Speculatively Granting Conflicting Accesses on Hardware Transactional Memory

Tomoki Tajimi
Nagoya Institute of Technology, Japan.
tajimi@matlab.nitech.ac.jp

Yuki Futamase
Nagoya Institute of Technology, Japan.
futamase@matlab.nitech.ac.jp

Masaki Hayashi
Nagoya Institute of Technology, Japan.
hayashi@matlab.nitech.ac.jp

Ryota Shioya
The University of Tokyo, Japan.
shioya@ci.i.u-tokyo.ac.jp

Masahiro Goshima
National Institute of Informatics, Japan.
goshima@nii.ac.jp

Tomoaki Tsumura
Nagoya Institute of Technology, Japan.
tsumura@nitech.ac.jp

Transactional memory (TM) [1] is a promising mechanism for improving both of productivity and performance of parallel processing programs on shared-memory systems such as multi-core processors. It can be complementary and/or alternative to traditional lock-based mechanisms that are awkward and intractable for programmers. Hardware transactional memories (HTMs), hardware implementations of TM, are now widely installed to the several latest processors, such as IBM zEC12, Power8, and Intel Haswell. Programmers define critical sections as transactions, and the transactions can be speculatively executed in parallel by HTMs, while traditional lock-based systems completely serialize them. HTMs dynamically detect access conflicts on shared variables between transactions and cancel speculative execution of transactions. Because the canceled execution comes to nothing and causes some performance overhead for rolling back the transaction execution, how to avoid and reduce access conflicts is very important for the performance of HTMs. Many studies [2, 5, 8–11] have been conducted on improving HTM performance, but some type of programs still suffer many access conflicts and large performance overhead, and they should be relieved.

Transactions must guarantee several properties including *Atomicity* and *Isolation*, and access requests that may violate them are generally denied by HTMs. However, we found that some of the requests can be granted without violating Isolation, and this can improve the HTM performance. On HTM, execution of a transaction is naturally speculative, and our proposal **boosts its speculativeness** by speculatively permitting some accesses that are judged as conflicting accesses.

In this presentation, we aim to make the following contributions:

(1) We carefully examine a typical benchmark program and disclose that some access requests can be granted without violating Isolation.

(2) We propose a novel transaction control for speculatively granting some access requests that may violate Isolation, and canceling the accesses when the speculation fails.

Specifically, with traditional HTM systems, an access request on a shared variable is denied for Isolation if the variable has been already modified by another concurrent transaction. However, by examining some typical benchmark programs, we found that if the transaction never accesses the variable again until its commit, the request can be granted, and the access can be permitted without violating Isolation. We propose some extensions for HTM to support such novel access control between transactions.

To exploit this idea, the *last-touch* on each shared variable in each transaction should be managed and maintained. We installed a counter and a small dedicated table to each processor core for this. On each memory access (load/store), the transaction increments the counter value and registers the tuple of the transaction ID, the accessed address, and the counter value on the dedicated table. Receiving an access request on a shared variable, the transaction judges whether its own modification on the variable is completed or not by comparing the registered *last-touch* and currently issued

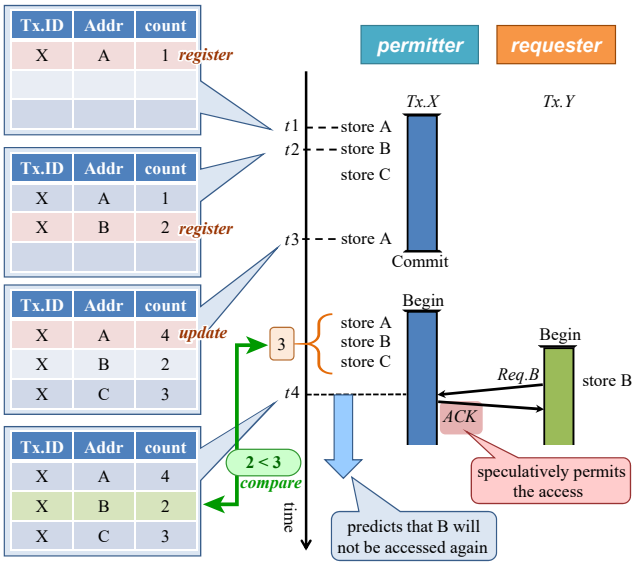

Figure 1: How to permit a conflicting access

MICRO-51 SRC, October 20–24, Fukuoka, Japan

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

memory access count. If it is predicted that the modification is completed, the transaction speculatively grants the access request. In the example shown in Figure 1, *Tx.X* (the transaction labeled with ID '*X*') grants the access request on B from *Tx.Y* (at *t*4), because *Tx.X* has already issued three memory accesses when receiving the request and it is remembered that the second memory access in *Tx.X* is the *last-touch* on B.

A transaction that sends an access request (*requester*) is speculatively permitted its conflicting access on the presupposition that the transaction that grants the request (*permitter*) has completed its modification on the shared variable, will commit, and the modified value of the shared variable will be persistent. If this speculation fails, or the *permitter* accesses the variable again or aborts, the execution of the *requester* must be canceled for Isolation. We also designed the control for this. When a 'permitter' touches a shared variable after the predicted *last-touch* or the *permitter* aborts, it needs to notify it to the *requester* and the *requester* aborts. In contrast, the *requester* must postpone its commit until the corresponding *permitter* commits even if it instantly can. There is another required control for roll-back. As explained, when the *permitter* aborts, the corresponding *requester* also should abort, and the should roll-back together. In that case, the state that should be restored is one of before the beginning of not *requester* but *permitter*. Hence, the *requester* should roll-back first, and after then, the *permitter* should roll-back.

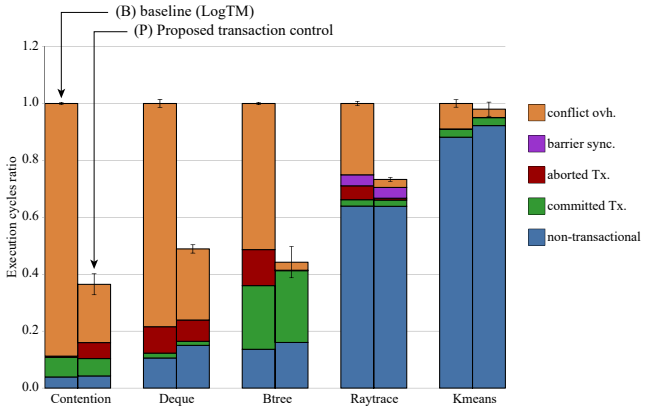

Figure 2: The normalized total sum of execution cycles

We evaluated the novel access control by using a full-system simulator WindRiver Simics [3] in conjunction with customized memory simulators build on Wisconsin GEMS [4]. We used some workloads from GEMS microbench suite [4], SPLASH-2 benchmark suite [12], and STAMP benchmark suite [6]. The evaluation result shown in Figure 2 indicates that the speculative access control can reduce the total execution cycles by 63.3% at a maximum and 38.8% on average, compared with a traditional HTM implementation LogTM [7]. Especially, the conflict overhead is largely reduced and this mainly contributes to the performance improvement.

ACKNOWLEDGMENTS

This work was partially supported by JSPS KAKENHI Grant Numbers JP17H01711, JP17H01764, and JP17K19971.

REFERENCES

[1] Maurice Herlihy et al. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proc. 20th Int'l Symp. on Computer Architecture (ISCA'93)*. 289–300.
[2] Marc Lupon, Grigorios Magklis, and Antonio González. 2010. A Dynamically Adaptable Hardware Transactional Memory. In *Proc. 43rd Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO-43)*. 27–38.
[3] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A Full System Simulation Platform. *Computer* 35, 2 (Feb. 2002), 50–58.
[4] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. 2005. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *ACM SIGARCH Computer Architecture News* 33, 4 (Sep. 2005), 92–99.
[5] A. McDonald, J. Chung, B. D. Caristrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. 2006. Architectural Semantics for Practical Transactional Memory. In *Proc. 33rd Annual Int'l Symp. on Computer Architecture (ISCA'06)*. 53–65.
[6] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proc. IEEE Int'l Symp. on Workload Characterization (IISWC'08)*.
[7] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. 2006. LogTM: Log-based Transactional Memory. In *Proc. 12th Int'l Symp. on High-Performance Computer Architecture (HPCA'06)*. 254–265.
[8] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. 2006. Supporting Nested Transactional Memory in LogTM. In *Proc. 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1–12.
[9] E. Moss and T. Hosking. 2006. Nested Transactional Memory: Model and Preliminary Architecture Sketches.. In *Science of Computer Programming*. 186–201.
[10] A. Shriraman, S. Dwarkadas, and M. L. Scott. 2008. Flexible Decoupled Transactional Memory Support. In *Proc. 35th Annual Int'l Symp. on Computer Architecture (ISCA'08)*. 139–150.
[11] S. Tomic, C. Perfumo, C. Kulkami, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. 2009. Eazyhtm, Eager-lazy Hardware Transactional Memory. In *Proc. 42nd Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO-42)*. 145–155.
[12] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. 22nd Annual Int'l. Symp. on Computer Architecture (ISCA'95)*. 24–36.