A Hardware-assisted Translation Cache for Dynamic Binary Translation in Embedded Systems

Filipe Salgado, Tiago Gomes, Adriano Tavares, and Jorge Cabral
Centro ALGORITMI, University of Minho (PORTUGAL)
{fsalgado, mr.gomes, atavares, jcabral}@dei.uminho.pt

***Abstract*—Approaches to Dynamic Binary Translation (DBT) on resource-constrained embedded systems are not straight forward, leading to several improvements and acceleration suggestions that rely on dedicated hardware. Software to hardware offloading is a common acceleration procedure used when software-only approaches do not meet the performance requirements, making such approach suitable to be successfully applied to DBT. This article approaches hardware offloading to address some limitations of an in-house DBT engine, the DBTOR, regarding its Translation Cache (TCache) management mechanism. The suggested approaches are non-intrusive to the target architecture, which cope with the commercial-off-the-shelf (COTS)-driven deployment of DBT for the resource-constrained embedded devices. This work proposes a TCache management hardware module that overpasses the linked list and hash table software-only approaches, resulting in a performance improvement of 25% and 26%, respectively.**

***Index Terms*—Dynamic Binary Translation, Embedded systems, Computer architectures, Hardware Accelerator, Constrained Devices, System-on-Chip.**

I. Introduction

Dynamic binary translation (DBT) is a technique that was developed for architectural compatibility, i.e., to run machine binary code on architectures different from the one it was compiled for. This technique also eases the bridging of legacy systems to cheaper and up-to-date platforms [1,2], assists virtualization systems and promotes legacy support, providing binary compatibility with minimal non-recurring engineering. Contrarily to static binary translation (BT), and because translation is performed dynamically, it usually costs to a software-implemented DBT thousands of instructions to translate and optimize a source Instruction Set Architecture (ISA) instruction, which makes DBT a hard challenge to perform in low-end devices, which are commonly scarce in resources. The major costs associated with a DBT system comprises several translation and emulation overheads, along with other potential runtime costs. Such overheads are mainly associated to code decoding and translation, condition codes (CC) handling (through emulation), memory management techniques, peripherals remapping and interrupt handling.

DBT has been widely addressed in the past. Borin et al. presented a strategy to identify the main sources of overhead involved in the process of DBT [3]. Despite the study concerning DBT aimed at desktop processors, thus, not applied directly to embedded systems, evaluation parameters such as cold code translation and translated code execution are transversal to the DBT topic. Among other suggestions, the authors point that research in overhead reduction through hardware support should be conducted in order to achieve near zero overhead DBT. Yao et al. identified that common DBT systems suffer performance loss because of architectural heterogeneity among ISAs, control flow and context switches [4]. In the same work, it is proposed an field-programmable gate array (FPGA)-based hardware/software co-designed acceleration solution, achieved through register replication in reconfigurable hardware and ISA extensions. The authors claim a global speed-up of 56.1%, but provide very shallow details on the integration of used techniques with their DBT engine characteristics. Despite efficient, the approach comes with the cost of architectural modifications to the target processor.

DBTIM is a hardware-assisted DBT architecture for full virtualization [5]. The solution targets high-performance systems and uses a reconfigurable DBT chip, deployed in a dual in-line memory module (DIMM), coupled with a motherboard in order to provide full hardware virtualization of the host central processing unit (CPU). The DBT chip receives the source code and the translation request through the memory interface, processes the request and delivers the translated code through the same mechanism. The approach is an example of hardware integration over traditional systems, without requiring architectural modifications to the target architecture. The use of FPGA fabric to promote binary compatibility is advocated in [6] and in [7]. The former proposes the use of hardware to promote binary compatibility, using reconfigurable coarse grained units to execute legacy functionalities through a Dynamic Instruction Merging technique, a form of BT in hardware. In [7] the authors attempt to understand the challenges of applying reconfigurable computing to accelerate DBT during runtime using co-processors. The method is based in the detection of execution patterns in the source code upon its profiling and subsequent loading of the accelerator's bitstream to the FPGA.

Approaches to DBT on resource-constrained embedded systems have also been explored and evaluated by authors, leading to several acceleration and improvement suggestions that rely on reconfigurable hardware [8]. This work evaluates features of the implemented DBT engine that would mostly benefit from hardware acceleration, e.g., the CCs handling. Software to hardware offloading is a common acceleration procedure used when software-only approaches do not meet the performance requirements. There are however other motivations to embark on software to hardware tasks migration, namely, functionality extensions and shortfalls overcome. Despite the

978-1-5386-7108-5/18/$31.00 ©2018 IEEE

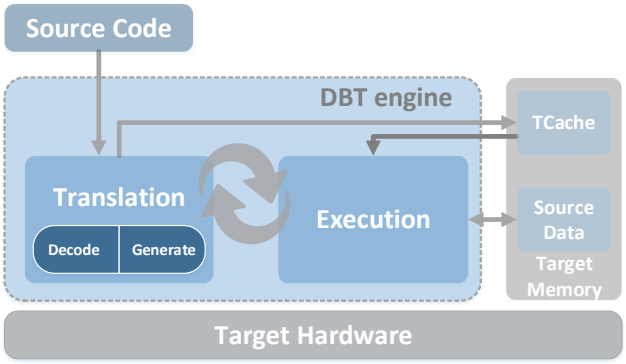


Fig. 1: DBT engine architectural model.

aforementioned mentioned works tackling the main challenges of a DBT system, hardware-based approaches to tackle the management and eviction of a Translation Cache (TCache) have never been attempt. This article presents a hardware-based mechanism to accelerate and extend the functionalities of DBT applied to the resource-constrained embedded devices, approaching hardware offloading to address some limitations of the DBT engine regarding the TCache management mechanism. The TCache management module was deployed in a non-intrusive hybrid acceleration architecture compliant with COTS solutions that integrate both a microcontroller system as well as FPGA fabric.

II. Translation Cache in DBT

The conceptual model of a DBT system is depicted in Figure 1. It is generally composed by (1) a DBT engine, which contains the Translation and the Execution modules; (2) the source binary code; (3) a Translation cache/buffer; (4) the (emulated) guest data source; and (5) the target host hardware. Regarding the Execution and Translation, the first refers to the native execution of the translated code, while the second can be further split into (6) the Decoder and (7) the Generator sub-modules. For optimization purposes, an adaptable DBT system can also profile program runtime behavior and optimize blocks of frequently executed instructions, designated by basic blocks (BB), which are considered the basic unit of translation, i.e., a sequence of instructions likely to be executed as a whole, composed by one entry point and one exit point [1].

The normal execution flow can be summarized as follows: The DBT engine manages the Translation and Execution processes of the source binaries, which is handled in units of BBs, where each BB is identified by its first instruction's address, i.e., the source architecture's program counter (PC) value. After translation, each source BB will generate its equivalent translated BB (TBB) stored in the TCache memory. The translation process starts with a query of the source PC value to the TCache. If the BB is already translated and its corresponding TBB is in cache, the TCache address where the TBB is stored is fetched and the DBT engine switches to the Execution environment in order to run the TBB block. After the TBB's execution, the DBT will return its execution and perform another query to the TCache for the next source BB to translate and/or execute. If the source PC is not yet translated, then a new translation must be performed. This is done by adding a new entry to the TCache for the new BB, and saving the address (for later execution) where the generated code of the TBB starts. Then, the Translation process starts its loop until a control flow instruction is fetched. Each instruction is decoded and translated individually and the PC updated to the next value, accordingly with the instruction length. The previously saved TBB location address is now used by the DBT to switch to the execution of the new translated BB.

This article focus on the TCache component, which is part of the DBTOR, an in-house DBT system developed by authors that targets resource-constrained embedded systems. TCache is a translation buffer where the translated code is stored prior its execution and it must be allocated on a section of the host machine's memory with write access and execution privileges. It needs to support content management and replacement mechanisms to accommodate incoming translations from the Generator and to keep record of the stored BBs.

A. Related Work

TCache management in DBT has already been approached in the literature. Baiocchi et al. [9] use scratchpad memory (SPM) as a auxiliary memory for quick context switch between the Translation and Execution environments, by reducing the translated code and by delegating code caching operations in the SPM. However, the context switch is extremely reduced (one instruction to save the return address and another to move the data memory base address). Furthermore, SPM, or Tightly Coupled Memory (TCM), is not commonly present in the resource-constrained low-budget embedded devices. In [10] the authors resort to hardware techniques in order to manage the code cache either in DBT or dynamic optimizers. Hazelwood et al. [11] identify and study the TCache performance on DBT, presenting a framework to access and manipulate the translation cache of a binary instrumentation system named PIN. Chen et al. developed extensive work in this topic, suggesting hardware to assist a specialized instruction decode cache, the DICache [12]–[15]. Despite the work presenting substantial improvements over software-only deployments, it proposes the integration of hardware extensions at architectural level into an intellectual property (IP) Arm core processor. Following Baiocchi, Yao et al. [4] also integrate SPM in FPGA to reduce context switching overheads. Furthermore, they present a hardware deployment of the mechanism proposed in [10], as a simple look-up-table (LUT) composed by a content-addressable memory (CAM) and a RAM. This mechanism seems to fit to the application scenario proposed in this work, but it still uses a software hash table as a secondary mechanism to decide if the TBB is cached or not.

B. TCache Requirements

The TCache, also called a fragment cache, is the storage entity where the TBBs are stored to be executed. Besides the storage functionalities it relates the original BBs with

the location of the TBBs. This association between source and target addresses is essential to DBT process because it records the equivalence between source and target PCs, and consequently a BB and its TBB. Due to its characteristics, this entity requires different features from the source binaries cache, which are described bellow:

1) **Write and Execution Privileges**: In order to store and natively execute ISA source code (TBBs stores in memory), two conditions must be fulfilled: (1) the memory where the TCache is allocated must be rewritable during run-time and (2) it must have execution privileges.
2) **Content Management**: The TCache requires a data structure to manage and locate the stored TBBs. This has two purposes: (1) check if the queried BB was already translated and (2) return the location of the TBB. In case of a hit (i.e., the BB was translated and the TBB is stored in the TCache) the TBB's address must be returned to the DBT engine to be executed, otherwise a miss is passed to the DBT engine to start a new translation. There are multiple suitable data structure for this purpose, and its complexity and overhead must be considered. Implementation apart, the TCache inputs a source PC (i.e., BB base address), which should be used as a unique key to locate the equivalent TBB.
3) **Eviction Mechanism**: When the TCache reaches its maximum capacity, its content must be evicted in order to accept new TBBs. This might be a partial or total eviction (i.e., cache flush), depending on the implementation. However, partial eviction requires coherency enforcement when directly linking BB during translation, which might contribute with excessive overheads.

Regarding the TCache content management, this work approached two software-based techniques (using linked list and hash table data structures) and one novel hardware-based technique, which are further analyzed, evaluated and compared with each other.

III. Hardware-assisted Translation Cache

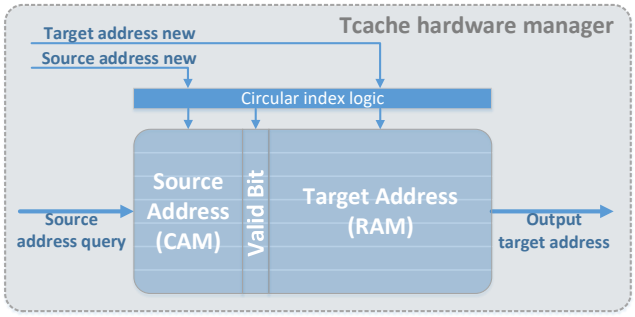


Fig. 2: TCache hardware manager diagram.

On an effort to speed up the translated code management (i.e., to add and search translation entries) and obtaining a scalable cache management method, the TCache management effort is delegated to hardware through an FPGA-based solution. The approach followed by [4] is well suited to be applied because it is non-intrusive and takes full benefit of available hardware, but it still relies on a heavy software-based backup mechanism to handle missing translations from the hardware management. The software hash table mechanism should be avoided in the resource-constrained embedded systems, because of the well known latency it originates. A full hardware TCache approach was considered, but issues were faced regarding its implementation. A full hardware cache is a straight-forward implementation, either following a fully-associative, direct-mapped or set-associative policy. However a TCache does not exactly resembles a cache, but rather a buffer. While a typical cache stores a determined memory position, which the size is known and equal to every entry, a TCache does not contain the equivalent representation of the source code at every address. Each translation is only addressable by its entry address and its size is variable and unknown upon its creation. Dealing with such traits in hardware would involve a complicated mechanism to manage the available space, resulting in small improvements, since most of the overhead associated with the TCache is related with the management mechanism, rather than the data caching and memory accesses.

A modified approach of [4] was followed, however the usage of software hash tables was avoided. The TBB caching, the eviction policy and the available space management are processed using a software approach, while adding new TBB entries and searching for a TBB are processed by auxiliary hardware. The hardware implementation is based on the regular TCache memory space allocated on the target data memory plus a LUT, similar to a fully associative cache (CAM + RAM) on a integrated implementation, as depicted in Figure 2. In this approach, and diverting from the one presented in [4], the output of the CAM is not returned to the microprocessor. In case of a source address hit, the target address is directly forwarded to the microprocessor. If an address miss occurs, then the address `0x00000000` is returned, indicating that the source address is not yet translated. This address may be used in case of a miss because the TCache memory is never allocated at the bottom of the source data memory. There is one additional valid bit in the architecture to prevent false hits during the first accesses and after TCache resets. Although the number of TBBs that fit into the TCache memory space varies, the number of the hardware LUT entries is fixed. To deal with this, other approaches include a software hash table to index additional TBBs after the hardware LUT being full. This induces the penalty of calculating a hash key and searching the hash table every time a TCache miss occurs, even when the TCache is not full.

The implemented approach avoids the use of the software hash table through an adjustment to the new entry insertion mechanism. It is implemented resembling a circular list. The new entries are inserted sequentially until the insertion index overflows and starts to overwrite the first entries. The eventual look-ups of overwritten entries will generate a false miss because the translation is indeed stored at the TCache allocated memory, but its look-up position was given to another TBBs, in favor of using the remaining TCache allocated memory

before requiring a full eviction. Moreover, older entries are less likely to be required. The hardware LUT size is modifiable through a parameter and should adequate to the TCache size, which translates into the typical number of TBB per TCache filling. From the benchmarks used it was empirically determined the following correspondence: 32 KB - 256 entries; 16 KB - 128 entries; 8 KB - 64 entries; 4 KB - 32 entries. The hardware look-up is performed in one clock cycle, plus the bus access latencies, which represent a total of 5 clock cycles. The access time is deterministic and therefore remains constant, no matter the number of TBBs in cache.

IV. Evaluation

To evaluate how the proposed contributions perform, it is required to provide input binaries to the DBTOR. The binaries should replicate the type of programs' behavior that run in the source architectures. Since this is also the purpose of any benchmark suit (i.e., mimic a typical workload of a system), it was decided to use a benchmark's binaries as the source binaries for the DBTOR. Bristol Energy Efficiency Benchmark Suite (BEEBS) is a set of ten benchmarks ported from other suites, selected accordingly with its type of operations (branching, memory intensity, integer and floating point operations), its applicability for resource-constrained embedded systems and required porting effort, in order to evaluate the energy consumption characteristics of the target platform [16].

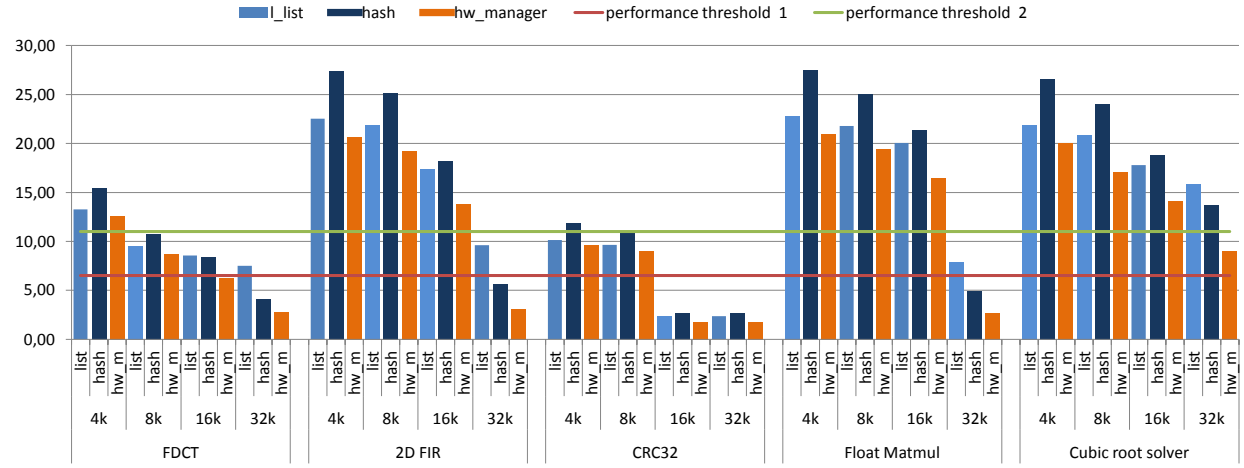
To test and demonstrate the feasibility of the DBTOR, we have paired two well-know architectures, Intel 8051 and Arm Cortex-M3, for the source and target ISAs, respectively. Each benchmark binary file was loaded into the flash memory of a reconfigurable System-on-Chip (SoC), the SmartFusion2 from Microsemi, which includes an Arm Cortex-M3 hard-core processor, besides flash-based FPGA technology. One at the time, the binary files were loaded into the code cache (CCache) and executed through the DBT engine. The execution was timed through a 64-bit timer from the Cortex-M3 hard-core, clocked at the same speed as the SoC, 122 MHz. The timer is started before calling the `runDBT()` function and stopped after returning from it. The tests were repeated for four different TCache sizes: 4 KB, 8 KB, 16 KB and 32 KB. The TCache minimum size must be enough to fit the largest TBB found during translation, due to the translator not supporting BB partitioning yet. It was determined experimentally that the biggest TBB was nearly 3200 bytes long, so the lowest TCache size was set to 4 KB. The biggest cache size is limited by the system's available memory, excluding heap and C stack utilization and variables. On the test platform, the ceiling TCache size was set to 32 KB. Besides, and considering that the translator targets low-budget embedded systems, it was considered that these TCache sizes were a good representation of the resources commonly offered by these platforms.

The hardware-managed hybrid TCache is seamlessly integrated in the developed DBT engine and its interface methods remain the same as the linked list and hash table approaches. The peripheral is connected through the AMBA 3 AHB-Lite bus and mapped in memory through a register interface.

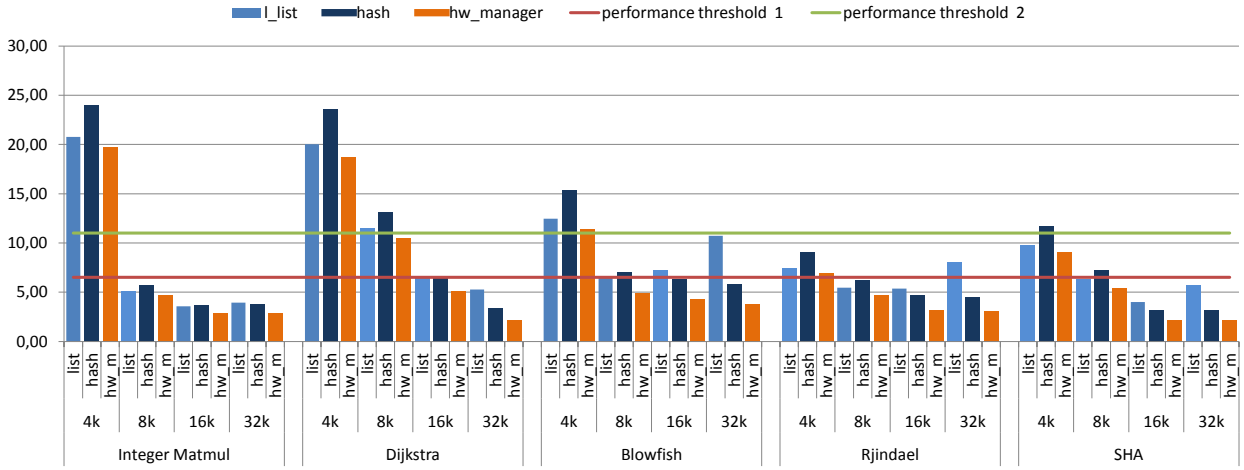
A. BEEBS Benchmark Suite Results

Figure 3 displays the obtained results from the performed experiments. The experiments consisted in evaluating three different TCache management approaches (linked list, hash table and hardware-based TCache manager) for different TCache sizes (4 KB, 8 KB, 16 KB and 32 KB). The graphic shows the ratio between the target execution clock cycles and the native execution clock cycles of every benchmarks, for every TCache size. Lower bars indicate smaller ratios between target and source clocks, which invoke better performance. The ratio between the target and source execution clock cycles varies between approximately 23× (Float Matmul, 4 KB) and 2.5× (CRC32, 16 KB and 32 KB) slower. The results are uplifting, considering the minimalist approach followed (deployment lacks of advanced decoding algorithms, unoptimized code generation, simplistic TCache), the use of an intermediate representation (IR) for multiple source/target architectures bridging, and the one-to-many instructions mapping. For a time based comparison, and on an speculative exercise, the Cortex-M3 mainstream core line with a clock speed of 72 MHz was selected, considering the modern Cortex-M3 processor's clock speeds variation from a few dozens of MHz (32 MHz for the low-power families) up to a few hundred MHz (216 MHz in the powerful Cortex-M7). For the source clock speed, the common MCS-51 legacy cores used a 11.059 MHz clock frequency. This originates a target/source clock ratio of 6.51, represented in Figure 3 as the "performance threshold 1". Under such condition, 13 out of the total 40 tests would be running faster in the translation engine than on the native platform. Going even further, and taking the obtained deployment of the translator, running at 122 MHz on the Microsemi's SmartFusion2 Arm Cortex-M3 core, the clock ratio of 11× is represented as a the "performance threshold 2". Under such condition, 25× of the tests execute in less time under the translator than under native execution, representing 62.5% of the tests.

1) TCache Size Variation: The size of the TCache has a direct impact on the execution performance. A size increase most commonly causes a performance increase, but the relation is not linear. There are even benchmarks where the TCache size increase does not correlate with the performance gain. The fundamental fact for the performance increase is that a bigger TCache accommodates more translated code, thus fewer translations must be evicted in order to give place to newer translations. This causes more TCache hits and less misses, reducing the issuing of repeated translations of code that was still in use but needs to be discarded. There are tests where the TCache variation has a sudden impact on the performance, such as the CRC32 (8 KB to 16 KB), Float Matmul (16 KB to 32 KB) and Integer Matmul (4 KB to 8 KB). This is due to occasional relations between the benchmarks' cycles size and the TCache size, suggesting that the performance bumps occur when the TCache size becomes large enough to accommodate translations of the full source binaries, or at least an extensively executed cycle(s). However,



(a) FDCT, 2D FIR, CRC32, Float Matmul and Cubic root solver.



(b) Integer Matmul, Dijkstra, Blowfish, Rjindael and SHA.

Fig. 3: Target/source global execution ratio, for linked list, hash table and hardware-based TCache management systems.

this phenomena does not directly explain why benchmarks such as the Blowfish, Rjindael and SHA do not consecutively reduce their execution time with the TCache size increase. On these three cases, the performance improvement for a TCache bigger than 4 KB, is clearly due to the size increase. However, the performance decrease, specially for the largest TCache size (32 KB), can not be explained with TCache and TBB size relations. It was found that in these cases, what was deployed as a simpler and apparently effective solution, degrades the performance for greater TCache sizes and large binaries whose translation do not fit completely in the TCache. The cause is the TCache search mechanism, deployed as a linked list and with insertion and access at the tail, for spatial locality of reference advantage purposes. The size of the TBB also plays a role in this assertion, because the smaller the TBBs are, the more TBBs will be accommodated in the TCache, and the more extensive the linked lists becomes, increasing the search time for missed TBB prior to order a new translation.

2) Software-based TCache Mechanisms: Two TCache management algorithms, linked list and one based on a minimal and efficient hash table, were deployed. This allows to compare both implementations and evaluate the impact of the constant-time search approach of the hash table method. It is possible to compare both deployments and observe that none fully exceeds the other. The linked list management results show better performance for smaller TCache sizes (4 KB and 8 KB), while for an intermediate 16 KB size there is no consensus. For the greater size (32 KB) the results for the TCache managed by hash table always overpass the linked list results. Nonetheless, hash table managed TCache tests perform consistently better with the TCache capacity increments. The explanation relies on the fact that while the search time for the linked list managed TCache varies with the number of TBB in the TCache, in the hash table managed the search time is always the same. This leads to although the hash key computation overhead being considerable, it is exceeded by the search time when a large number of TBB are cached. This is the case for some of the 16 KB results and all of the 32 KB results, except when all the TBB fit into the TCache. Thus, hash tables have prejudicial impact on small TCache sizes

TABLE I: FPGA resources utilization of TCache manager.

Entries	*4LUT*	*DFF*	*Combined Resources*
32	1357	2109	3466
64	2704	3678	6382
128	5398	6816	12214
256	10785	13092	23877

configurations of the DBT engine, because of the implied hash key algorithm computation, however this cost starts to pay off for greater TCache sizes.

3) Hardware-assisted TCache vs. Software-based TCache: The new approach was evaluated using the same method used to compare the linked list and hash table TCache management approaches, with standard CC evaluation for the non full legacy support version of the DBTOR. The obtained performance exceeds both of the previously presented management techniques in all tests, either for short or long programs, with big or small TCache sizes. The hardware LUT results in faster management than the simple linked list approach and does not show the performance degradation on the greater TCache sizes observed in the latter approach. Furthermore, since its search time remains constant regardless of the number of TBBs, it outperforms the hash table management on the longer programs for the greater TCache sizes. The results are on average 25% and 26% better than the linked list and hash table implementations, respectively, with the highest performance increase for the linked list approach with a 32KB TCache.

B. FPGA Resources Utilization

Table I summarizes the synthesis results on the SmartFusion2 FPGA technology, for the different suggested entries count, in terms of FPGA resources (4-input Look-Up Table (4LUT), and D-type flip-flop (DFF)). As the entries number increases, the number of resources needed also increases linearly. However, due to the large number of available resources, this causes minimal impact on the final solution.

V. Conclusion and Future Work

This article addresses the functionality extension of the DBT engine using external hardware support. A TCache partially deployed in hardware and a COTS compliant architecture for DBT functionality extension and hardware acceleration were presented. The TCache hardware management with a circular BB registry dismisses the use of a hash table or other secondary software management mechanisms, resulting in significant performance enhancement, compared with hardware-only management methods, for every test scenario. The hardware LUT has a reduced and fixed insertion and search time, leading to better performance and scalability (nearly 25% and 26% better than the linked list and hash table implementations).

The proposed acceleration and functionality extension architecture is based on an external hardware module, integrated as a bus sniffer, which results in a hardware and non-intrusive execution flow technique, never tried before in the state of the art. As on-going work, the bus sniffer is being tested and used to trigger software or hardware modules to serve the DBTOR, based on the source architecture memory accesses, providing great flexibility on the type of application to serve without disturbing the base DBT program flow. Three types of application to the proposed bus sniffer can be adopted: to handle the CC, to remap source peripherals and to provide interrupt support.

VI. Acknowledgments

This work has been supported by COMPETE: POCI-01-0145-FEDER-007043 and FCT - *Fundação para a Ciência e Tecnologia* within the Project Scope: UID/CEC/00319/2013.

References

[1] Probst, Mark, "Dynamic Binary Translator," in *UKUUG Linux Developers' Conference Linux 2002, 4-7 July 2002, Bristol*, 2002, pp. 1–12.

[2] M. Gschwind, E. R. Altman, S. Sathaye, P. Ledak, and D. Appenzeller, "Dynamic and transparent binary translation," *Computer*, vol. 33, no. 3, pp. 54–59, Mar 2000.

[3] E. Borin and Y. Wu, "Characterization of dbt overhead," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 2009, pp. 178–187.

[4] Y. Yao, Z. Lu, Q. Shi, and W. Chen, "FPGA based hardware-software co-designed dynamic binary translation system," *2013 23rd International Conference on Field Programmable Logic and Applications, FPL 2013 - Proceedings. IEEE Computer Society.*, pp. 0–3, 2013.

[5] W. Chen, H. Lu, L. Shen, Z. Wang, and N. Xiao, "DBTIM: An Advanced Hardware Assisted Full Virtualization Architecture," *2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, no. 2007, pp. 399–404, Dec. 2008.

[6] A. C. S. Beck, M. B. Rutzig, G. Gaydadjiev, and L. Carro, "Transparent reconfigurable acceleration for heterogeneous embedded applications," *Design, Automation and Test in Europe, DATE*, pp. 1208–1213, 2008.

[7] P. Kinsman and N. Nicolici, "Dynamic binary translation to a reconfigurable target for on-the-fly acceleration," *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pp. 286–287, 2011.

[8] F. Salgado, T. Gomes, S. Pinto, J. Cabral, and A. Tavares, "Condition codes evaluation on dynamic binary translation for embedded platforms," *IEEE Embedded Systems Letters*, vol. 9, no. 3, pp. 89–92, Sept 2017.

[9] J. a. Baiocchi, B. R. Childers, J. W. Davidson, and J. D. Hiser, "Enabling dynamic binary translation in embedded systems with scratchpad memory," *ACM Transactions on Embedded Computing Systems*, vol. 11, no. 4, pp. 1–33, dec 2012.

[10] H.-S. Kim and J. Smith, "Hardware support for control transfers in code caches," in *22nd Digital Avionics Systems Conference. Proceedings (Cat. No.03CH37449)*. IEEE Comput. Soc, 2003, pp. 253–264.

[11] K. Hazelwood and R. Cohn, "A Cross-Architectural Interface for Code Cache Manipulation," in *International Symposium on Code Generation and Optimization (CGO'06)*. IEEE, 2006, pp. 17–27.

[12] W. Chen, Z. Wang, H. Lu, L. Shen, N. Xiao, and Z. Zheng, "A Hardware Approach for Reducing Interpretation Overhead," *2009 Ninth IEEE International Conference on Computer and Information Technology*, pp. 98–103, 2009.

[13] W. Chen, D. Chen, and Z. Wang, "An approach to minimizing the interpretation overhead in Dynamic Binary Translation," *The Journal of Supercomputing*, vol. 61, no. 3, pp. 804–825, Jun. 2011.

[14] W. Chen, L. Shen, H. Lu, Z. Wang, and N. Xiao, "A light-weight code cache design for dynamic binary translation," *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, pp. 120–125, 2009.

[15] W. Chen, Z. Wang, and D. Chen, "An emulator for executing IA-32 applications on ARM-based systems," *Journal of Computers*, vol. 5, no. 7, pp. 1133–1141, 2010.

[16] J. Pallister, S. J. Hollis, and J. Bennett, "BEEBS: open benchmarks for energy measurements on embedded platforms," *CoRR*, vol. abs/1308.5174, 2013.