Leveraging On-Chip Debug Features for Condition Codes Handling in Dynamic Binary Translation

Filipe Salgado*, José Mendes*, Adriano Tavares* and Mongkol Ekpanyapong†
*Centro ALGORITMI, University of Minho, Portugal
†Asian Institute of Technology, Thailand

***Abstract*—A widely recognized performance issue in DBT is the Condition Codes (CC) or flag bits handling. The currently proposed techniques to ease the handling of CC consist in the lazy evaluation of the condition codes, target ISA extensions or additional hardware support. In this paper we present a DBT engine developed for embedded systems and a novel technique to handle Condition Codes using architectural debug features. To prove the reduction on the translation overhead and generated code size we benchmarked our system. The results show signicant reduction of the generated code size and translation overhead reduction. The biggest impact of the Super Lazy evaluation technique was in the total code size, a reduction of 7.99%. The translation overhead reduced 5.65% over the common approach.**

***Index Terms*—Dynamic Binary Translation, lazy evaluation, condition codes, debug features**

I. Introduction and Related Work

The way Dynamic Binary Translation (DBT) works is creating an intermediary layer between the source code and the execution platform during run-time (dynamically) that translates the alien code for native execution. Although the process starts with the source code decoding, like emulators and simulators do, on DBT and opposed to the former, the source code is translated into target Instruction Set Architecture (ISA) code, stored into memory (a translation cache, Tcache), and hereafter fetched and executed from there.

DBT is an attractive technique for embedded systems because of its potential to provide features like code tracing, memory usage monitoring and identifying software inefficiencies in such systems [1], [2] .

A widely recognized performance issue in DBT is the Condition Codes (CC) or flag bits handling. The currently proposed techniques to ease the handling of CC consist in the lazy evaluation of the condition codes, target ISA extensions or additional hardware support [3], [4]. However, when the deployment technology does not include configurable logic (e.g., FPGA), architectural modifications are not an option. In this paper we present a DBT engine developed for embedded systems and resourceably and retargetably engineered; specifically we present a novel technique to handle Condition Codes using architectural debug features.

II. DBT Engine Description

We've developed a DBT engine, depicted on Figure 1, to explore the possibilities of binary translation, study its integration on System on Chip (SoC) and how to reduce its overhead through hardware support.

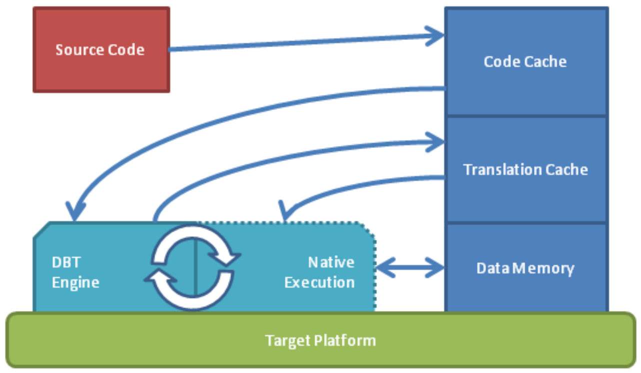

Fig. 1. DBT Engine Architectural Model

The DBT engine, running on top of the target processor, accesses the *Ccache* and translates the code which is then stored in the *Tcache*, which becomes ready for native *execution*.

III. Problem Definition

As the literature points out [3], [5], [6], Condition Codes is one of the hardest features to translate and emulate. We've paired the 8051 and the ARM Cortex-M3 architectures as source-target pair in our DBT engine. Apart from the expected difficulties in handling the CC we've found additional challenges, which are here explained.

A. Condition Codes Handling

In the 8051 architecture the condition codes are kept in the Program Status Word register (PSW), which is memory mapped in the address 0xD0. Once Cortex-M3 doesn't have the same condition codes, these must be emulated.

The CC status is computed by software every time an instruction affects, or is capable of affecting, the conditions. This is source of great overhead.

To minimize this problem a technique known as *lazy condition code evaluation* is employed in several DBTs and related [7], [8]. The technique consists in keeping a record of the last instruction that affected the CCs and its operands. The record might be overwritten several times, but the CC are calculated once and only before an instruction that requires them. Despite the good *execution* overhead reduction there is still *translation* overhead caused by the generation of code for (1) updating the lazy CC evaluation record and (2) calling a function to compute the CCs.

Another problem relies on ensuring memory coherency when condition codes are memory mapped. This is a double issue, one, because the affectation of the condition codes have impact on memory content and two, because of the reciprocity of the situation, any modification in the memory address of the condition codes will affect the state of these latter.

We propose a novel solution, by mixing the lazy evaluation approach with the hardware debug features present on the core and created an event-triggered lazy CC evaluation that we've called "super lazy CC evaluation"'. This approach incurs in even less *translation* overhead and generates less code because instructions depending on the CC do not include code generation. It also ensures memory coherency, because if the CC's memory address is read an updating event will be trigger, thus ensuring the coherency. The authors of this paper believe that this approach is pioneer in this area.

B. Lazy Condition Codes Evaluation Integration with Debug Features

One component of ARM's Debug Architecture is the Debug Watchpoint and Trace (DWT) block, which can be programmed to watch a certain data address and trigger a debugging exception on a match. We've programmed it to watch the accesses to the PSW in the memory and trigger a watchpoint debug exception. The from the debug exception handler we call the lazy evaluation routine. Bellow there is a code snippet with the configuration routine of the DWT block.

```
void DEBUGMONInit(int * Mem_Address)
{
  // Configure the Debug Exception and Monitor Control
     for watchpoint generated interrupts
  CoreDebug->DEMCR =  (1 << CoreDebug_DEMCR_TRCENA_Pos) |
                      (1 << CoreDebug_DEMCR_MON_EN_Pos) ;

  //Setting the Address to be watched
  DWT->COMP0 = *(int*)(void*)&Mem_Address;
  DWT->MASK0 = 0;
  DWT->FUNCTION0 =  0x5;
}
```

Whenever the PSW is read the debug monitor exception handler executes and calls the CC update routine.

IV. Experimental Results

A. Application Scenario

To prove the reduction on the translation overhead and generated code size we've run the "16-bit 2-dim Matrix" benchmark application from [9]. Despite not computation intensive there are several control flow instructions, which are CC affected, thus will require CC updating. We run the benchmark first with the common approach, and then using the Super Lazy Evaluation technique. We've gathered Tcache usage info and isolated the *translation* time from the total execution time to evaluate the translation overhead. The table I shows the comparative results.The biggest impact of the Super Lazy evaluation technique was in the total code size, a reduction of 7.99%. The translation overhead reduced 5.65% over the common approach. Due to incompleteness of system development, the presented benchmark is the only one currently runnable from the suite we will use to validate the whole system. Our preliminary experiences, however, lead us to be fairly confident these results will hold across different workloads.

TABLE I
Experimental Comparative Results

Metric	Lazy CC Evaluation (conventional method)	Super Lazy CC Evaluation	Reduction
Generated Code Size	1	0.92	7.99%
Translation Overhead	1	0.94	5.65%

V. Conclusion and Future Work

In this paper we've presented a novel approach to handle Condition Codes in Dynamic Binary Translation. The proposed Super Lazy CC Evaluation benefits are (1) generated code size reduction and (2) reduced translation overhead. Thus this technique proves itself of great impact on generated code size reduction and performance impact. The integration of architectural debug features in DBT systems should not be limited to condition codes handling. This technique was deployed on a specic target architecture (ARM Cortex-M3), however due to the wide availability of embedded processors with on-chip hardware debug features the technique is passive to be migrated for other architectures.

Acknowledgment

Filipe Salgado is supported by FCT, Fundação para a Ciência e Tecnologia (grant SFRH/BD/81681/2011). This work has been supported by the PEst-OE/EEI/UI0319/2014.

References

[1] A. Guha, "Memory Optimization of Dynamic Binary Translators for Embedded Platforms," Ph.D. dissertation, University of Virginia In, 2010.

[2] J. Paredes, "Dynamic binary translation for embedded systems with scratchpad memory," Ph.D. dissertation, University of Pittsburgh, 2011.

[3] H. Guan, H. Yang, Z. Qi, Y. Yang, and B. Liu, "The Optimizations in Dynamic Binary Translation," *2010 Proceedings of the 5th International Conference on Ubiquitous Information Technologies and Applications*, pp. 1–6, Dec. 2010.

[4] Y. Yao, Z. Lu, Q. Shi, and W. Chen, "FPGA based hardware-software co-designed dynamic binary translation system," *Field Programmable Logic and . . .* , pp. 0–3, 2013.

[5] D. Ung and C. Cifuentes, "Dynamic binary translation using run-time feedbacks," *Science of Computer Programming*, vol. 60, no. 2, pp. 189–204, Apr. 2006.

[6] S. Bansal and A. Aiken, "Binary Translation Using Peephole Superoptimizers." *OSDI*, pp. 177–192, 2008.

[7] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator." *USENIX Annual Technical Conference, FREENIX . . .* , pp. 41–46, 2005.

[8] R. Hookway and M. Herdeg, "Digital FX! 32: Combining emulation and binary translation," *Digital Technical Journal*, vol. 9, no. 1, pp. 3–12, 1997.

[9] G. Morton and K. Venkat, "Msp430 competitive benchmarking," Texas Instruments, Tech. Rep. July, 2005.