Des Autom Embed Syst (2015) 19:329–344
DOI 10.1007/s10617-015-9159-8

A multiple-ISA reconfigurable architecture

Fernanda M. Capella · Marcelo Brandalero · Luigi Carro · Antonio C. S. Beck

Received: 6 March 2014 / Accepted: 9 January 2015 / Published online: 24 January 2015

Abstract In these days, every newly added hardware feature must not change the underlying instruction set architecture (ISA), in order to avoid adaptation or recompilation of existing code. Nevertheless, this need for compatibility imposes a great number of restrictions to the designers, because it keeps them tied to a specific ISA and all its legacy hardware issues. Considering that the market is mainly dominated by three different ISAs (and, very likely, more to come): ×86, used in the general purpose field; ARM, used in embedded systems, and PowerPC which covers a wide gamut of solutions, the need for another level (at the ISA) of adaptability is evident. Binary translation (BT) appears as a solution for that, since it is capable of transforming binary code so it can be executed on another target architecture. However, BT adds another layer between code and actual execution, therefore bringing huge performance penalties. To overcome this drawback, we propose a new mechanism based on a dynamic two-level binary translation system. The first level can translate from multiple architectures into an intermediate-level code, which will be optimized by the second level for execution on a dynamic reconfigurable array. In this way, the designer can take advantage of a BT system and program for multiple fields of application, without worrying about the underlying architecture. We present three case studies, along with a discussion as to how the first BT level is easily expandable to other ISAs.

Keywords Binary translation · Reconfigurable architecture · Code optimization · Transparent execution

F. M. Capella (✉) · M. Brandalero · L. Carro · A. C. S. Beck
Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil
e-mail: fmcapella@gmail.com; fmcapella@inf.ufrgs.br

M. Brandalero
e-mail: mbrandalero@inf.ufrgs.br

L. Carro
e-mail: carro@inf.ufrgs.br

A. C. S. Beck
e-mail: caco@inf.ufrgs.br

1 Introduction

Technological development has already started to stagnate as a result of the decline in Moore's law [1]. Therefore, one can observe that the processing capabilities of traditional architectures are not growing in the same pace as before [2]. In the same way, the great amount of already deployed software makes both designers and customers tied to a specific instruction set architecture (ISA). Nevertheless, companies develop their products focusing on the improvement of a given organization that will execute the same ISA as before, so that the large quantity of tools and applications already deployed can be reused. It is evident that this need for compatibility imposes a great number of restrictions to the design team and makes the process of software development segmented. While in the general purpose market the $\times 86$ dominates; embedded systems carry within different organizations of the ARM architecture; while PowerPC covers a wide range of applications from embedded to supercomputers. In this scenario, new alternatives are necessary to minimize this problem. From a software development perspective, the burden of passing the source code through the compilation process, which involves debugging, testing and code adaptation, must be alleviated, so one of the most important requirements of nowadays—time to market—can be kept as short as possible. Hardware designers, on the other hand, must have the freedom of implementing whatever architecture and organization they consider the best alternative, taking into account the field of application and non-functional requirements, such as performance and energy consumption.

Binary translation (BT) systems may be a solution, since they have the ability to transform an already compiled code into another to be executed on a different processor. Therefore, BT systems can give back to designers the freedom previously lost, since designers need not be tied to a specific ISA anymore. At the same time, neither software engineers nor users have to suffer with the inherent problems of code portability.

Considering the aforementioned discussion, the ideal scenario would be the execution of instructions compatible to any ISA on the very same underlying processor architecture. However, the maintenance of binary compatibility only is not enough to handle market needs. It is also necessary to translate and execute code in a competitive fashion, when compared to native execution [3], so the concept of binary translation must also be tightly connected to code optimization and acceleration [4].

In this way, this work extends the approach proposed by Fajardo [5], including support for the ARM and PowerPC ISAs as source code for a dynamic two-level binary translation system that, besides maintaining binary compatibility, amortizes its translation costs. An overview of the proposed system is presented in Fig. 1. The first BT level is responsible for translating the source code into an intermediate (common) code, as any conventional BT system would do, and the second BT level is responsible for transforming the translated code (intermediate representation) to be executed on the target architecture. The system we present on this paper supports three different ISAs ($\times 86$, ARM and PowerPC) as source architectures; the MIPS ISA is used as the common language; and a dynamic reconfigurable architecture is responsible for the acceleration of code after translation.

With the two-level BT mechanism, and having a clear interface between the translation and the optimization levels, another advantage emerges: during design time, by only changing the first BT level, it is possible to execute different ISAs transparently to the second BT level. Therefore, this greatly facilitates the porting of radically different ISAs without the need for changing the underlying architecture, as long as different first BT level layers are available. In the same way, it is possible to switch to another target architecture, according to the application needs or to the available architecture at the moment. None of the related

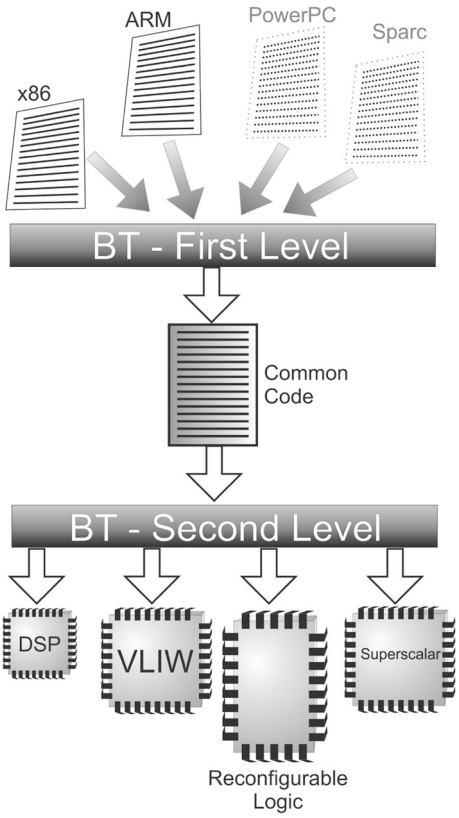

Fig. 1 Proposed approach

work, discussed in the next section, can easily switch between source/target architectures (when implemented in hardware), or use a special architecture (like a reconfigurable array) to accelerate code execution after translation.

The rest of this paper is organized as follows. In Sect. 2 we show some related binary translation architectures, with a brief explanation about their operation. In the next section, an overview on the proposed architecture is given. In Sect. 4 we present experimental results. Finally, the last section concludes this article.

2 Related work

Binary translation systems have been used mainly because companies need to reduce the time-to-market and maintain backward software compatibility. One example is Rosetta [6]: used into Apple systems to maintain compatibility between the PowerPC and $\times 86$. It is implemented in software and on the application layer with the sole purpose of maintaining binary compatibility, causing a great overhead. Another case is the FX!32 [7,8] that allows 32-bit programs to be installed and executed like an $\times 86$ architecture running Windows NT 4.0 on Alpha systems. The FX!32 is composed of an emulator and a binary translator system. As other examples, the HP Dynamo [9] analyzes the application at runtime in order to find the best parts of the software for the BT process, while the Daisy architecture uses BT at runtime to better exploit the ILP of a PowerPC application, transforming parts of code to

be executed on a VLIW micro architecture [10]. The Transmeta Crusoe processor [11] had the main purpose of translating ×86 code in software to execute onto a VLIW processor, reducing power consumption and saving energy. In this case, the BT is implemented in software, but the Crusoe hardware was designed to speed up the BT process with minimum energy, which decreases the translation overhead. The Godson3 processor [12] is another similar example: by using a software layer for binary translation (QEMU), it converts ×86 into MIPS instructions. As a strategy to optimize the running program, Godson3 is a scalable multi-core architecture, which uses a combination of a NoC (network on-chip) and a crossbar system for its communication infrastructure.

The approach proposed here is totally implemented in hardware, which allows for the fastest translation speed. Since there is a well-defined interface between the two BT levels, there is the possibility of source or target architectures migration at design time by only changing the corresponding level of the BT system. In this way, hardware modifications can be fine-tuned to several markets with different requirements. We aim to produce the effect of *write once, run everywhere* with minimal impact on performance. Therefore, comparing the proposed technique with other BT implementations, our main contributions are:

- Amortized performance overhead when translating from the source to the target machine by using dedicated hardware resources, thus freeing the CPU to execute the target code;
- Performance gains when compared to the execution of the original code in the source machine, since an optimization mechanism is used (in this case, a dynamically reconfigurable system);
- Design flexibility through the employment of the two-level BT system, making it easier to migrate code to another ISA or target architecture (or update them to a new version of the family). The system has therefore almost the same flexibility as if it were implemented in software.

3 Proposed architecture

Figure 2 gives a general overview of the proposed architecture. The first BT level interfaces the memory and the rest of system, which is composed of the second BT level mechanism, a translation cache, a MIPS processor to execute the common code when necessary and a dynamically reconfigurable array. The first BT level translates code from three different ISAs, namely ×86, ARM and PowerPC.

We use a reconfigurable array for the optimization process because they have already proven to accelerate software execution and reduce energy consumption [13–16]. Moreover, it is common sense that as the more the technology shrinks, the more an important characteristic of reconfigurable systems is highlighted: regularity. Besides being more predictable, regular circuits are also low cost, since the more customizable the circuit is, the more expensive it becomes. This way, regular fabric could solve the mask cost and many other issues such as printability, power integrity and other aspects of the near future technologies. The reconfigurable architecture that comprises the second level was extended from [17], and is responsible for optimizing the common code (MIPS) after the first level translation.

As an example of operation, let us consider an application compiled for the ×86 ISA, and that will be executed for the first time. The first BT level then fetches ×86 instructions from the memory and translates them into MIPS instructions. At this level there are no translations savings for future reuse: all the data is processed at run-time in order to avoid storage overhead. Then, while the MIPS processor executes the MIPS code, the second BT

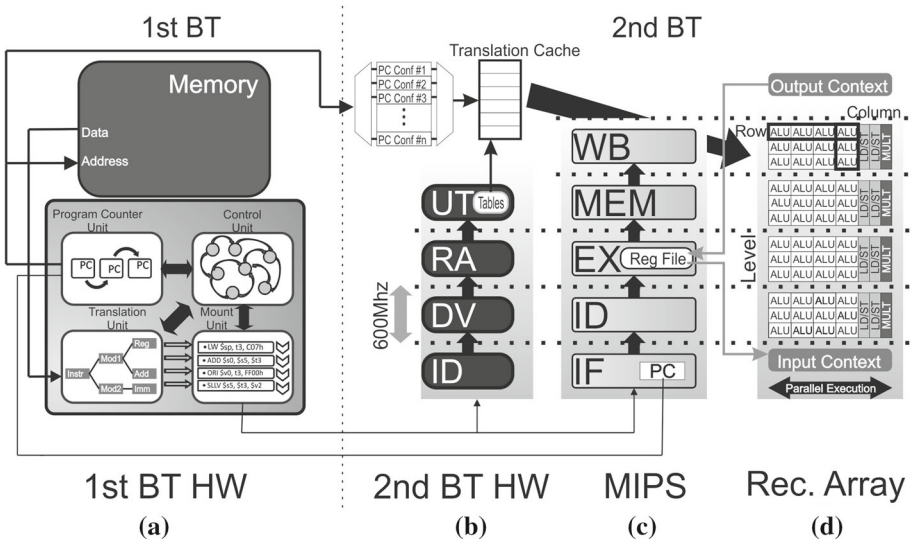

Fig. 2 **a** First BT level. **b** Four pipeline stages of the second BT level. **c** Five pipeline stages of the MIPS processor. **d** Reconfigurable array

level translates it into a configuration that will be executed in the reconfigurable array, and saves it in the translation cache (TCache) for future reuse. The second BT level only optimizes code sequences once it identifies that they are hotspots (most executed code parts).

Each entry in the TCache is indexed by the program counter (PC) of the source architecture. Therefore, next time a chunk of ×86 code that has already been transformed to MIPS and been optimized to the reconfigurable array is found, its optimized form (a configuration) is fetched from the TCache. When this happens, the first BT level, the MIPS processor and the second BT level are stalled and bypassed, and the reconfigurable array starts its reconfiguration and execution. As more and more sequences of instructions are executed and translated, and the TCache is being filled, the impact of the two levels of BT are amortized and the performance gains provided by the array start to appear.

3.1 First binary translation level—ARM and powerPC

In this ×86 implementation, 50 different integer instructions can be translated, out of a total of nearly 190 integer instructions, considering the ×86 ISA. All addressing modes are supported. The implemented subset is enough to compile and execute all the tested benchmarks. Segmentation is emulated, but there is no support for paging. Interruptions, and multimedia instructions, such as the MMX and SSE, are not implemented. For the ARM ISA, only the user mode with the ARMv4 instruction set is supported. Excepting the coprocessor instructions, all the others were implemented, including all addressing modes. These were also enough to execute all the tested benchmarks. Thumb mode and multimedia extensions are not supported. The ×86 translator was implemented as a separate unit from the ARM and PowerPC translators, as the CISC/RISC architecture distinction leads to different design options. We present here both implementations.

The First BT level, for the ARM and PowerPC architectures, is composed of two different hardware units, namely the decoding and the encoding units. We shall first present each of the module's role on the decoding process, then discuss its implementation (Fig. 3).

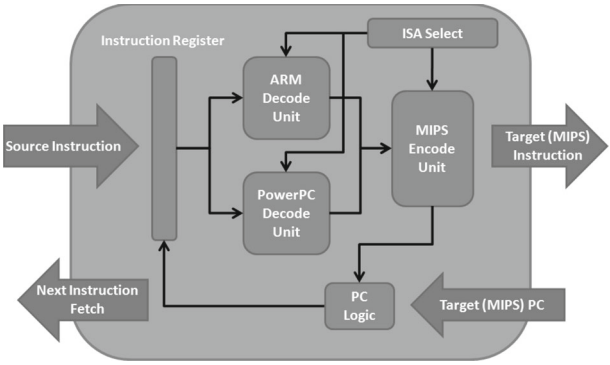

Fig. 3 Block diagram for the ARM and PowerPC first-level translators

3.1.1 Decoding unit

This unit is responsible for analyzing one instruction fetched from memory and extracting its semantics. The instruction semantics refers to the operation to be performed and where are its operands (register or memory addresses). Each of the extracted information is forwarded to the next unit.

This module comprises only combinatorial logic, which splits the fields of each instruction based on its format. This process is done in a parallel fashion: the instruction is used as input to multiple blocks, each of which splits the instruction fields assuming one particular format. Some fields can be merged, as many distinct operations may specify the same information on the same field (say, the destiny register).

Let us consider for instance the ARM architecture, and the structures required to decode data processing instructions. This class of instructions represents one possible encoding and covers basic arithmetic and logical operations on registers. All ARM instructions are fixed-length, then it is already known exactly which fields need to be analyzed in order to determine the instruction type. To decode an instruction such as "ADD r2, r1, r0", the following bits must be checked.

- Bits 27 = '0' and 26 = '0'. If this condition holds then it is a data processing instruction.
- Bit 25 = '0'. If this condition holds then the second operand is a register that may be shifted by an immediate value or by the value specified in another register.
- Bit 4 = '0'. If this holds, then the addressing mode for the operation is register shifted by an immediate value.

The conditions above are performed by the hardware block that identifies the instruction format. At the same time, the following contents from the instruction are extracted:

- The operation (ADD) is specified on bits 24 to 21, the destination register (r2) on bits 19 to 16 and the first operand register (r1) on bits 15 to 12.
- The second operand register (r0) is specified on bits 3 to 0, and the shift value is specified on bits 11 to 7.

This aforementioned procedure covers all data processing instructions that use this addressing mode in the ARM architecture.

3.1.2 Encoding unit

This unit receives the information extracted from the decoding unit, and uses it to generate a sequence of equivalent MIPS instructions. Since the decoding is done in a 1-to-many fashion (i.e. one instruction from the source architecture may generate more than one MIPS instruction), it is clear that the PC from the MIPS processor cannot be used to address the instruction memory. Therefore, this unit also has an internal mechanism that controls another PC that addresses the instruction memory. We shall make a distinction between these two PCs by writing PC_{BT} (the one that addresses the instruction memory) and PC_{MIPS} (the one inside the MIPS processor).

This unit is a finite state machine, whose inputs are all the outputs from the decoding unit, and which outputs one MIPS instruction every time the MIPS processor attempts to fetch a new instruction (by increasing the value of PC_{MIPS}). When the last MIPS instruction is about to be generated, a signal is sent to increment PC_{BT} in order to fetch the next instruction.

Since the translation circuit presents a critical path much shorter than that of the MIPS processor, clock speed is not an issue, and performance is limited by the number of instructions generated. However, the number of cycles required to fetch a new instruction and start executing the decoded ones on the MIPS processor must be taken into account, since this could lead to huge stall times. Another issue that was considered is the treatment of branch instructions. The translation scheme for this sort of instruction is tightly coupled to the source architecture being considered, as some architectures may allow for direct arithmetical operations on the PC, while others may allow register indirect branches. In cases of instructions that operate on the PC, it is important to note that the R15 (correspondent to the PC), as any other register, is also mapped to a regular MIPS register. However, it is not updated at every execution of a new instruction (otherwise the performance overhead would be huge). Therefore, when an instruction that operates on the R15 is detected, the current value of PC_{BT} is copied to the correspondent register in the MIPS, and the necessary operation is performed. However, the PC that needs to be updated is not the one inside the MIPS processor, but the PC_{BT}. Therefore, a mechanism is required to transfer the PC_{BT} value to the MIPS processor, and to operate on it if needed, and to capture back to PC_{BT}changes on the PC_{MIPS} value. The first transfer can be accomplished by encoding ADDI instructions with the value contained on PC_{BT}, while the second can be accomplished by encoding JR instructions and then capturing the updated PC_{MIPS} value and writing it to PC_{BT}.

3.2 First binary translation level—×86

The ×86 translator was implemented following a different approach, primarily due to its CISC and variable-length instruction scheme. The block diagram is the one shown on Fig. 2a and is described next.

3.2.1 Program counter unit

This unit is responsible for fetching instructions from the Instruction Memory;for the alignment of the incoming instructions and for calculating the address of the next instruction that must be fetched.

3.2.2 Translation unit

It is the main component of the system. It is responsible for analyzing the instruction format in order to classify them according to the type, operators, and addressing modes; and generating the equivalent MIPS instructions. This unit is constituted mainly of ROM memories that hold all possible equivalent MIPS instructions translations. For this reason, it concentrates the major part of the BT system area. Besides that, this unit provides some information to the other auxiliary units, such as: number of generated MIPS instructions, quantity of bytes to calculate the next PC and the type of instructions (e.g. logical operation, conditional or unconditional jumps, etc).

In the case of $\times 86$ instructions, it takes one or more cycles to perform such operations, has four different tables and a special hardware component to handle immediate values. These are directly related to fields found in an $\times 86$ instruction. The data from each field [prefix, OPCode, address mode and Scale Index Base (SIB)] are inputs for these four tables, while the offset and immediate values are merged together and sent to the immediate handler hardware. Depending on the instruction (e.g. an instruction with a register to register addressing mode), some of these tables may return null values.

Each one of these tables and the immediate hardware will be merged and sent to the Configuration Bits hardware component. Some bits of an ARM instruction are directly passed as Configuration Bits as for example bit P of a pre/post index transfer. At this point, the set of operations and its operands to perform the certain $\times 86$ or ARM instruction are known. They are in an intermediate form, composed of the configuration bits generated before. Then, with this information, they are mounted in the MIPS instruction format by the Code Generator/Assembler hardware. MIPS instructions will finally be generated.

3.2.3 Mounting unit

It provides an interface between the MIPS processor and the First BT mechanism. It fetches all the equivalent MIPS instructions in a parallel fashion from the Translation unit and sends them serially to the MIPS processor, making the BT mechanism behave as if it were a regular memory. It is composed of a queue of twenty four 32-bit registers in which each MIPS instruction is allocated. The maximum amount of MIPS instructions generated by an $\times 86$ instruction, considering the list of supported translations, is 12. In the case of ARM instructions, the average amount of MIPS instructions need to translate a single ARM instruction is about 2. However, there are some ARM instructions that will raise this number, such as LDM and STM, which are used for block transfers. The size of the queue must be twice of the maximum instruction count, because while the first half of the queue is receiving translations in the form of MIPS instructions from the translation unit, the second half is sending MIPS instructions to the MIPS processor. As instructions are processed, this unit constantly feeds the control unit with the number of occupied slots in the queue, in order to guarantee that it will not empty and the MIPS processor will not stall.

3.2.4 Control unit

It is implemented as a FSM machine. Its function is to keep the timing and consistency of information between the other units by using the information flags provided by each unit. Through this information, the control unit decides the behavior for all the system, such as the fetch of a new instruction from memory at the instant there are free slots in the queue in

the Mounting Unit; or the need for the calculus of a new source Program Counter when the instruction (branch or regular) is fetched from memory, besides its other functions, explained in the three subsections above.

3.3 Extended MIPS

The great advantage of using the MIPS ISA as common code is the regularity of code with well-known behavior, making it easy to translate from another ISA to this one. However, the translation of a complex ISA such as ×86 to MIPS is inefficient, since most of the times one single ×86 instruction is converted to many MIPS instructions. For instance, the support for flag registers on the ×86 ISA, which can be used for conditional branching, is inexistent on the MIPS architecture. In this case, more than 20 MIPS instructions could be generated per ×86 instructions to correctly emulate these flags support on the MIPS processor. Other constructs, such as segment addressing modes, also suffer from this lack of support.

As for the ARM ISA, although also a RISC architecture as is the MIPS ISA, the conditional execution support can overcharge the translations process. Extra test and branch instructions must be placed with a standard MIPS instruction in order to achieve the correct conditional execution behavior. We could observe, in the tested benchmark codes, a rate of 22 % of conditional instructions on average, which means that a considerable speedup could be achieved if some kind of support for these instructions were implemented in the MIPS processor. Another difference between ARM and MIPS is the use of a second complex operand, which can be shifted in several ways. About 39 % of total ARM instructions in the benchmarks tested make use of this kind of operand, which also characterizes a huge overhead when the translation occurs.

One can notice some PowerPC instructions make use of the carry flag, and some instructions must update some fields of the Fixed-Point Exception Register (XER) and/or the Condition Register fields. On average, 10 % of these instructions were found in the tested benchmarks, which would cause a substantial overhead on translated instructions.

In order to lower the overhead caused by the lack of some features on the MIPS ISA, the MIPS processor was extended to give hardware support to some of these issues (which involved simple modifications in the ALU and a few extra registers), but still maintaining compatibility with the standard code.

3.4 Reconfigurable array

An overview of its general organization is shown in Fig. 2d. The reconfigurable unit, based on the one presented in [17], is a dynamic coarse-grained array tightly coupled to the processor [18]. It works as an additional functional unit in the execution stage of the pipeline. This way, no external accesses (with respect to the processor) to the array are necessary. The reconfigurable array has already proven, in previous works, to be capable of accelerating applications with low ILP levels. In [19], a high level analysis on the applications (discussing their parallelism degree and how control/data flow they are) and potential of optimization that could be performed by reconfigurable architectures is done (pointing the advantages of using a coarse-grained reconfigurable array). In [17], it is shown that the reconfigurable array used here as case study is capable of accelerating applications with a high rate of control instructions (with small basic blocks that limit the amount of ILP available for exploitation) and low ILP because it takes advantage of "merging" sequential instructions. Moreover, in [20] it is demonstrated that this reconfigurable system presents a higher ILP than a 4-issue superscalar processor.

3.5 Second binary translation level

The second level of the binary translation hardware was extended from [17]. It starts working on the first instruction found after a branch execution, and stops the translation when it detects an unsupported instruction or another branch (in cases when the limit for speculative execution is reached). If more than three instructions were found, a new entry in the cache (FIFO-based) is created and the data of a special buffer, used to keep the temporary translation, is saved in the TCache. The proposed mechanism relies on a set of tables, used to hold the information on the sequence of processed instructions, i.e. the routing of the operands and the configuration for the functional units.

4 Results

4.1 Simulation enviroment

To perform all of the tests we have used a MIPS R3000 processor with a unified instruction/data cache memory with 32 Kb. The reconfigurable array has 48 rows and 16 columns. Each column has 8 ALUs, 6 LD/ST units and 2 multipliers. The Translation Cache is capable of holding 512 configurations. In previous works [17], this setup has already shown to be the best tradeoff considering area overhead and performance boosts. The Mibench benchmark set [21] was executed on a Linux based operating system environment. In all cases the applications were compiled and statically linked using GCC with –O3 optimization. To gather data on performance, ×86 and ARM execution traces were generated with the GEM5 simulator [22]. After that, cycle accurate simulators (described in SystemC at RTL level) were used for the BT mechanisms, reconfigurable architecture and the MIPS processor. A hardware prototype of the translator was implemented on a Xilinx Virtex5 FPGA and was used to gather the area usage. There results were then converted into logic gates count, based on the TSMC 90nm library, so we could use and extend and compare the results from [5].

4.2 Impact of the MIPS extensions

As explained before, the MIPS processor was modified to give additional support to the binary translation process. Figure 4 shows the mean number of MIPS instructions generated from an ×86 instruction when there is no support for the translation (original hardware); when

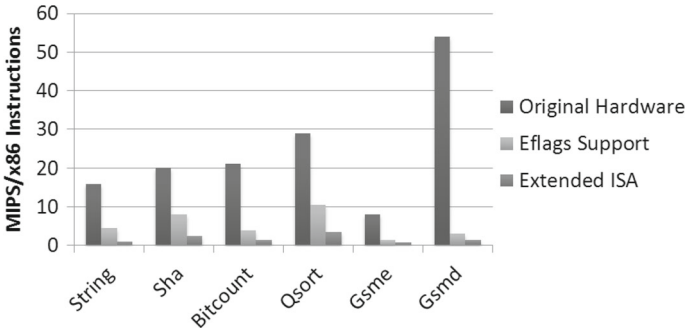

Fig. 4 The impact of using hardware support for ×86/MIPS translation

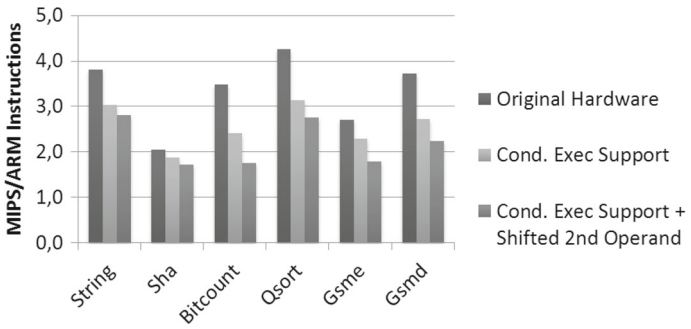

Fig. 5 The impact of using hardware support for ARM/MIPS translation

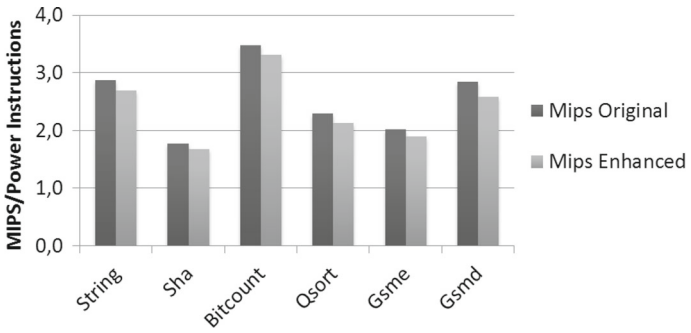

Fig. 6 The impact of using hardware support for PowerPC/MIPS translation

there is support to EFlags computation only (EFlags Support); and when other hardware modifications are also included (Extended ISA). It highlights the performance overhead that the second BT level system must overcome.

Considering the ARM ISA, with the exception of the conditional instructions and a few others (for example, those that use the post increment address mode), usually about two MIPS instructions are generated per ARM instruction. Figure 5 demonstrates the impact of implementing hardware support for MIPS translations, considering: ARM translation for native MIPS (Original Hardware); ARM translation for a MIPS with conditional execution support; ARM translation conditional executions and second shifted operand support. One can note a reduction of 28 % in the number of translated instructions when conditional execution is implemented in MIPS and an additional saving of 16 % with the support for the second shifted operand. On average, 44 % less instructions were generated with these simple hardware modifications.

Considering the PowerPC ISA, Fig. 6 shows the impact of MIPS modifications to support the binary translation. As already mentioned, modifications were made to support instructions that make use of Conditional Register and the Fixed-Point Exception Register, about 10 % of total instructions. On average, 8 % of reduction was noted with this modification.

4.3 Performance

Considering that the addition of the first level translation to the system does not affect the critical path of the MIPS processor, our metrics for performance is based on how many MIPS instructions are generated per source instruction. Figures 7, 8 and 9 demonstrate the

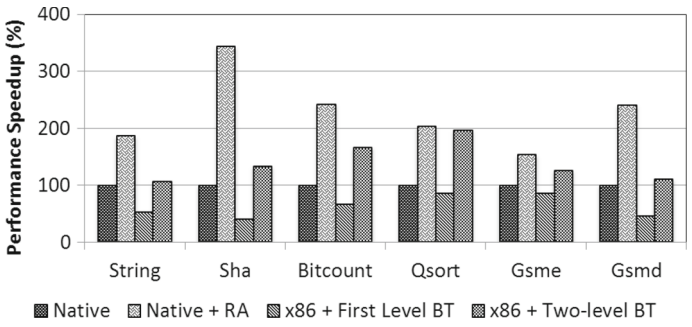

Fig. 7 ×86 speedup performance

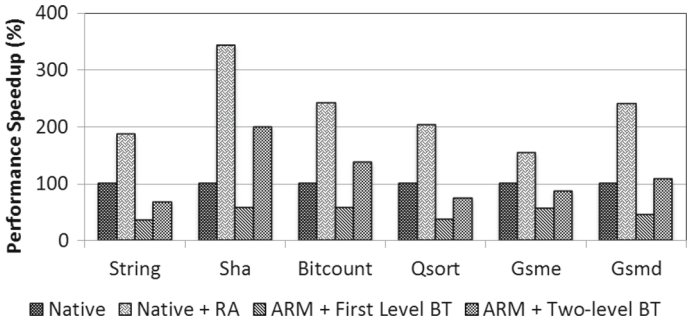

Fig. 8 ARM speedup performance

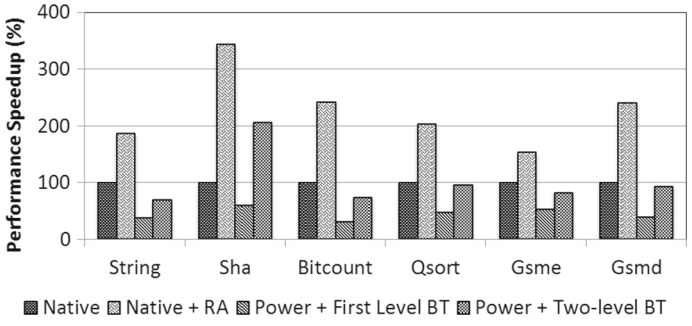

Fig. 9 PowerPC speedup performance

performance for × 86, ARM and PowerPC respectively. The first two bars are the same setup for the three architectures. First bar (*MIPS Code Execution*) represents native MIPS code execution on the standalone MIPS processor; and the second bar (*MIPS Code Execution* + *RA*) native MIPS code execution with reconfigurable acceleration. In this case, the first BT level is bypassed: only the second BT level plus the Reconfigurable Architecture (RA) are used.

The number of cycles taken to execute the native code on the standalone MIPS processor was normalized to 100 % (Native). The Native + RA achieve a speedup of more than two times on average. For example, SHA presents a speedup of 3.43 times, Bitcount has gains of

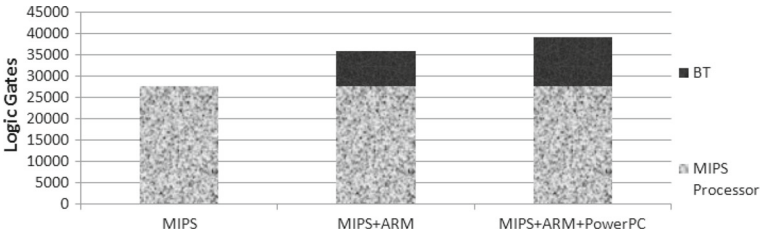

Fig. 10 Logic gates increase on the system

2.42 times, whereas the GSM Encoder presents a speedup of 1.53 times, which is the worst case considering the benchmark set.

Now, let focus on Fig.7, and consider the ×86 code being translated to MIPS code but not optimized by the reconfigurable system (×86+ First level BT). As it should be expected, there are performance losses because of the translation mechanism. For example, in the GSMD, a slowdown of more than two times is presented when compared to the native execution of the same algorithm in MIPS code. However, ×86 code execution on the proposed system (×86+ Two-level BT) is faster than the native MIPS code execution on the standalone MIPS processor, hence amortizing the original BT costs. The speedup over the standalone MIPS execution varies between 1.11 and 1.96 times. On average, the performance gains are of 45 %.

It is worthwhile to note that the optimization levels of the reconfigurable array with native code (source code directly compiled to MIPS) and with the post-translated ×86 to MIPS code are very similar. This can observed by comparing the first and second bars of each algorithm against the third and fourth ones, in Fig. 7. It means that the array is capable of optimizing code regardless if it was generated by a "smart" compiler (in the first case, GCC), or a simple compiler (which would be the equivalent of being generated by the first BT level). Similarly, Fig. 8 also demonstrates the performance speedup for the ARM processor. The speedup over the standalone MIPS execution varies between 0.66 % and 1.99 times. On average, the performance gains are of 13 %. In Fig. 9 one can note the performance speedup for the PowerPC processor over MIPS, which varies between 0.69 to 2.06, with an average gain of 1.03 times.

4.4 Area

We first present the results for the first BT level translator, comparing its area to that of a single MIPS processor. We also analyze the entire system, with the second level translator and the reconfigurable array.

Consider three scenarios, shown in Fig. 10: first the system comprising only the MIPS processor, then considering the addition of the first level ARM translator and finally with the addition of the PowerPC translator. Results show an increase of 29 % when adding the first new architecture (ARM), and further 12 % when adding the PowerPC architecture. This adds up to a 41 % area increase when supporting two new architectures. The ×86 translator, due to the inherent complexity of this architecture, was implemented as a separate module, and represents an area addition of 83 % with respect to the MIPS processor.

As for the area distribution inside the first level translator, shown on Fig. 11, we consider only the ARM and PowerPC ISAs implemented together, according to the implementation described on session III In this scenario, from a total of 11,510 logic gates required by the

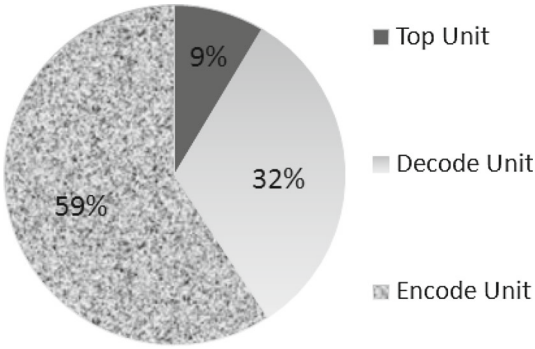

Fig. 11 Logic gates distribution inside the translation unit

Table 1 Area distribution of the whole system

Unit	Area (gates)	%
First-level BT (×86)	22,406	2.05
First-level BT (ARM & PowerPC)	11,510	1.05
MIPS R3000	26,866	2.46
Second-level BT	15,264	1.40
Rec. array	1,017,620	93.05

implementation, 59 % of these are used on the encoding block and 32 % on the decoding block. The decoding block itself comprises two separate blocks, one for decoding ARM instructions and one for PowerPC instructions. Both architectures use a similar amount of gates. As will be discussed next, the sub-module that scales the most, as new architectures are added to the system, is the decoding module.

When considering the whole system, we have the area distribution given in Table 1. The area of the translator represents less than 6 % of the entire system. Even though the Reconfigurable array is almost 40 times bigger than the MIPS processor (which is extremely simple), it can be produced using current manufacturing technologies. In comparison, according to Yeager [23], the total number of transistors of the old MIPS R10000 core is 2.4 million. Considering that one gate is equivalent to 4 transistors, which would be the amount necessary to implement a NAND or NOR gates, this processor would take nearly 600k gates to be implemented. In Sandy Bridge, 2.27 billion transistors are necessary to implement the whole system (including cache).

4.5 Scalability

We now proceed to an analysis of system scalability when considering multiple architectures. The goal was to find an intermediate representation for all instruction sets, which is obtained after decoding the instruction, and afterwards map this representation to MIPS instructions through the encoding unit. In this sense, the decoding unit must be redesigned for every newly added ISA, while the encoding unit is reused.

On our implementation, this decoding is performed by taking fixed-length instructions (4 bytes wide), extracting its operation and analyzing its fields. For both the ARM and PowerPC implementations, this module required from 200 to 250 LUTs. The main factors that affects the size of this unit are the number of different possible encodings and the number of different

operations supported by the source architecture. Two issues that increase the complexity of the decoding unit design are: source architecture with variable-length instructions (e.g. ×86 ISA) and the need to map a higher number of registers available in the source architecture to the target architecture (e.g. PowerPC ISA). For the ARM ISA, these issues do not show up.

As for the encoding unit, we designed it as a finite state-machine (FSM), for which each state corresponds to a MIPS instruction and the transitions allow for different sequences of instructions to be generated. Once all of the states are implemented, the number of registers used on this module no longer scales with the addition of new architectures. Each new ISA added to the system will impact only on the transitions between these states, slightly influencing on the number of gates required to implement the next state logic.

5 Conclusion

In this paper, we demonstrated a case-study of a totally flexible binary system, completely implemented in hardware, where both source and target architectures can be easily changed. In this case study, we have proved the effectiveness of our technique by showing the possibility of executing the large amount of available ARM, PowerPC and ×86 applications in a non-native architecture in a totally transparent fashion, where no kind of user intervention is necessary and no performance losses are presented. We also showed that it is possible to obtain binary compatibility with average gains in all proposed architectures. It is very likely that the implementation of other ISAs, which are more RISC like, would present even more performance gains, since they would be simpler to translate than the ×86, ARM or PowerPC ISAs. We intend to improve our system, by increasing the number of supported instructions and adding support for different ISAs. Also, we intend to implement different target architectures and the support for interrupts and traps. Moreover, we will investigate the effectiveness of using MIPS as intermediate language, and study whether there are better alternatives or not.

References

1. Kim NS, Austin T, Blaauw D, Mudge T, Flautner K, Hu JS, Irwin MJ, Kandemir M, Narayanan V (2003) Leakage current: Moore's law meets static power. Computer 36(12):68–75
2. Mak J, Mycroft A (2009) Limits of parallelism using dynamic data dependence graphs. WODA, Chicago
3. Sites RL, Chernoff A, Kirk MB, Marks MP, Robinson SG (1993) Binary translation. Commun ACM 36 2:69–81
4. Altman ER, Kaeli D, Sheffer Y (2000) Welcome to the opportunities of binary translation. Computer 33(3):40–45
5. Junior JF, Rutzig MB, Beck ACS, Carro L (2011) Towards an adaptable multiple-ISA reconfigurable processor. In: International workshop on applied reconfigurable computing, 2011, Belfast. Lecture notes in computer science: reconfigurable computing: architectures, tools and applications, vol 6578. Springer, New York, pp 157–168
6. Rosetta, Apple Inc., http://www.apple.com/rosetta/
7. Chernoff A, Herdeg M, Hookway R, Reeve C, Rubin N, Tye T, Yadavalli SB, Yates J. (1998) FX!32: a profile-directed binary translator. In: IEEE Micro, pp 56–64
8. Hookway RJ, Herdeg MA (1997) DIGITAL FX!32: combining emulation and binary translation. Digit Tech J 9(1):3–12
9. Bala V, Duesterwald E, Banerjia S (2000) Dynamo a transparent dynamic optimization system. In: PLDI'00, ACM Press, New York, pp 1–12
10. Daisy K, Ebcioglu EA (1996) DAISY: dynamic compilation for 100 % architectural compatibility. IBM T. J. Watson Research Center—Technical Report, Yorktown Heights

11. Dehnert JC, Grant BK, Banning JP, Johnson R, Kistler T, Klaiber A, Mattson J. (2003) The Transmeta Code Morphing^{TM} Software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In: Proceedings of the international Symposium on code generation and optimization: feedback-directed and runtime optimization, San Francisco, California, ACM International Conference Proceeding Series, vol. 37. IEEE Computer Society, Washington, DC, p 15–24
12. Hu W, Wang J, Gao X, Chen Y, Liu Q, Li G (2009) Godson-3: a scalable multicore RISC processor with x86 emulation. IEEE Micro 29(2):17–29
13. Beck Filho ACS, Carro L (2005) Dynamic reconfiguration with binary translation: breaking the ILP barrier with software compatibility. In: Proceedings of design automation conference, DAC, 42, Anaheim, ACM Press, New York, pp 732–737
14. Lysecky R, Stitt G, Vahid F (2006) Warp processors. In: ACM transactions on design automation of electronic systems (TODAES), pp 659–681
15. Beck ACS, Carro L (2010) Dynamic reconfigurable architectures and transparent optimization techniques: automatic acceleration of software execution, 1st edn. Springer, New York
16. Beck ACS, Lisboa CAL (2012) Adaptable embedded systems. Springer, New York
17. Beck AC, Rutzig MB, Gaydadjiev G, Carro L (2008) Transparent reconfigurable acceleration for heterogeneous embedded applications. In: Proceedings of DATE. ACM, New York, pp 1208–1213
18. Compton K, Hauck S (2002) Reconfigurable computing: a survey of systems and software. ACM Comput Surv 34(2):171–210
19. Beck ACS, Rutzig MB, Gaydadjiev G, Carro L (2008) Run-time adaptable architectures for heterogeneous behavior embedded systems. In: Proceedings of the 4th international workshop on reconfigurable computing, pp 111–124
20. Beck ACS, Carro L (2007) Transparent acceleration of data dependent instructions for general purpose processors. In: Proceedings of the IFIP VLSI-SoC 2007, IFIP WG 10.5 international conference on very large scale integration of system-on-chip, Atlanta, pp 15–17
21. Guthaus MR, Ringenberg JS, Ernst D, Austin TM, Mudge T, Brown RB (2001). MiBench : a free, commercially representative embedded benchmark suite. In : Proceedings of the workload characterization. IEEE international workshop. WWC. IEEE Computer Society, Washington, DC, pp 3–14
22. GEM5 (2012) http://www.gem5.org
23. Yeager KC (1996) The Mips R10000 superscalar microprocessor. IEEE Micro 16:28–40