that not all the initially marked instructions, as specified by the ***rdc***, are actually reduced ($\boldsymbol{x'_i <= x_i}$). Some of these instructions are discarded from reduction. The main cause of discard is overflow. An instruction is discarded by overflow if some specific bit field in its normal form, such as the immediate value, cannot be expressed using the reduced number of bits in the corresponding reduced form. However, there are two other reasons for discarding. If the number of contiguous instructions, marked to be reduced, is too small, the potential reduced block to be formed will cause an expansion in the code, instead of a reduction (the later addition of change mode instructions will cause this expansion). These contiguous instructions must then be discarded from reduction. Branches and jumps between normal and reduced blocks are allowed only when the source is a normal instruction and the target is a change mode instruction at the beginning of a reduced block. In all other cases, the instructions acting as source and target must be discarded from reduction.

We decided to measure the discard of instructions, during compilation, related to the three causes presented above. These numbers were not presented by previous works. The results for the MiBench program *qsort*, reduced using the design ***rISA_4444***, are shown in Figure 3 (in the bars on the right in each pair). The first pair of bars show the total number of marked instructions, the last pair shows the number of instructions that were actually reduced, and the other pairs show the number of instructions discarded and corresponding causes. The values in Figure 3 show a lot of instructions discarded by the design ***rISA_4444***. The other MiBench programs, used in this work, presented similar values.

Furthermore, we remember that a smaller number of opcodes may be considered by rISA designs when handling different sections of an application. These two reasons led us to try a new rISA design with a set $\boldsymbol{\Gamma}$ of cardinality 8 ($\boldsymbol{n = 8}$). It was called ***rISA_8ops*** and used in the same *qsort* program of the MiBench. The discard results are also shown in Figure 3 (in the bars on the left). A significant improvement was reached: almost three times more instructions were actually reduced. Because of this, the new design was adopted. It has not been considered by previous works about rISA, whose $\boldsymbol{\Gamma}$ set cardinality varies from 16 to 128. In this experiment, the ***rISA_8ops*** used the eight most executed opcodes in the corresponding ***rISA_4444*** design configuration, for the whole application.

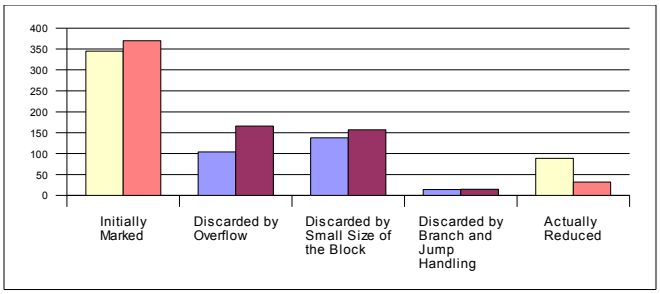

Figure 3. Discard of *instrs*., *qsort* program – *r_4444* (right) x *r_8ops* (left)

B. Code Conversion

The input to our *conversion-to-rISA* algorithm is the Assembly code generated by the *gcc cross-compiler*. This Assembly code is traversed and operated by a series of methods that, based on the rISA design and configuration previously specified, produce the final reduced code (Figure 4).

```
INPUT: application's Assembly code produced by gcc
PARAMETERS:rISA design and rISA design configuration
if (mips.usingRISA ( )) {
          mips.rISA.mapRegisters ( );
          mips.rISA.markCandidates ( );
          mips.rISA.isPossibleToReduceCandidates();
          mips.rISA.discardSmallBlocks ();
          while(mips.rISA.treatBranchesAndJumps())
               mips.rISA.discardSmallBlocks ();
          mips.rISA.countFinalBlocks ();
          mips.rISA.translateToRISAstep1 ( );
          mips.rISA.translateToRISAstep2 ( );
          mips.rISA.generateFinalCode ( output);
}
```

Figure 4. *conversion-to-rISA* algorithm

Since reduced instructions have limited register file accessibility, the method *mapRegisters* implements a simple register mapping strategy where all the instructions of the application are allowed to access only a fixed window of 16 contiguous registers of the register file (half of the total number available). It was not a limitation for the programs used in the experiments of this work, but it is a point to be improved for future work. The method *markCandidates* is called to mark all the instructions, in the application code, that constitute occurrences of the opcodes belonging to the ***Γ*** set of the rISA design configuration being used. The method *IsPossibleToReduceCandidates* discards (removing the mark) those marked instructions that cannot be reduced because the number of bits needed to specify their operands is greater than those available in the reduced instruction. Special reduced instructions are not used.

DiscardSmallBlocks is used to make an analysis on the blocks formed by adjacent marked instructions, taking into account the later insertion of the change mode instructions. The instructions inside blocks that cause an increase on the size of the final code are discarded from reduction. *TreatBranchesAndJumps* is called to discard those marked instructions involved, as origin or target, in branches or jumps from a reduced block to a normal block, or vice versa. The exception is a branch or jump from a normal block to the beginning of a reduced block. This method is called in conjunction with the method *DiscardSmallBlocks* until there are no more branches or jumps to be discarded. *CountFinalBlocks* is a method for statistical data generation.

The *translateToRISAstep1* method effectively creates the reduced blocks, inserting the change mode instructions (`mx` and `rISA_mx`) and also the reduced nops. *rISA_nops* are also used to position reduced branches and jumps in the least significant 16 bits of a word in the instruction memory. As a result, there is no need to offer a special handling to the latched reduced instruction when a branch or a jump occurs. The method *translateToRISAstep2* recalculates the offsets of branches and jumps, generates the final 16 bits sequences of the reduced instructions, and encapsulates the pairs of reduced instructions in word size boundaries. Finally, *generateFinalCode* is employed to generate the final code in the format used by our simulator.

C. Design Space Exploration

The DSE (design space exploration) process to obtain the best rISA design configuration for an application focus on the dynamic aspects of its code execution. It includes the following steps:

1. The application is executed with a small dataset to get its execution profile, and the instructions inside the most executed sections of code are marked. The different opcodes of these marked instructions are identified and stored. A small dataset means a subset of the applications' input data or a reduced number of iterations;
2. A DSE process is triggered using combinations of these opcodes (8 or 16 each time) to try improved results in terms of (i) total number of reduced instructions, (ii) average block size, and (iii) total number of blocks. The application is not actually executed;
3. The most promising combinations are used to form a rISA design and configuration database. Each record of this database is applied on the application using the *conversion-to-rISA* algorithm presented in Section IV-B. The application is then executed.

The adaptive rISA architectural feature arises when the granularity of this DSE process is changed from an entire application to its individual routines. The result is a set of rISA design configurations to be applied by the *conversion-to-rISA* algorithm in the reduction of the application's individual routines. Assembly directives (`risabegin` and `risaend`) are also provided to allow the coexistence of two or more rISA design configurations inside a single routine. The use of these directives is done by the designer manually at this moment.

D. Implementation Details

Some additional software and hardware aspects are needed to support the adaptive rISA architectural feature. First, the final reduced code must provide a way to inform the processor not only the execution mode, as in rISA, but also which rISA design configuration is being employed. The solution adopted is simple and direct: the `mx` instruction, used to change the

execution mode from normal to reduced, carries the rISA design configuration identifier as an immediate value.

The translation unit must receive, as an input, the rISA design configuration identifier provided by the `mx` instruction. It may also store the translation information partitioned into smaller and independent sub-units. These subunits are activated by the ***rdc*** identifier in the mx instruction. We remember that each reduced instruction has a corresponding instruction in the normal set. Then the size of these sub-units will be proportional to the number of opcodes employed by the rISA design - less opcodes require smaller sub-units. Thus, even having only simulated the translation unit, we argue that this partitioned characteristic may make *adapt-rISA* improve power, when compared to rISA, not only by reducing the number of fetch requests, but also during the transformation of reduced to normal instructions. More specifically, power gains may be obtained because (i) only one part of the total data structure, used for translation, is needed at each time; and (ii) less hardware is required to find the correct data for translation in a small data structure.

A framework [13] for design space exploration of embedded processors has been used in this work. We adopted the MIPS simulator, one of the processor models available in the framework.

V. Experiments and Results

The methods and tools described in the last section were used to experiment with the *bitcount*, *CRC32*, *qsort* and *stringsearch* programs of the MiBench benchmark. First, the experiments were executed using these programs individually and, afterwards, they were grouped, two by two, in six different and also more complex embedded applications, with different computational requirements. Our interest is to show that *adapt-rISA* can improve power and energy concerns in embedded processors, thus the main metric focused is the number of fetch requests to the instruction memory. A reduction in this metric directly corresponds to an energy reduction due to the reduced number of bus transactions. The reduction in overall energy and power is, however, not calculated. In the experiments, we present the following metrics: (1) normalized number of fetches – presented as a percentage of the number of fetches required by the application without reduction; (2) percentage of actual reduced instructions in the final code; (3) average size of the reduced blocks – considering only the instructions in the original code, i.e., change mode instructions and *rISA_nops* are not considered; (4) total number of reduced blocks; and (5) application's code size reduction – presented as a percentage of the normal code size. Since the number of fetch requests is the primary metric, the other four are defined based on the most executed instructions. Because of this, the application's code size reduction numbers, presented in the graphs, are only informative. It is worth to mention that (a) the code size reduction is higher in the presence of adapt-rISA[1] and (b) these numbers would be improved if we had chosen the most frequent instructions. In the next paragraphs, we will refer to these other metrics as code compression metrics.

Figures 5a and 5b present the results for the programs *bitcount* and *stringsearch*. The number of fetches are only slightly smaller in the presence of *adapt-rISA* and the average block size experienced a small reduction. But the number of reduced blocks and also the number of reduced instructions have a significant improvement: 21% and 24% on average, respectively. This compensates the reduction on the average size of the reduced blocks. In the case of the programs *CRC32* and *qsort* (not shown in the figure), the *adapt-rISA* approach produced the same results as the one optimal rISA design configuration found by the DSE process. These programs, even having a small number of reduced instructions, 39 and 89 respectively, achieve good levels of reduction related to the number of fetch requests: 24% and 28%, respectively. As illustration, the numbers of reduced instructions are 158 and 152 for *bitcount* and *stringsearch*, respectively. The term *adapt-rISA** in the figures identifies the use of the *adapt-rISA* Assembly directives introduced in the last section.

Figure 5c to 5h present the results for the six applications formed by the combination of the four MiBench programs, two by two. The experiments were arranged in a way that each program, in the whole application, executes during an equivalent number of clock cycles. For each of the four metrics, there are three numbers: one is for *adapt-rISA* and the other two are related to the (one) optimal ***rdc*** for each individual program: both were used in the respective combined application. The explanation for this choice is that the optimal ***rdcs*** for the combined applications have never shown better results than those of one of the individual ***rdc***s – what indicates the limitations of the *one optimal rISA per application* approach. There is one exception for this presentation of three numbers: it is for the pair *qsort+stringsearch*, whose individual ***rdc***s are the same. In general, *adapt-rISA* achieves better results, i.e., less fetches and better values in the code compression metrics. There were, in four of the six applications, less fetches to the instruction memory, from a minimum of 2% to a maximum of 7% of reduction. In the other two applications, there were the same or a by 1% increased number of fetches. However, in such cases, all other metrics were improved by *adapt-rISA*. The total number of reduced instructions was always larger in the presence of *adapt-rISA*: considering the ***rdcs*** with less fetches to the instruction memory (one of the two individual ***rdcs*** for each combined application), the average improvement was 19%. The number of reduced blocks experienced a reduction, using *adapt-rISA*, in one of the applications, but, in this case, there was a 30% improvement on the average size of the reduced blocks. In all other applications, the average size of the reduced blocks was improved by *adapt-rISA*.

These results were obtained using only the new design ***rISA_8ops***. It presented better results than those of the design ***rISA_4444*** for all applications used in this work. Each experiment was validated by comparing the result(s) produced by the program, when running on the host platform (x86 with Linux), with the corresponding result(s) produced by the simulator, available in the instruction memory of the processor model.

1 - The only exception is for the *stringsearch* program

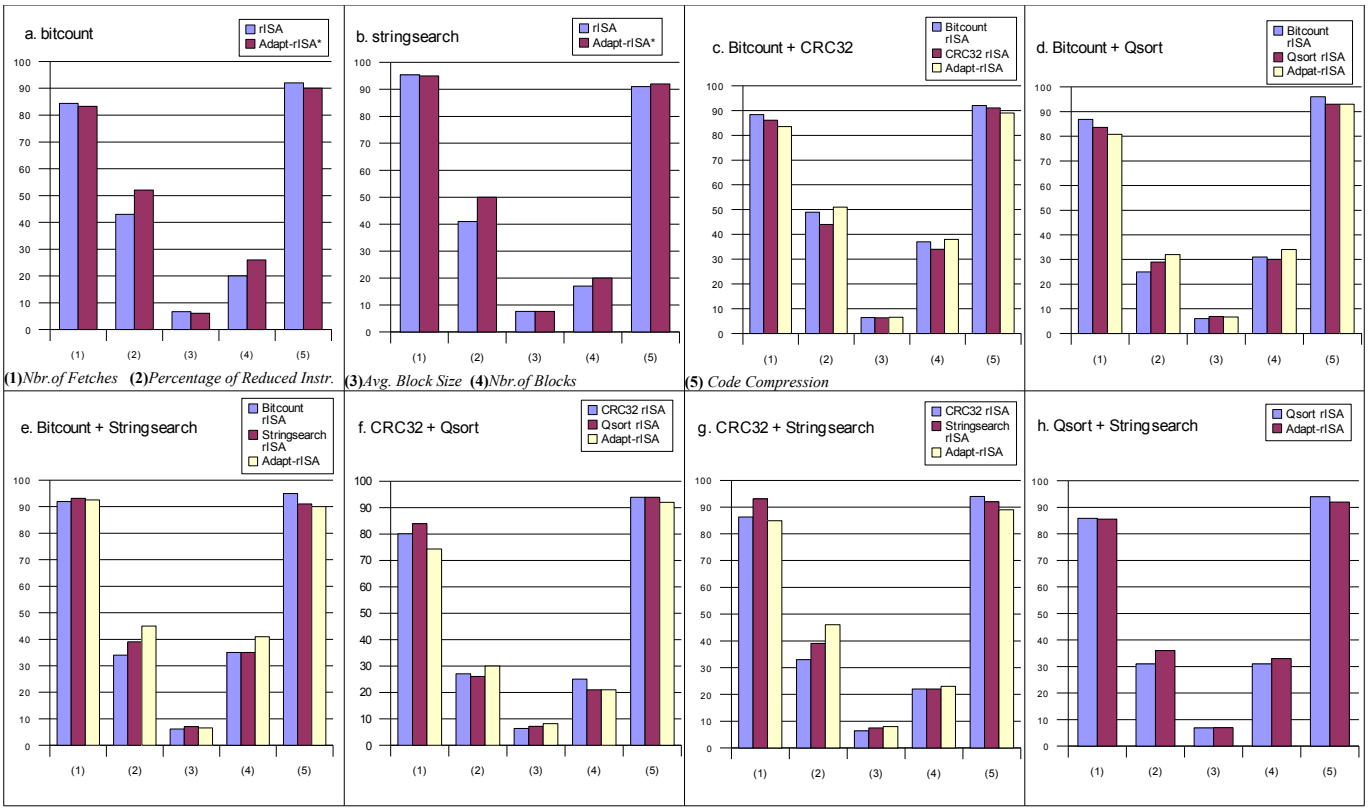

Figure 5. Application's results[2]

VI. Conclusion and Future Work

This work introduced the adaptive rISA architectural feature. It goes beyond rISA (as explored by previous works) because it is aware of the different computational requirements inside a single embedded application, what is important nowadays with the current trend of increasing software content on embedded systems by 10X per decade. It was showed that *adapt-rISA* presents better results than rISA in almost all the applications used in the experiments, and also for most of the metrics employed. For the code compression main metric, the average improvement was 19%, and, concerning the fetch requests, there were up to 7% less fetch requests. This work also described a new rISA design and discussed how its simplicity may be employed to reduce power consumption on the translation unit.

We enumerate the following future activities to address some limitations of this first work using the new *adapt-rISA* architectural feature: (1) the work focused mainly on DSE (design space exploration) for rISA design configuration. The path is opened for a DSE focused on different rISA designs; (2) the definition of a more robust heuristic to find the best rISA design and configuration, in the presence of the *adapt-rISA* Assembly directives; (3) the hardware implementation of the *adapt-rISA* translation unit and a precise estimation of the power reduction; and (4) evaluation using other embedded applications.

References

[1] *A. Shrivastava, P. Biswas, A. Halambi, N. Dutt and A. Nicolau. Energy Efficient Code Generation using rISA. Proceedings of the Asia and South Pacific Design Automation Conference – ASPDAC 2004.*

[2] *Richard Phelan. Improving ARM Code Density and Performance - New Thumb Extensions to the ARM Architecture. June 2003. Available at: www.arm.com*

[3] *M.R.Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge and R.B. Brown, MiBench: A Free, Commercially Representative Embedded Benchmark Suite, 4th Workshop on Workload Characterization, Dec. 2001.*

[4] *ARC Company, available at: www.arc.com*

[5] *A. Shrivastava, P. Biswas, A. Halambi, N. Dutt and A. Nicolau. "Compilation Framework for Code Size Reduction using Reduced Bit-width ISAs". ACM TODAES: ACM Transactions on Design Automation of Electronic Systems, 2006.*

[6] *A. Shrivastava, P. Biswas, A. Halambi, N. Dutt and A. Nicolau. An Efficient Compiler Technique for Code Size Reduction using Reduced Bit-width ISAs. Proceedings of the International Conference on Design Automation and Test in Europe – DATE 2002.*

[7] *A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. MICRO, 81-91, 1992.*

[8] *S. Liao, S. Devadas, and K. Keutzer. Code density optimization for embedded DSP processors using data compression techniques. Advanced Research in VLSI, 393-399, 1995.*

[9] *C. Lefurgy, P. Bird, I. Chen, and T. Mudge. Improving code density using compression techniques. MICRO, 194-203, 1997.*

[10] *M. Ros and P. Sutton. A hamming distance based VLIW/EPIC code compression technique. CASES, 2004.*

[11] *S. Seong and P. Mishra. A bitmask-based code compression technique for embedded systems. ICCAD, 2006.*

[12] *S. Seong and P. Mishra, An Efficient Code Compression Technique using Application-Aware Bitmask and Dictionary Selection Methods, Design Automation and Test in Europe (DATE), 2007.*

[13] *S.N. Soares and F.R. Wagner. Design Space Exploration using T&D-Bench. Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing, pp. 40 -- 47. (2004).*

2 Smaller numbers are better in metrics 1 and 5, larger are better in 2,3 and 4