Aggressive Memory Speculation in HW/SW Co-Designed Machines

Simon Rokicki, Erven Rohou, Steven Derrien

To cite this version:

Simon Rokicki, Erven Rohou, Steven Derrien. Aggressive Memory Speculation in HW/SW Co-Designed Machines. DATE 2019 - 22nd IEEE/ACM Design, Automation and Test in Europe, Mar 2019, Florence, Italy. pp.332-335, 10.23919/DATE.2019.8715010 . hal-01941876

HAL Id: hal-01941876
https://hal.archives-ouvertes.fr/hal-01941876
Submitted on 2 Dec 2018

Aggressive Memory Speculation in HW/SW Co-Designed Machines

Simon Rokicki Erven Rohou Steven Derrien
Univ Rennes, INRIA, CNRS, IRISA

Abstract—**Single-ISA heterogeneous systems (such as ARM big.LITTLE) are an attractive solution for embedded platforms as they expose performance/energy trade-offs directly to the operating system. Recent works have demonstrated the ability to increase their efficiency by using VLIW cores, supported through Dynamic Binary Translation (DBT) to maintain the illusion of a single-ISA system. However, VLIW cores cannot rival with Out-of-Order (OoO) cores when it comes to performance, mainly because they do not use speculative execution. In this work, we study how it is possible to use memory dependency speculation during the DBT process. Our approach enables fine-grained speculation optimizations thanks to a combination of hardware and software. Our results show that our approach leads to a geo-mean speed-up of 10% at the price of a 7% area overhead.**

I. Introduction

The need for performance and energy efficiency has led to hardware platforms where energy/performance trade-off can be adjusted dynamically (e.g through Dynamic Voltage and Frequency Scaling (DVFS)). However, with the end of Dennard's scaling, the efficiency of DVFS has been decreasing, restricting the trade-off spectrum. This has led to the emergence of single-ISA heterogeneous multi-processor systems, where the use of different types of core exposes even more energy/performance trade-offs. For example, the ARM big.LITTLE architecture features low-power in-order cores and high-performance Out-of-Order processors [1]. In such systems, dynamic adaptation is made possible through a same single instruction set for all cores, enabling quasi-transparent dynamic code migration between cores.

It is possible to enrich these platforms with VLIW cores, which offer excellent energy/performance trade-off for certain types of workloads (compute intensive loops with predictable control-flow). However because VLIW processors expose parallelism directly in their instruction set, transparent code migration is not possible. This restriction can be overcome through Dynamic Binary Translation (DBT) techniques, which can be used to maintain the illusion of a single-ISA system. This approach was followed by Transmeta with the Code Morphing Software (CMS) to allow the execution of x86 binaries on a VLIW core [2] and more recently with NVidia's Denver architecture for the ARM ISA [3]. Because the DBT process is directly managed by the hardware (and thus transparent to the user), we will refer to such architecture as *codesigned machines*.

In a codesigned machine, the DBT involves additional computation at run-time to translate and optimize the binaries, which negatively impacts the performance. In the meantime, taking advantage of the VLIW core implies costly compiler optimizations (instruction scheduling, etc.). Because of this, designing an efficient DBT is very challenging, yet little is known about existing commercial implementations.

As of today, the performance of codesigned machines falls behind that of an OoO core. There is empirical evidence [4] that this performance gap is mostly due to speculative execution abilities in OoO cores. To close this performance gap, the DBT process must, therefore, use similar techniques.

In this work, we study how memory speculation can be implemented in codesigned machines. The proposed system dynamically identifies speculation groups and generates speculative code, along with support for rollback. Misspeculations are dynamically detected through a light-weight hardware mechanism which shares similarity with OoO Load-Store Queues (LSQ). Because the speculation process is managed by the DBT engine, it offers a fine grain control, enabling iterative continuous optimization. Our flow was implemented within Hybrid-DBT [5], an open-source codesigned machine aimed at supporting research on the topic. More specifically, our contributions are the following:

- An complete open-source implementation of memory dependence speculation in a codesigned machine, including modifications to the host VLIW core and to the hardware accelerated DBT.
- A quantitative analysis of the performance benefits and area overhead of the approach, along with a comparison against reference RISC-V implementations.

The remaining of this document is organized as follows : section II presents the hardware and software modifications on Hybrid-DBT and Section III presents the experimental study we conducted.

II. Proposed approach

In this section, we describe how we support speculative memory accesses in the Hybrid-DBT framework. Our approach builds on the notion of memory instruction groups, which consists of a set of load/store operations where load instruction can be speculatively executed before stores. In the following, we summarize how misspeculated operations are detected and corrected by our modified DBT flow depicted in Figure 1.

- The whole speculation process is managed by the software optimizer, running on the DBT processor. It is in charge of building speculation groups and deciding whether speculation should be activated (or not). A more detailed description of this stage is provided in Section II-A.

- We extended our current processor design with a Partitioned Load-Store Queue (PLSQ), which we use to dynamically detect misspeculations. The PLSQ is also used to profile memory operations and detect frequently aliasing accesses. More details on the PLSQ are provided in Section II-B.

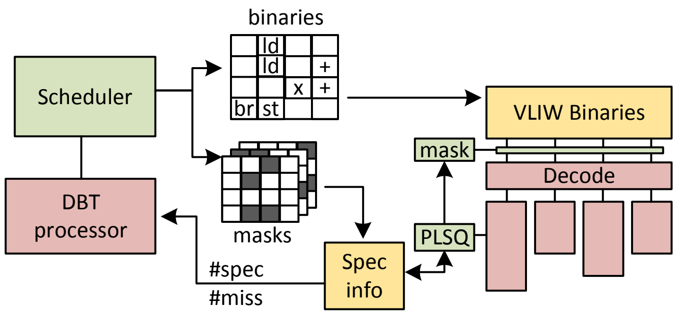

Fig. 1. Overview of the HW/SW co-designed system. It is composed of the DBT processor assisted by a hardware scheduler, the VLIW core with additional support for speculation: Partitioned Load/Store Queue and execution mask.

A. Overview of the speculation process

The speculation process is executed during the second optimization level in Hybrid-DBT. At this stage, the CFG of hot procedures has been built and inter-block optimizations have been applied. The first step consists of building *speculation groups* on frequently executed blocks. A speculation group is a set of store and load operations, where the loads are scheduled after the stores in the original machine code. We allow a given store instruction to be present in only one speculation group, as the group ID is explicitly encoded in the instruction. While building the speculation groups, we have to handle the limited size of the PLSQ: if the DBT finds more memory accesses than what can be stored in a single bank of the PLSQ, it creates another speculation group, which is assigned to another bank. The number of speculation groups inside a single block of binaries cannot be greater than the number of bank in the PLSQ.

Once speculation groups have been built by software DBT, instructions are scheduled without speculation, and profiling is activated. During profiling, the PLSQ keep tracks of how many time memory operations were executed, but also how often they alias with other accesses. This profiling information is stored within the *spec Info* memory.

In the meantime, the software DBT regularly checks all active speculation groups, and based on the profiling information, decides whether speculation (for a given group) should be triggered or not. When speculation is activated, the store to load dependencies in the target speculation group are removed from the Intermediate Representation. The block is then re-scheduled. In this version, speculative loads are flagged with a rollback mask (stored within the *spec Info* memory), which specifies the ID of the speculation group to which it is associated (these instructions will have to be re-executed when misspeculating).

During execution, addresses are checked by the PLSQ. Whenever aliasing is detected, the PC is reset back to the faulty load operation and the basic block is re-executed using the speculation mask. This ensures a coherent machine state when reaching the speculative store.

A key issue is how to choose the threshold that controls the activation/deactivation of the speculation process. In our implementation, speculation is activated when a speculation group has been used more than 70 times with an aliasing rate below 10%. In contrast, speculation is deactivated whenever the aliasing rate increases above 15%. Deactivating/reactivating speculation for the same group too frequently would trigger the scheduling process too often. Similarly, using a higher aliasing rate threshold would increase the cost of rollbacks. It is to note that to obtain the best possible performance this aliasing rate threshold should be chosen on a per-application basis, as rollback cost is not always the same.

B. Partitioned Load Store Queue

In order to dynamically check that the memory dependency speculation is correct, the VLIW core is extended with a Partitioned Load Store Queue (PLSQ). Whereas traditional Load Store Queues check and record every memory operation, our PLSQ is used in a software controlled environment. Consequently, we opted for a partitioned load/store queue, where smaller banks are built to handle a single speculation group. This design reduces the area cost of the PLSQ. The list of memory operations to check is built by the software optimizer and is finely controlled to fit into one bank. The number of bank in the PLSQ defines the number of concurrent speculation groups.

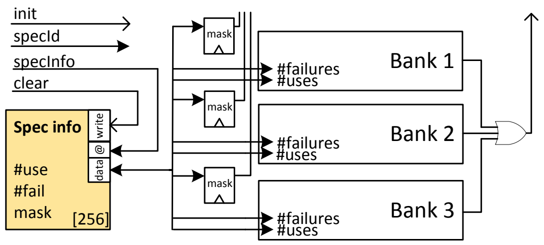

Fig. 2. Global view of the Partitioned Load Store Queue with three banks

The organization of the PLSQ is depicted in Figure 2 with three banks. It embeds a small memory block *specInfo* which contains information on speculation groups: the number of use, of aliasing and the rollback mask. When a *specInit* instruction is executed, the 8-bit address contained in the instruction is used to load this information from the memory and initialize the bank identified by *specId*. The software controlled PLSQ interacts with the VLIW processor through different custom instructions:

- Instruction *specInit* is used to bind a given bank of the PLSQ with a speculation group. It uses an 8-bit address to refer to the correct speculation group in *spec Info* memory and a *specId* field to identify the bank to use.

- Instruction *specClear* resets a bank of the PLSQ, identified by the *specId* field and increments the number of use of the associated speculation group.
- Speculative *loads* and *stores* have a field *specId* to identify the speculation bank to use and a bit *set* to specify if the incoming address should be added in the PLSQ or if the address should be checked.

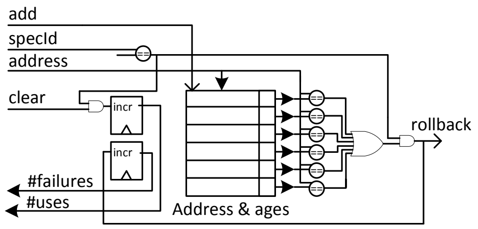

Fig. 3. Organization of a single bank of the PLSQ.

The internal organization of a given bank is shown in Figure 3. Each bank contains (i) two counters (for the number of aliasing and the number of uses), (ii) a set of registers to store the hashed addresses along with the ages of previous speculative instructions and (iii) parallel compare units to check if a given address conflicts with a stored address.

When a speculative memory operation occurs, the *add* bit is used to indicate if the new address has to be added to the queue or if it has to be checked. If this *add* bit is set, the hashed address is pushed to the queue. Note that we use the PLSQ for two purposes: detecting aliasing during speculation but also monitor addresses to detect speculation group candidates. During the speculation phase, memory loads are kept in the PLSQ and verified against the store instructions. During the profiling step, memory stores are kept in the PLSQ and verified against memory loads. Its *age* bit is then set. If the *add* bit is unset, the hashed address is compared with all the hashed addresses stored in the queue, whose age bit is equal to one. If any of these hashes matches, the *rollback* signal is set to one. On a *clear* instruction, all ages stored in the queue are set to zero. The bank also uses two registers to count the number of time a speculation area is used (i.e. the number of *clear* instruction on the bank) and the number of times rollback is triggered.

III. Experimental results

This section provides a quantitative and comparative analysis of the benefit for our memory speculation technique. Our aim is to evaluate both performance gain and area overhead, but also to compare our results against other types of micro-architectures.

These benchmarks have been compiled using the RISC-V GCC tool-chain, with the O3 optimization flag. The generated binaries have been used for all the experiments, with different cycle accurate models. To measure the hardware area, we synthesized the different components with Design Compiler, targeting STMicroelectronics 28 nm technology.

A. Performance results

The first experiment conducted is focused on the performance of our system. We try to highlight i) the performance improvement of our system coming from the new speculation process, ii) how the system compares to a platform similar to the big.LITTLE. To answer those two questions, we measured the performance of all benchmarks using the different simulators. For Hybrid-DBT results, we made the first experiment without the speculation process and another with it.

Figure 4 shows the results obtained during this experiment. Our results show that our memory speculation leads to a geomean speed-up of 10% compared with the Hybrid-DBT framework without speculation. We can observe that our approach brings speed-up from most of Polybench applications. Indeed, the speculation was correctly applied to the most critical loops of the system. However, we notice several applications where no improvements were observed (from *deriche* to *seidel*). Figure 4 also provide a comparison with other micro-architectures. It is interesting to note that, for several kernels, Hybrid-DBT can outperform an OoO core.

B. Area overhead

The first experiment demonstrated that the speculation process brought a speed-up in the execution of benchmark applications. However, our approach also has a cost in hardware. We measured the size of the different hardware components added or modified in the system.

As a baseline, we provide the size of the VLIW core as well as the size of in-order and out-of-order cores. We also synthesized BOOM load/store queue alone.

Area results are displayed in Figure 5. The first observation to do is the difference between the LSQ and the PLSQ, which is around $7\times$ smaller. This is mainly because we do not use store and data queue and because our queue is partitioned, which reduces the number of address comparisons address to do in parallel. We can also observe that the combination of the PLSQ, the extra memory *Spec Info* and the *IR Scheduler* only represents 17% of the VLIW size and the combination is smaller than the BOOM LSQ. It is important to remind that an LSQ in an OoO core cannot work properly without the reorder buffer. As a comparison, OoO core is around $5\times$ larger than the VLIW core, with comparable execution units.

IV. Related Work

In this section, we discuss prior work on memory dependency speculation in the context of DBT toolchains.

A. Dynamic Binary Translation

Previous work on Dynamic Binary Translation mainly focused on executing legacy binaries on an in-order VLIW processor [2], [3]. Transmeta Code Morphing Software (CMS) is used to execute x86 binaries on custom VLIW cores, which are called Crusoe and Efficeon [2]. NVidia's Denver architecture is a 7-issue in-order core capable of translating and executing ARM instructions [3]. Those two processors claim a form of memory dependency speculation. Transmeta's Crusoe processor has access to special load and store operations, which check if an aliasing occurred. However, there is no detailed information on the actual organization of the table used to keep these addresses. Crusoe and Efficeon architectures both

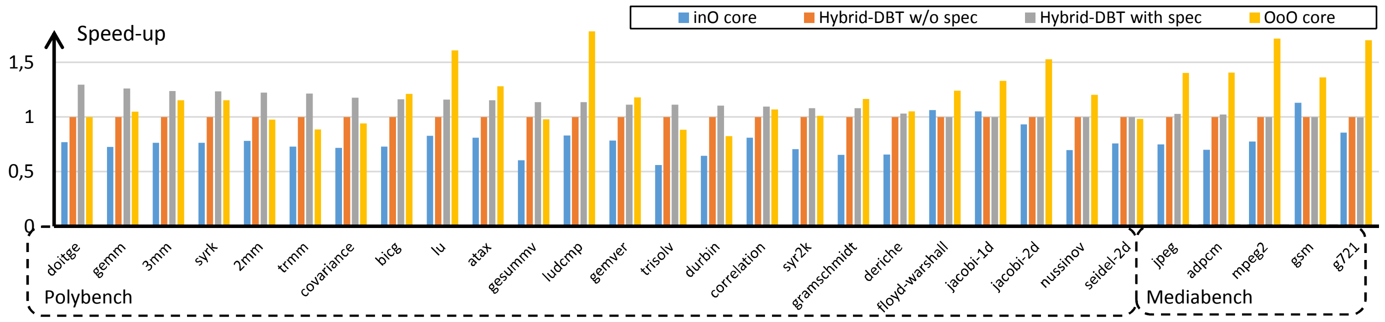

Fig. 4. Speed-up for the different applications, using the non-speculative Hybrid-DBT as a baseline (higher is better)

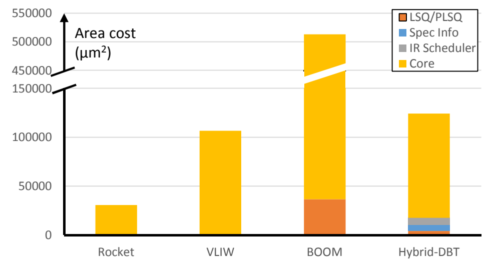

Fig. 5. Area cost of the different components.

have access to a shadow register file with special `commit` and `rollback` instructions, which facilitate rollback in case of misspeculation [2]. NVidia's Denver architecture would use mechanisms similar to those done for the EPIC architecture, that is a mechanism similar to the memory conflict buffer [6], [7]. Again, there is no detailed information on the actual implementation and its effectiveness.

The only open codesign machine framework we are aware of is Hybrid-DBT [5], which does not support speculative execution. In addition to being open-source, we provide precise result on performance and area overhead.

B. *Memory dependency speculation*

The emergence of EPIC architectures (e.g. Itanium) in the late 90's raised the issue of efficient speculative execution for in-order processors [7], [8]. For example, Gallagher et al. proposed to use a Memory Conflict Buffer (MCB) to record the address of speculative load in order to check for an potential aliasing. This approach has strong similarities with our approach, but with a few key differences. MCB is based on a set-associative structure that assigns load to a given bank. If this bank is full when a load is executed, a previous one has to be removed. This triggers the rollback process to ensure that the execution is safe. Of course, the table has to be large enough to reduce the number of unnecessary rollbacks, which is not the case in our approach. The second difference lies in the way rollback is handled: the MCB approach relies on compiler generated correction code to handle these situations, leading to an increased binary size. In their experimental study, the geometric mean of this augmentation is 10%. On the contrary, our approach is based on rollback masks, which are generated by the *IR Scheduler* without code size overhead.

The most obvious difference lies in their use of a static compilation. Indeed, the MCB approach relies on the static compiler to detect eligible load instructions, to schedule instructions and to generate correction code [6]. Relying on the compiler implies that the technique cannot be applied to legacy code. In contrast, our technique is based on DBT and, consequently, is totally transparent to the end-user.

V. Conclusion

In this work, we proposed a technique to support speculative memory access in a codesigned machine framework. Our approach uses a mix of software and hardware to enable aggressive speculation while minimizing overheads. Our results show that performance improvements of 10% can be achieved on compute intensive kernels with regular control flow for an area overhead below 7%. They also suggest that the impact of memory speculation on performance is severely limited by the lack of support for traces/hyperblocks [9]. As a future work, we will investigate the possibility of building traces based on dynamic information, as in [10].

References

[1] P. Greenhalgh, "Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7," *ARM White Paper*, 2011.
[2] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The Transmeta Code Morphing^{TM} Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges," in *CGO'03*, IEEE Computer Society.
[3] D. Boggs, G. Brown, N. Tuck, and K. S. Venkatraman, "Denver: Nvidia's First 64-bit ARM Processor," in *IEEE Micro 2015*, vol. 35, pp. 46–55.
[4] D. S. McFarlin, C. Tucker, and C. Zilles, "Discerning the Dominant Out-of-order Performance Advantage: Is It Speculation or Dynamism?," ASPLOS '13, pp. 241–252, ACM.
[5] S. Rokicki, E. Rohou, and S. Derrien, "Hybrid-DBT: Hardware/Software Dynamic Binary Translation Targeting VLIW," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1.
[6] A. S. Huang, G. Slavenburg, and J. P. Shen, "Speculative Disambiguation: A Compilation Technique for Dynamic Memory Disambiguation," in *ISCA'94*, pp. 200–210, IEEE Computer Society Press.
[7] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B.-C. Cheng, P. R. Eaton, Q. B. Olaniran, and W.-m. W. Hwu, "Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture," in *ISCA '98*, pp. 227–237, IEEE Computer Society.
[8] H. Sharangpani and H. Arora, "Itanium Processor Microarchitecture," in *IEEE Micro 2000*, vol. 20, pp. 24–43.
[9] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective Compiler Support for Predicated Execution Using the Hyperblock," in *IEEE Micro 1992*, MICRO 25, pp. 45–54.
[10] M. T. Yourst and K. Ghose, "Incremental Commit Groups for Non-Atomic Trace Processing," in *IEEE Micro 2005*, pp. 12 pp.–80.