A Transparent and Adaptable Multiple-ISA Embedded System

Jair Fajardo Junior, Mateus B. Rutzig, Luigi Carro, Antonio C. S. Beck
{jffajardoj, mbrutizg, carro, caco}@inf.ufrgs.br
Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, RS, Brazil

Abstract - *In these days, every new added hardware feature must not change the underlying instruction set architecture (ISA), in order to avoid adaptation or recompilation of existing code. Therefore, Binary Translation (BT) opens new possibilities for designers, previously tied to a specific ISA and all its legacy hardware issues, since it allows the execution of already compiled applications on different architectures. To overcome the BT inherent performance penalty, we propose a new mechanism based on a dynamic two-level binary translation system. While the first level is responsible for the BT de facto to an intermediate language, the second level optimizes the already translated instructions to be executed on the target architecture. The system is totally flexible, supporting the porting of radically different ISAs and the employment of different target architectures. This paper presents the first effort towards this direction: it translates code implemented in the x86 ISA to MIPS assembly (the intermediate language), which will be optimized by the target architecture: a dynamically reconfigurable architecture. We show that is possible to maintain binary compatibility with performance improvements when compared to native execution.*

Keywords: Binary Translation, Reconfigurable Systems

1 Introduction

With the constant growth of the embedded systems market, more complex applications have been developed to fulfill consumer needs. At the same time, technological development has already started to stagnate as a result of the decline in Moore's law [1], and one can observe that the processing capabilities of traditional architectures are not growing in the same pace as before [2]. In this scenario, new alternatives are necessary to minimize this problem. However, the support for binary compatibility, so that the large quantity of tools and applications already deployed can be reused, is an important requirement to introduce new processors into the market. With this in mind, companies develop their products focusing on the improvement of a given architecture that will execute the same Instruction Set Architecture (ISA) as before. Nevertheless, this need for compatibility imposes a great number of restrictions to the design team.

Binary translation systems can give back to designers the freedom previously lost, since they do not need to be tied to a specific ISA anymore. Therefore, the ideal scenario would be as shown in Figure 1: the execution of instructions compatible to any given ISA on the very same underlying architecture. However, the maintenance of binary compatibility only is not enough to handle market needs. It is also necessary to translate code execution in a competitive fashion, when compared to native execution [3]. This way, the concept of binary translation must also be tightly connected to code optimization and acceleration [4].

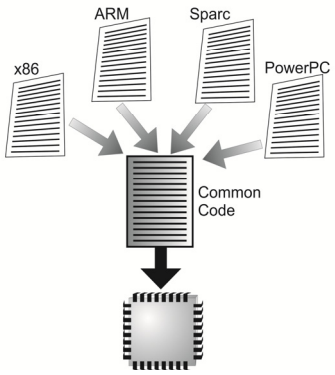

Fig. 1. Ideal scenario for current embedded systems.

With the aforementioned issues in mind, this work proposes a new approach based on a dynamic two-level binary translation system that, besides maintaining binary compatibility, amortizes its costs. An overview of the proposed system is presented in Figure 2. The first BT level is responsible for translating the source code to an intermediate (common) code, as any conventional BT machine would do. The second BT level is responsible for transforming the already translated code (intermediate code) to be executed on the target architecture. 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 in a completely transparent fashion to the second BT level, thus greatly facilitating 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.

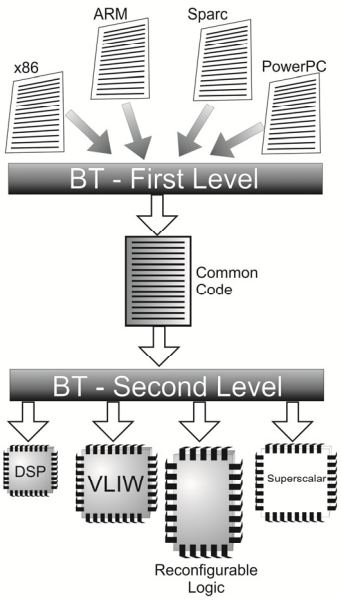

Fig. 2. Proposed Approach.

In the case study presented in this paper, which presents the first step towards this objective, x86 code is translated at the first BT level, MIPS assembly is used as the intermediate language, and a dynamically reconfigurable system [5] is used as the optimization machine, as shown in Figure 3a. This way, performance improvements are reached because both BT mechanisms are completely implemented in hardware for fast translation and minimum performance overhead; and once a sequence of code has passed through the two levels, the next time it is found both BT levels will be skipped (both translations will not be necessary, as illustrated in Figure 3b), and the reconfigurable array will be directly used, as long as there memory available. This is another advantage of the proposed technique and will be explained in more details later.

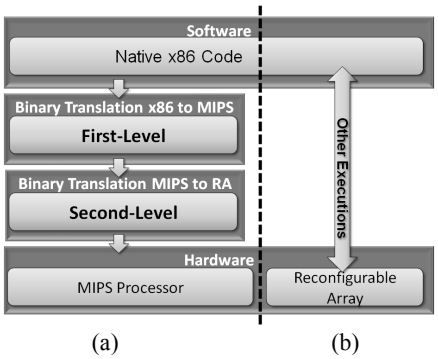

Fig. 3. Execution layers of the case study of the proposed approach.

The rest of this paper is organized as follows. In section 2, we show some related binary translation architectures, with a brief explanation about their operation. In the next section, an overview about the proposed architecture is given. In section 4 we present the experimental results and a discussion on these tests. In Section 5 we conclude this article and discuss future works.

2 Related Work

2.1 BT Systems

Binary translation systems have been used mainly because companies need to reduce the time-to-market and maintain backward software compatibility. They can work at different layers in a computing architecture: as if it was another regular application, visible to the user; or yet implemented in hardware, working below the operating system [7]. One example is Rosetta [8]: used into Apple systems to maintain compatibility between the PowerPC and x86. It works in the application layer with the sole purpose of maintaining binary compatibility, causing a great overhead. Another case is the FX!32 [9] [10] that allows 32-bit programs to be installed and executed like an x86 architecture running Windows NT 4.0 on Alpha systems. The FX!32 is composed of an emulator and a binary translator system. The emulator performs code conversion, and also provides profiling information at run-time. The binary translator uses the profiling information to generate optimized images through identified hotspots into the code, saving them for future reuse and avoiding excessive run time overhead. As other examples, the HP Dynamo [25] 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 [26].

The Transmeta Crusoe processor [11] had the main purpose of translating x86 code 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 (a VLIW processor) was designed to speed up the BT process with minimum energy, which decreases the translation overhead. The Godson3 processor [12] has the same goal as the Transmeta Crusoe: using a software layer for binary translation (QEMU), it converts the x86 to MIPS instructions. However, it uses a different strategy to optimize the running program. Godson3 is a scalable multicore architecture, which uses a mix between a NOC (network on-chip) and a crossbar system for its communication infrastructure. This way, up to 64 cores are supported. Each core is a modified superscalar MIPS to assist the dynamic translation. Therefore, Godson3 can achieve satisfactory execution time for applications already implemented and deployed in the field. However, the cost of having several superscalar processors is not small.

In this case study, instead of using Superscalar or VLIW processors, we use reconfigurable logic as the main optimization mechanism. Reconfigurable systems have already proven to accelerate software and reduce energy

consumption, showing gains over both systems [14][15]. Moreover, it is common sense that as the more the technology shrinks, the more an important characteristic of reconfigurable systems is highlighted: regularity – since this will impact the reliability of printing the geometries employed today in 65 nanometers and below [16]. 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.

2.2 Implementation

The binary translation process in software is more flexible due to the possibility to execute the BT system on other processors by recompiling the translator. However, it causes a huge overhead in execution time [13]. On the other hand, the implementation of the BT in hardware amortizes the translation overhead. In this case, the flexibility is strongly reduced: the hardware translation is hence tied to a specific ISA. Consequently, there is no opportunity to migrate to a new ISA or target architecture (or a new version of them) because the hardware was specifically tailored to that system. As a meet in the middle approach considering the two aforementioned methods, some BT systems present some kind of hardware modification to give better support for the software execution of the BT system. That is the case of Godson and Crusoe. Nevertheless, these works still rely on software for the main binary translation mechanism.

Our proposed approach is different because, besides being totally implemented in hardware, with the fastest translation speed, it uses a two-level BT mechanism: the first level is responsible for translating binary code from the source to the target processor, while the second is responsible for the code optimization. Since there is a well-defined interface between both, there is the possibility of easily ISA or target architecture migration at design time by only changing the correspondent level of the BT system. In this way, hardware modifications can be fine-tuned to several markets with different requirements. Therefore, comparing the proposed technique with other BT implementations, our main contributions are:

- Amortized performance overhead in the translation from the source to the target machine, because it is implemented in hardware, so it is faster than if it were implemented in software;
- 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);
- Flexibility through the employment of the two-level BT system, making it easier to migrate to another ISA or target architecture (or update them to a new version of the family), so the system has almost the same flexibility as if it were implemented in software. The next section details some implementation aspects of our system.

3 System Overview and Operation

Figure 4 gives a general overview of the system. The first BT level represents the hardware to make x86 to MIPS translations. It interfaces the memory and the rest of system, which is composed of the second-level BT mechanism, a special cache, a MIPS processor and a dynamically reconfigurable array. The BT in the second level analyzes the MIPS code at run-time and uses the reconfigurable logic to optimize and execute the hotspots found in the code. The system works like a native x86 processor, but with an additional possibility to run MIPS code too.

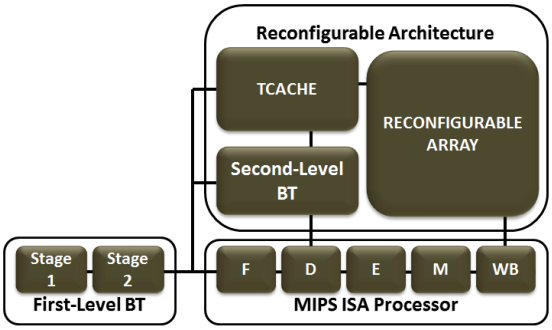

Fig. 4. Case Study Implementation.

3.1 Architecture Operation

Let us consider an application compiled using the x86 ISA that will be executed for the first time. Initially, the first level starts to fetch instructions from memory. As the first level is translating the code, the MIPS processor actually executes the instructions. In this level there are no translations savings for future reuse: all the data is processed at run-time by the first BT level in order to maintain small storage overhead. It also must be said that the same code, but in the optimized form, will be saved by the second-level for future reuse, as we shall see next. The second BT level analyzes the interpreted code (already in MIPS ISA) during execution. When a hotspot is identified, this level generates and saves a configuration of the reconfigurable array for that hotspot in a special cache (TCache), indexed by the x86 Program Counter (PC). The next time a chuck of x86 code that has already been transformed to MIPS and been optimized to the reconfigurable array is found, the equivalent configuration if fetched from the TCache. Then, the first BT level, MIPS processor and the second level mechanism are stalled, and the reconfigurable array starts its reconfiguration and execution.

Therefore, once a sequence of x86 instructions that were translated from the x86 to MIPS ISA was found and, after that, also became a configuration for the array, none of the BT mechanisms neither the processor need to work. As 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 starts to appear. In the next subsections the whole system is explained in more details.

3.2 First BT Level

In the current implementation, it is possible to translate 50 different instructions in a total of 150 considering the IA32 ISA, with all addressing modes supported. The implemented subset is enough to compile and execute all the benchmarks tested. Segmentation is emulated, but there is no support for paging. Interruptions, and other multimedia instructions, such as the MMX and SSE, are still not implemented. The First BT level is composed of four different hardware units, with two pipeline stages: Translation, Mounting, Program Counter and Control Units.

The main component of the system is the Translation Unit. It is responsible for fetching x86 instructions from the memory, analyzing their format in order to classify them according to the type, operators, and addressing mode and generating the equivalent MIPS instructions. It takes one or more cycles to perform such operations. This unit is constituted mainly of a ROM memory that holds 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).

The role of the Mounting Unit is to provide an interface between the processor and the BT mechanism, by fetching all the equivalent MIPS instructions in a parallel fashion from the Translation unit and sending them serially to the MIPS processor, making BT mechanism behave as if it were a regular memory. The Mounting Unit is composed of a queue of registers in which each MIPS instruction is allocated. As instructions are processed, this unit constantly sends to the other ones the number of occupied slots in its queue, in order to guarantee that it will not empty and the MIPS processor will not stall. The Program Counter Unit was developed to calculate the address of the next x86 instruction that must be fetched from memory. In opposite to the MIPS instructions, x86 instructions have different sizes, so the x86 and MIPS addresses translation cannot be considered on a one-to-one basis. Finally, the function of the Control Unit is to keep the timing and consistency of information between the other units by using the information flags found in 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 x86 PC when the instruction (branch or regular) is fetched from memory.

3.3 Extended MIPS

The great advantage of using the MIPS ISA is the regularity of code with well-known behavior, making it easy to translate another ISA to this one. However, the translation of a complex ISA such as the x86 to MIPS is inefficient, because in several times one x86 instruction is converted to many MIPS instructions. For example, in X86 instructions it is possible to use the memory contents as operators in arithmetic instructions. Furthermore, there are flag registers, which are automatically updated in most of the arithmetic/logical operations, so these can be used in branch instructions. Such flags are not supported in the MIPS architecture. In this case, more than 20 instructions would be necessary per x86 instructions to correctly emulate these flags on the MIPS processor. The same can be considered for segment addressing modes and so on for several other constructs. Therefore, to lower this overhead, the MIPS processor was extended to give hardware support to these issues, but still maintaining compatibility with the standard code, as follows:

- *Byte Manipulation* – Several operations that occur in x86 code are based on manipulation of variables with 8, 16 and 32-bit in a register. As the MIPS processor only executes 32-bit operations, a special hardware was added to manipulate small variables the same way as x86 processors do, avoiding the need of using mask operations to insert or extract information to/from 32-bit registers.
- *Address Mode* – The MIPS is a load-store architecture. In contrast to the X86 ISA, which supports several addressing modes, the MIPS supports only the Base (register) + Index (immediate) addressing mode. To reduce this gap, the MIPS was extended, so Base (register) + Index (Register) and Base + Index + Immediate (byte) operations are possible.
- *EFlags*– Additional hardware that generates the flag values and stores the results into a mapped register in the MIPS processor was implemented.

3.4 Reconfigurable Array

The reconfigurable unit [5] is a dynamic coarse-grain array tightly coupled to the processor [27]. It works as an additional functional unit in the execution stage of the pipeline, using similar approach as Chimaera [28]. This way, no external accesses (in respect to the processor) to the array are necessary. An overview of its general organization is shown in Figure 5. The array is two dimensional, and each instruction is allocated in an intersection between one row and one column. If two instructions do not have data dependences, they can be executed in parallel, in the same row. Each column is homogeneous, containing a determined number of ordinary functional units of a particular type, e.g. ALUs, shifters, multipliers etc. Depending on the delay of each functional unit, more than one operation can be executed within one processor equivalent cycle. It is the case of the simple arithmetic ones. On the other hand, more complex operations, such as multiplications, usually take longer to be finished. The delay is dependent of the technology and the way the functional unit was implemented. Load/store (LD/ST) units remain in a different group of the array. The number of parallel units in this group depends on the amount of ports available in the memory. The current version of the reconfigurable array does not support floating point operations. For the input operands, there is a set of buses

that receive the values from the registers. These buses will be connected to each functional unit, and a multiplexer is responsible for choosing the correct value (Figure 5a). As can be observed, there are two multiplexers that will make the selection of which operand will be issued to the functional unit. We call them input multiplexers. After the operation is completed, there is a multiplexer for each bus line that will choose which result will continue through that bus line. These are the output multiplexers (Figure 5b). As some of the values of the input context or old results generated by previous operations can be reused by other functional units, the first input of each output multiplexer always holds the previous result of the same bus line. Furthermore, note that in the example used in Figure 5, the first group supports up to two loads to be executed in parallel, while in the second group four simple logic/arithmetic operations are allowed.

3.5 Second BT Level

The second level of the binary translation hardware was extended from [5]. 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 (when the limit for speculative execution is reached). If more than three instructions were found, a new entry in the cache (based on FIFO) is created and the data of a special buffer, used to keep the temporary translation, is saved. This translation relies on a set of tables, used to keep the information about the sequence of instructions that is being processed, e.g. the routing of the operands as well as the configuration of the functional units. Other intermediate tables are also needed; however, they are used only in the detection phase. This information is not saved in the TCache since it is not needed during the reconfiguration phase.

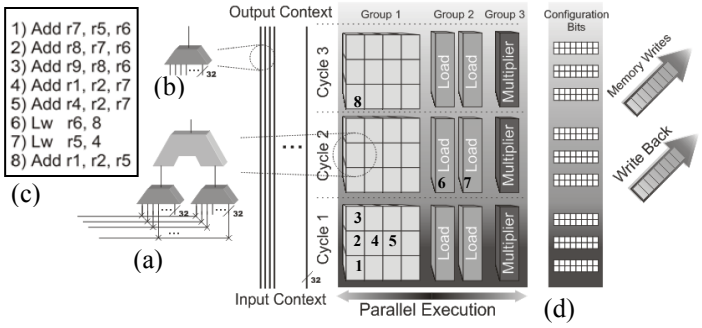

Fig. 5. The Reconfigurable Array

The BT algorithm takes advantage of the hierarchal structure of the reconfigurable array: for each incoming instruction, the first task is the verification of RAW (read after write) dependences. The source operands are compared to a bitmap of target registers of each row (which compose the dependence table). If the current line and all above do not have that target register equal to one of the source operands of the current instruction, it can be allocated in that line, in a column at the first available position from the left, depending on the group (using the resource table). When this instruction is allocated, the dependence table is updated in the correspondent line. Finally, the source/target operands from/to the context bus (input/output tables) are configured for that instruction. For each row there is also the information about what registers can be written back or saved to the memory (context table). Hence, it is possible to write results back in parallel to the execution of other operations. Figures 5c and 5d show an example of how a sequence of instructions would be allocated in array after detection and translation.

The algorithm supports functional units with different delays and functionalities. Moreover, it handles false data dependencies, and it also performs speculative execution. In this case, each operand that will be written back has an additional flag indicating its depth concerning speculation. When the branch relative to that basic block is taken, it triggers the writes of these correspondent operands. The speculative policy is based on bimodal branch predictor [6]. For each level of the tree of basic blocks, the counter must achieve the maximum or minimum value (indicating the way of the branch). When the counter reaches this value, the instructions corresponding to this basic block are added to that configuration of the array. The configuration is always indexed by the first PC address of the whole tree. If a miss speculation occurs a predefined number of times for a given sequence, achieving the opposite value of the respective counter, that entire configuration is flushed out and the process is repeated.

4 Results

4.1 Simulation Enviroment

To perform all of the tests we used a MIPS R3000 Processor with a unified instruction/data cache memory with 32 Kbytes. The reconfigurable array has 48 columns and 16 rows; each column has 8 ALUs, 6 LD/ST units and 2 multipliers. The Translation cache is capable of holding 512 configurations. In previous works [5], this setup has already shown to be the best tradeoff considering area overhead and performance boosts. The Mibench benchmark set [17] was executed on a Linux based OS. In all cases the applications were compiled and statically linked using GCC with -O3 optimization. X86 execution traces were generated by using the Simics instruction set simulator [18]. After that, cycle accurate simulators were used for the BT mechanisms, reconfigurable architecture and the MIPS processor. For the area evaluation, we used the Mentor Leonardo Spectrum [19] (TSMC 0.09u library) with VHDL versions of the MIPS [20], the reconfigurable architecture and the first BT level [5]. None of them increased the critical path of the MIPS processor, which runs at 600Mhz.

4.2 Binary Translation Data

In Figure 6, we analyze the memory occupation of the generated binary code. On average, considering the whole set

of benchmarks, the MIPS compiler generated a binary code 26.69% bigger than the same code for the x86 processor. Furthermore, as can be observed in Table 1, the MIPS processor executes, on average, 36,63% more instructions than the x86, considering the execution of the same algorithm.

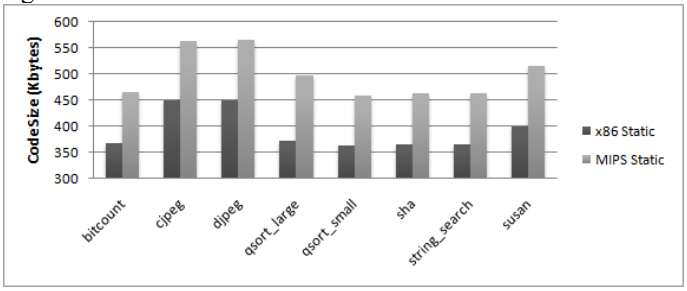

Fig. 6. Memory usage for different benchmarks, in Kbytes

Table 1. Number of executed instructions considering the two different ISAs

Benchmark	MIPS	x86
String Search	279,725	199,362
Sha	15,976,677	12,274,689
Bitcount	59,810,191	41,334,546
Qsort	51,695,224	27,386,935
Gsme	30,578,227	16,975,259
Gsmd	13,896,515	11,038,642

As explained before, the MIPS processor was modified to give additional support to the binary translation process. The Figure 7 shows the mean number of MIPS instructions generated from an x86 instruction when there is no support for the translation (Original Hardware), when there is support to EFlags computation only (EFlags Support) and when other hardware modifications, as explained in section III.C, are also included (Extended ISA). Figures 6 and 7, and Table I, show a clear difference in number of executed instructions between both architectures and how expensive the translation is. Both facts reflect in a performance overhead that the BT system must overcome.

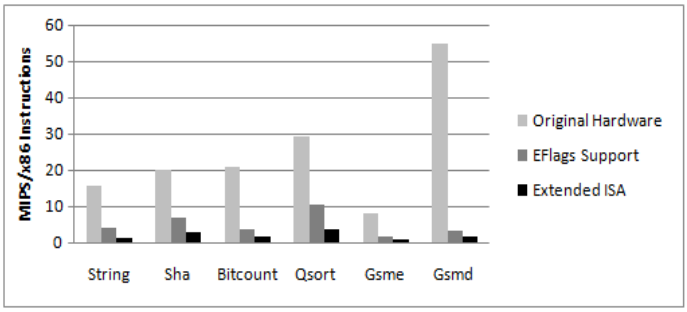

Fig. 7. The impact of using hardware support for the BT process.

4.3 Performance Evaluation

Figure 8 demonstrates the performance for four different setups:

- Native MIPS code execution on the standalone MIPS processor (MIPS Code Execution);
- 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 (MIPS Code Execution +RA);
- x86 code execution without reconfigurable optimization, so only the first BT level is used (X86 Code Execution without RA);
- x86 execution using the two BT mechanisms and the reconfigurable array (X86 Code Execution – Two Levels BT).

The native code execution on the standalone MIPS processor was normalized to 100%. The MIPS Code Execution + RA presents a speedup of more than two times on average. For example, Sha presents a speedup of 3.43 times, Bitcount has gains of 2.42 times, whereas the GSM Encoder presents a speedup of 1.53 times, which is the worst case considering the benchmark set. Similar speedups are found when using other reconfigurable architectures [21][22]. Now, let us consider the x86 code being translated to MIPS code but not optimized by the reconfigurable system. As it should be expected, there are performance losses because of the translation mechanism. In the GSMD, a slowdown of more than 2 times is presented when compared to the native execution of the same algorithm in MIPS code. However, X86 code execution on the proposed system 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%.

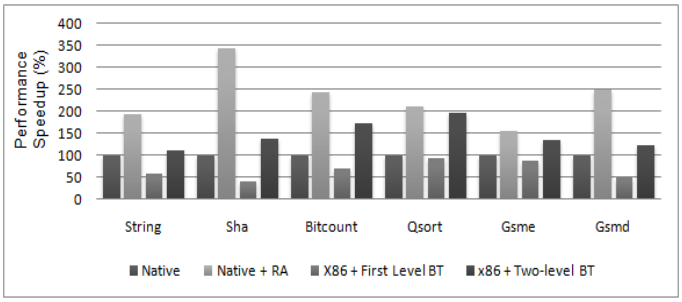

Fig. 8. Performance evaluation

These results can be considered very satisfactory: only the virtualization process (with no binary translation involved) of the Qemu virtual machine (VM) is 4 times slower than native execution of x86 instructions [22]. Because of such overhead, the Godson3, when translating code from X86 to MIPS using Qemu and without hardware support for the BT, is on average 6 times slower than native MIPS execution of the same software [12]. With hardware support for the BT mechanism, Godson3 performs on average 1.42 times slower than MIPS native execution. Although it is not our intent to directly compare our architecture to Godson3, since the Godson3 supports the whole X86 ISA, including interrupts and virtual memory, it gives one an idea that the translation is not an easy task to do, and that it presents significant overhead even if heavy hardware support is given.

4.4 Area Overhead

Table 2 demonstrates the number of gates that each hardware component takes. As can be observed, the fist BT level represents only 2% of the total system area. If we consider that each gate is composed of 4 transistors, the whole system would nearly take 4,87 million of transistors to be built. It is important to note that, if one compares the proposed system, which executes X86 instructions, to the standalone MIPS processor, there is a significant reduction in the instruction memory footprint, which amortizes the area overhead. According to our experiments, the MIPS compiler generated a binary code 26.69% bigger than the same code for the x86 processor, considering the whole set of benchmarks. Moreover, as already stated, the Godson-3 processor uses 4-superscalar MIPS R10000 cores. According to [24], each one of them takes nearly 2.4 million gates. Therefore, around 9.6 millions of transistors would be necessary to implement Godson, which is 2 times the size of our system.

Table 2. Area overhead of each component into the system.

Unit	Area (Gates)
First-Level BT	22,406
MIPS R3000	26,866
Second-Level BT	15,264
Rec. Array	1,017,620
Total	**1,219,535**

5 Conclusions

In this paper, we demonstrated the first step towards a totally flexible binary system, where both source and target architectures can be easily changed. In this case study, we proved the effectiveness of our technique by showing the possibility of executing the large amount of available x86 applications in a non x86 architecture in a totally transparent fashion, where no kind of user intervention is necessary and no performance losses are presented. We intend to improve our system, by increasing the number of supported X86 instructions and by adding support for different ISAs (e.g. ARM); and for different target architectures (VLIW, DSP and Superscalar processors). Moreover, we will also measure the power and energy consumption of the mechanism.

6 References

[1] Kim, N. S., Austin, T., Blaauw, D., Mudge, T., Flautner, K., Hu, J. S., Irwin, M. J., Kandemir, M., Narayanan, V.: Leakage current: Moore's law meets static power. Computer 36(12), 68–75 (2003)

[2] Mak, J., Mycroft, A.: Limits of parallelism using dynamic data dependence graphs. WODA, Chicago, Illinois, USA (2009)

[3] Sites, R. L., Chernoff, A., Kirk, M. B., Marks, M. P., and Robinson, S. G. 1993. Binary translation. Commun. ACM 36, 2, 69-81 (1993)

[4] Altman, E. R.; Kaeli, D.; Sheffer, Y.: "Welcome to the opportunities of binary translation," Computer , vol.33, no.3, pp.40-45, (2000)

[5] Beck, A. C., Rutzig, M. B., Gaydjiev, G., Carro, L. Transparent reconfigurable acceleration for heterogeneous embedded applications. In Proceedings of *DATE* . ACM, New York, NY, USA, 1208-1213 (2008)

[6] Smith, J. E. "A study of branch prediction strategies". In Proceedings of the 8th annual symposium on Computer Architecture, p.135-148, May 12-14, 1981

[7] Altman, E. R., Ebcioglu, K., Gschwind, M., Sathaye, S.: Advances and Future Challenges in Binary Translation and Optimization, In: Proceedings of the IEEE Special Issue on Microprocessor Architecture and Compiler Technology (2001)

[8] Rosetta, Apple Inc., http://www.apple.com/rosetta/

[9] Chernoff, A., Herdeg, M., Hookway, R., Reeve, C., Rubin, N., Tye, T., Yadavalli, S. B., and Yates, J.: FX!32: A Profile-Directed Binary Translator. In: IEEE Micro, 56-64, (1998)

[10] Hookway, R. J., Herdeg, M. A.: DIGITAL FX!32: combining emulation and binary translation. In: Digital Tech. J. 9, 1, 3-12 (1997)

[11] Dehnert, J. C. et al. The Transmeta Code Morphing™ Software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In: International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, vol. 37. IEEE Computer Society, Washington, DC, 15-24 (2003)

[12] Hu, W., Wang, J., Gao, X., Chen, Y., Liu, Q., and Li, G.: Godson-3: A Scalable Multicore RISC Processor with x86 Emulation. In: IEEE Micro 29, 2, 17-29 (2009)

[13] Gschwind, M., Altman, E., Sathaye, P., Ledak, Appenzeller, D.: Dynamic and Transparent Binary Translation. In: IEEE Computer, vol. 3 n. 33, 54-59 (2000)

[14] Beck Filho, A. C. S.; Carro, L.: Dynamic Reconfiguration with Binary Translation: Breaking the ILP barrier with Software Compatibility. In: Design Automation Conference, DAC, 42, Anaheim. Proceedings… New York: ACM Press, p. 732-737 (2005)

[15] Lysecky, R., Stitt, G., Vahid, F., "Warp Processors". In ACM Transactions on Design Automation of Electronic Systems (TODAES), pp. 659-681, July 2006

[16] Or-Bach, Z. "Panel: (when) will FPGAs kill ASICs?". In 38th DAC, (2001)

[17] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B. 2001. "MiBench: A free, commercially representative embedded benchmark suite." in Proceedings of the Workload Characterization. IEEE international Workshop. WWC. IEEE Computer Society, Washington, DC, 3-14 (2001)

[18] Magnusson, P. S., Christensson, M., Eskilson, et al: Simics: A Full System Simulation Platform, Computer, vol. 35, no. 2, pp. 50-58, (2002)

[19] Leonardo Spectrum, http://www.mentor.com

[20] Minimips VHDL, http://www.opencores.org

[21] Goldstein, S. C., Schmit, H., Budiu, M., Cadambi, S., Moe, M., Taylor, R. R.: Piperench: A reconfigurable architecture and compiler, Computer vol. 33, n.4, pp. 70–77, (2000)

[22] Clark, N., Kudlur, M., Park, H., Mahlke, S., Flautner, K.: Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization, In: MICRO-37, pp. 30-40, (2004)

[23] Bellard, F. "QEMU, a Fast and Portable Dynamic Translator", USENIX 2005 Annual Technical Conference, FREENIX Track (2005)

[24] Yeager, K.C.: The Mips R10000 superscalar microprocessor, In: Micro, IEEE , vol.16, no.2, pp.28-41, (1996)

[25] Bala, V., Duesterwald, E. , Banerjia, S. "Dynamo: A Transparent Dynamic Optimization System". In PLDI'00, pp. 1–12, ACM Press, 2000.

[26] Daisy K. Ebcioglu, E. A.. "DAISY: Dynamic compilation for 100% architectural compatibility". IBM T. J. Watson Research Center - Technical Report, Yorktown Heights, NY, 1996.

[27] Compton K., Hauck, S., "Reconfigurable computing: A survey of systems and software". In ACM Computing Surveys, vol. 34, no. 2, pp. 171-210, June 2002.

[28] Hauck, S., Fry, T., Hosler, M., Kao, J., "The Chimaera reconfigurable functional unit". In Proc. IEEE Symp. FPGAs for Custom Computing Machines, Napa Valley, CA, pp. 87–96, 1997.