Clean-prefetcher: look-ahead prefetching without cache pollution

Minshin Cho¹, Jae Young Hur², and Wooyoung Jang¹, a)

Abstract In this letter, we propose a novel prefetching technique that is free from cache pollution and thus achieves high performance for multicore processors. Unlike the conventional prefetchers that cause incorrect predictions, the proposed prefetcher reads instructions in advance only in determined paths and charges dynamic random access memory (DRAM) cells storing instructions in undetermined paths via refreshing DRAM cells. The DRAM cells highly charged will be quickly accessed. Since caches served by our prefetcher always store useful instructions, as a result, they are free from cache pollution that results in lower cache hit rate. In the case that SPEC CPU2006 benchmarks run on an 8-core processor, the proposed prefetcher consumes more 3.2% DRAM power, but achieves 12% higher performance on average.

Keywords: prefetch, DRAM, refresh

1. Introduction

As the number of cores integrated on a single die increases at faster rate than the improvement of memory performance [1, 2], a memory system becomes a critical performance bottleneck in state-of-the-art multicore processors. Many researchers have studied a variety of approaches for improving memory latency and capacity [3, 4, 5, 6, 7, 8, 9, 10]. Of them, a cache prefetcher that reads instructions from low-speed memories in advance can achieve the notable improvement of memory performance [11]. However, the prefetcher does not always improve memory performance. The reason is that the mispredictions of the prefetcher can evict instructions with high locality from a cache, and then fill useless ones in the cache, called cache pollution [12]. In addition, the prefetcher can make urgent demand requests contend with mispredicted prefetch requests [13] and require meta-data unavailable in multicore processors for reading instructions in advance [14].

In this letter, we propose a novel clean-prefetcher that does not cause any cache pollution and thus improves the overall system performance. Our key observation is twofold. First, prefetching instructions in incorrect paths has critical negative impacts on cache hit rate. Even if a prefetcher is equipped with a reasonable branch predictor, the sum of performance gains from predicting correct paths can be less than that of performance losses from predicting incorrect paths. In particular, the losses can be more serious for multicore processors since cores share the cache polluted by useless instructions. Second, based on the branch prediction, reducing the latency of dynamic random access memories (DRAMs) can be more beneficial than prefetching instructions. Since DRAMs can be accessed much faster depending on the level of charges in the capacitor of DRAM cells [15, 16, 17], techniques that highly keep the charge of target DRAM cells prior to accessing the DRAM cells can be useful. Based on these observations, the major novelty and contribution of this letter are as follows: 1) We propose a novel prefetcher that generates not only demand and prefetch requests, but also row-refresh requests that recharge DRAM cells that are predicted to be accessed, and 2) we propose a memory controller that effectively performs demand, prefetch, and row-refresh requests, and 3) Our clean-prefetcher is evaluated in terms of performance and power consumption.

2. DRAM structure and operation

DRAM has three-dimensional structure, i.e., a bank, a row, and a column [18, 19]. Each bank is activated and deactivated by row activation (ACT) and precharge (PRE) commands, respectively, and read or write commands are executed to an activated bank. DRAM cells start leaking charge after they are disconnected from their bitline [20, 21, 22]. Thus, a refresh command is issued to avoid data loss within an allotted refresh period due to charge leakage from the capacitor of DRAM cells [23]. Each command is completed with the predefined interval time.

Timing parameters relevant to row activation can be further optimized [15, 16, 17] even though they have been determined by the DRAM standards [18]. Fig. 1 shows ACT and PRE operations depending on the level of charge in the capacitor of DRAM cells. When a row activation command is issued, the cell and bitline of DRAM share charges each other. Such charge sharing operation induces $\Delta V$, and then $\Delta V$ is amplified by a sense amplifier. The initial level of charges has an effect on the response time of the sense amplifier. In Figs. 1(a) and 1(b), DRAM cells are partially and fully charged, respectively. After the charge sharing operation, the sensing and restoring speed of the fully charged DRAM cell is faster than that of the partially charged DRAM cell. DRAM timing parameters relevant to sensing and restoring operations have been decided under the assumption that the sense amplifier is driven by the worst $\Delta V$. Thus, the DRAM timing parameters can be reduced if DRAM cells that are soon accessed are detected and charged.

¹ Department of Electronics and Electrical Engineering, Dankook University, Yongin-City, 16890, South Korea
² Department of Electronic Engineering, Jeju National University, Jeju-City, 63243, South Korea
a) wyjang@dankook.ac.kr

DOI: 10.1587/ellex.18.20210027
Received January 19, 2021
Accepted February 1, 2021
Publicized February 9, 2021
Copyedited March 25, 2021
3. Prefetching without cache pollution

As instructions prefetched in a cache increase, cache hit rate can increase owing to improved memory utilization. However, in the case that the working set exceeds the limited cache capacity, other useful instructions with high temporal locality start to be evicted from a cache [11]. Furthermore, even if a prefetcher is equipped with an excellent conditional branch (CB) predictor, it does not always correctly prefetch instructions that will be executed after a CB instruction into a cache [12, 24]. The reason is that the latest applications such as deep learning [25], virtual/augmented reality [26], and high-quality multimedia [27] have extremely complex and irregular memory accesses. If the prefetcher often incorrectly predicts the next path for a CB instruction, instructions with high locality may be evicted from caches, and the caches can be filled with unnecessary instructions [28]. When a multicore processor simultaneously executes a variety of applications, especially, its shared cache that is polluted by useless instructions has more negative impacts on the overall system performance [13]. Therefore, a prefetching technique that does not result in cache pollution is required.

Fig. 2(a) is the structure of the proposed clean-prefetcher that is modified in the wrong-path instruction prefetcher proposed in [12]. The wrong-path instruction prefetcher reads instructions in both taken and not-taken paths of a CB instruction in advance. On the contrary, our prefetcher reads up to the coming CB instruction. A decoder checks if what LAPC reads is a branch instruction, and RAS stores a return address when a subroutine is called. A branch target buffer (BTB) stores the address, target address, and decision of the most recently executed branch instructions. LAC assigns taken or not-taken paths of a CB instruction to LAPC, and transfers prefetch and row-refresh requests to a memory controller. CBTS newly added to our clean-prefetcher tracks a branch instruction in BTB for its assigned path.

Fig. 2(b) shows the operation of our clean-prefetcher with a simple example. We assume that $a0$ is being executed, and other instructions have not been in caches. LAPC starts fetching instructions such as $a1$, $CB0$, etc. in advance in a determined path via a prefetch request, and then stores them in caches. A decoder checks if there are some branch instructions prefetched by LAPC. In this example, it detects $CB0$, and informs LAC of the $CB0$ detection. Next, LAC checks the recently executed $CB0$ in BTB. If any history of $CB0$ has not been in BTB, LAC sends row-refresh requests to a memory controller for recharging DRAM cells that store instructions in both taken and non-taken paths of $CB0$. The absence of the $CB0$ history in BTB means that DRAM cells storing instructions executed after $CB0$ have not been recently used, and thus the DRAM cells may be discharged. In this example, previous $CB0$s were taken and not-taken as shown in the first and forth listings of BTB, respectively. In this case, LAC does not send row-refresh requests to a memory controller since DRAM cells storing instructions in both taken and non-taken paths of $CB0$ have been recently accessed. Then, LAC assigns the taken and not-taken paths of $CB0$ to CBTS 0 and 1, respectively. That is, LAC can send a row-refresh request to a memory controller for a CB path of which the history is not in BTB, and assign CBTS to a CB path of which the history is in BTB.

CBTS 0 and 1 track their assigned paths to find the history of branch instructions in BTB. In this example, $CB1$ was not taken as shown in the first and forth listings of BTB, respectively. In this case, LAC does not send row-refresh requests to a memory controller since DRAM cells storing instructions in both taken and not-taken paths of $CB0$ have been recently accessed. Then, LAC assigns the taken and not-taken paths of $CB0$ to CBTS 0 and 1, respectively. That is, LAC can send a row-refresh request to a memory controller for a CB path of which the history is not in BTB, and assign CBTS to a CB path of which the history is in BTB.

CBTS 0 and 1 track their assigned paths to find the history of branch instructions in BTB. In this example, $CB0$ finds $CB1$. Since the taken path of $CB1$ has been included in the current path and $CB1$ previously executed was not taken as...
shown in the second listing of BTB, LAC does not send row-refresh request to a memory controller and keeps assigning the not-taken path of CB1 to CBT 0. Next, CBT 0 finds CB2 in the third listing of BTB. Since CB2 previously executed was just taken, LAC sends a row-refresh request to a memory controller for the not-taken path of CB2 and assigns the taken path of CB2 to CBT 0. Then, CBT 0 tracks the taken path including an d14, d15, and an instruction with an indirect address (Indir). After Indir is executed, its next instructions can be determined. Since CBT 0 can no longer track CB instructions in BTB, thus, it is free from the assigned path.

Meanwhile, CBT 1 tracks the not-taken path of CB0 in BTB. It sequentially encounters Call, Return, and Jump instructions converted to a CB instruction that is always taken. If CBT finds such a CB instruction in BTB, LAC should make CBT track its taken path and does not send row-refresh requests to a memory controller. Otherwise, CBT no longer tracks any paths. In this example, a Call instruction is in the fifth listing of BTB, and thus CBT 1 tracks a taken path starting at b6. Then, CBT 1 encounters a Return instruction in the sixth listing of BTB. Similarly, it tracks a taken path starting at a5. Next, CBT 1 finds a Jump instruction in the seventh listing of BTB and tries to track a taken path starting at c8. However, since the path has been already tracked by CBT 0, CBT 1 is free from its current path.

Whenever a CB instruction is executed and its path is determined, this procedure is repeated. When the path of CB0 is determined in Fig. 2(b), our clean-prefetcher writes a determined path in BTB and resumes either prefetching instructions in the determined path or row-refreshing DRAM cells storing instructions not recently executed in undetermined paths, referring to BTB. The depth of a path tracked in an instruction sequence depends on the number of CBT and the depth of BTB. Multiple CBTs and a deep BTB can make DRAM cells charged in more rows for fast accesses, but may increase design cost and power consumption.

4. Memory system supporting clean-prefetcher

4.1 Processing row-refresh requests

Our memory controller should perform a new row-refresh request which makes DRAM cells charged to support the proposed clean-prefetcher. The row-refresh request in DRAMs can be performed via a pair of ACT and PRE commands. In the case that a DRAM bank activated for refreshing a target row is not accessed by other demand or prefetch requests, our memory controller can skip a PRE command and operate in an open-page policy. Such an activated row can serve instructions for demand and prefetch requests just with column address strobe (CAS) latency since it saves time to deactivate a different row and activate the refreshed row again. Row-refresh requests should not interfere with not only demand requests, but also prefetch requests since instructions read in advance by our clean-prefetcher unlike conventional prefetchers are used soon. Thus, prefetch requests in our memory controller have a lower priority than demand requests, but a higher priority than others. On the contrary, row-refresh requests are served with a best-effort priority. Furthermore, in the case that any row to be refreshed by a row-refresh request is accessed by demand or prefetch requests, the priority of the row-refresh request is same as that of the demand or prefetch requests.

Our memory controller equips a small cache that stores row addresses recently refreshed (CRARR). The reason is that our clean-prefetcher can repeatedly generate the same row-refresh requests depending on the pattern of conditional branch instructions, and thus power consumption can be increased. For example, if the first listing of BTB is empty in Fig. 2(b), LAC will send a row-refresh request to a memory controller for the taken path of CB0. The row-refresh request makes all the DRAM cells in a single row charged. Since DRAM cells that store not only instructions c8, c9, CB1, c10, and CB2, but also instructions c11, c12, and c13 are in the same row, they are charged together. Let CB0 and CB1 taken and not-taken, respectively, and then, CB2 executed now. Since BTB does not have the history of previous CB2, LAC sends a row-refresh request for DRAM cells storing instructions c11, c12, and c13. However, the DRAM cells have been already charged via the previous row-refresh request. To prevent DRAM cells from being unnecessarily charged, our memory controller stores row addresses recently refreshed in CRARR and filters overlapped row-refresh requests.

4.2 Supporting multiple \( t_{\text{RCD}} \) and \( t_{\text{RAS}} \)

Most of the DRAM timing parameters are programmed via mode register set (MRS) commands. The MRS commands take a long time to be executed, but they do not affect the system performance during power-up and initialization processes. However, in the case that some DRAM timing parameters are necessary to be changed, executing MRS commands while applications are running can result in the serious loss of the system performance. Thus, industrial memory systems usually rarely change DRAM timing parameters via MRS commands while applications are running.

On the contrary, activate-to-internal read or write delay time \( t_{\text{RCD}} \) and activate-to-precharge command period \( t_{\text{RAS}} \) are changed without executing MRS commands according to the Joint Electron Device Engineering Council (JEDEC) standard. Thus, CAS and PRE commands can be issued immediately for \( t_{\text{RCD}} \) and \( t_{\text{RAS}} \) changed, respectively. That is, if DRAM cells in a target row are ready to be accessed via a RAS command, CAS and PRE commands can be issued even though \( t_{\text{RCD}} \) and \( t_{\text{RAS}} \) have not elapsed, respectively. The reason is that \( t_{\text{RCD}} \) and \( t_{\text{RAS}} \) described in the JEDEC standard excessively guarantee that all the discharged DRAM cells in a single row are ready to be accessed. Thus, in the industry, \( t_{\text{RCD}} \) and \( t_{\text{RAS}} \) different from those described in the JEDEC standard have been used. For example, some major DRAM companies fabricate DRAMs requiring shorter \( t_{\text{RCD}} \) and \( t_{\text{RAS}} \) thanks to their improved semiconductor technologies and provide them for a performance-sensitive system. In addition, DRAMs showing longer \( t_{\text{RCD}} \) and \( t_{\text{RAS}} \) are used for a cost-sensitive system after testing and binning in the case that they have no functional problems.

Our memory system issues CAS and PRE commands to
DRAMs for multiple $t_{RCD}$ and $t_{RAS}$, respectively, regardless of whether it is equipped in a system-on-chip (SoC) with cores that may or may not use a pre-refresh request. The memory system decides which $t_{RCD}$ and $t_{RAS}$ should be applied by referring the access and return addresses of a memory request. After a RAS command for activating a target row, CAS and PRE commands are issued when clock cycles for the $t_{RCD}$ and $t_{RAS}$ of the memory request have elapsed, respectively. Therefore, our memory system supporting the proposed clean-prefetcher does not cause any performance penalty even if multiple $t_{RCD}$ and $t_{RAS}$ are applied.

5. Experimental results

We verify the potential improvement of DRAM latency in terms of $t_{RCD}$ and $t_{RAS}$ depending on charge variation in the capacitor of a DRAM cell. Simulation program with integrated circuit emphasis (SPICE) is used to experiment how the charge level of a DRAM cell affects the required activation latency. The improvement of DRAM latency occurs mainly from the sensitivity of a DRAM sense amplifier to $\Delta V$. We implement the circuit of a DRAM sense amplifier using 55nm process technology [29, 30] for simulation parameters such as bitline and DRAM cell capacitance.

Bitline voltage increases at different speeds depending on the initial charge amounts of a DRAM cell. Fig. 3 shows the variation of bitline voltage while a DRAM cell is activated by an ACT command. When a DRAM cell is partially charged, a sense amplifier drives bitline voltage to the ready-to-access voltage level ($V_{RTA}$) in 13.75ns. On the contrary, a fully charged DRAM cell brings bitline voltage up much faster. That is, a bitline connected to a fully charged DRAM cell reaches $V_{RTA}$ in 8.05ns. In JEDEC memory standards, timing parameters are decided by the worst-case scenario for the charge state of a DRAM cell just before performing a refresh operation. If DRAM cells to be accessed are predicted and thus fully charged in advance, they can serve demand and prefetch requests with 5.7ns shorter $t_{RCD}$. Similarly, the charge of a DRAM cell capacitor is restored at different times according to the initial voltage of a DRAM cell. That is, a fully charged cell can achieve 11.2ns shorter $t_{RAS}$ than a partially charged cell. Shortened $t_{RAS}$ can make a bank that includes a refreshed row early deactivated by a PRE command and thus DRAM latency can be reduced when a different row of the bank are accessed.

We modify MARSSx86 [31] and DRAMSim2 [32] simulators for evaluating the proposed clean-prefetcher. MARSSx86 is a cycle-accurate full system simulator for multicore x86 CPUs, and DRAMSim2 is a cycle-accurate memory system simulator for DDR 2/3/4 DRAMs. They are configured as shown in Table I and run SPEC CPU2006 benchmarks [33].

Fig. 4 shows instruction per cycle (IPC), power consumption and cache miss rate of our clean-prefetcher normalized by the wrong-path instruction prefetcher [12]. The proposed prefetcher makes DRAM consume on average 2.4% higher power for four benchmarks executed at the same time, as shown in Fig. 4(a). The reason is that all of the DRAM cells in a single row are fully charged thanks to some instructions that are expected to be accessed. CRARR in our memory controller minimizes this increase in power consumption by filtering overlapped row-refresh requests. On the contrary, our prefetcher achieves on average 10.5% higher performance. This is because DRAM cells fully charged via a row-refresh request can serve demand and prefetch requests with shorter $t_{RCD}$ and $t_{RAS}$, and the cache miss rate resulting...
from our prefetcher decreases on average 2.5%.

In the case that the proposed prefetcher supports eight benchmarks simultaneously, it makes DRAM consume 3.2% more power on average, as shown in Fig. 4(b). On the contrary, the proposed prefetcher achieves on average 12% higher performance. This is because it is free from cache pollution in a last-level cache shared by many cores, instructions with high locality are hardly evicted, and thus the cache miss rate decreases on average 7.2%. As a result, our prefetcher can achieve higher performance in a processor equipping more cores and running more applications simultaneously.

6. Conclusion

Cache pollution resulting from the misprediction of a prefetcher has a significant impact on the performance of multicore processors. The reason is that the mispredicted instruction evicts the high-locality instructions of other cores from a shared last-level cache. Thus, in the letter, we proposed a charge-level-aware prefetcher that does not cause any cache pollution. Rather than predicting and prefetching instructions that may be unused in the next paths of a conditional branch instruction, our prefetcher charges DRAM cells storing the uncertain instructions, and prefetches instructions to be definitely used. Such a fully charged DRAM cell can be accessed with shorter DRAM latency for urgent demand requests. As a result, the proposed prefetcher achieves higher performance than the conventional prefetcher with a conditional branch predictor that causes cache pollution. In addition, our memory controller minimizes the increase in power consumption by filtering overlapped row-refresh requests. In conclusion, our clean-prefetcher can provide more opportunities to support multicore processors with high performance and reasonable cost.

Acknowledgments

The present research was conducted by the research fund of Dankook University in 2019.

References

[1] B.M. Rogers, et al.: “Scaling the bandwidth wall: challenges and avenues for CMP scaling,” ACM SIGARCH Computer Architecture News 37 (2009) 371 (DOI: 10.1145/1555815.1555801).
[2] ITRS: “International technology roadmap for semiconductors,” (2015) http://www.itrs2.net.
[3] D. Lee, et al.: “Tiered-latency DRAM: a low latency and low cost DRAM architecture,” IEEE International Symposium on High Performance Computer Architecture (HPCA) (2013) 615 (DOI: 10.1109/HPCA.2013.6522354).
[4] S.-L. Liu, et al.: “Improving DRAM latency with dynamic asymmetric subarray,” IEEE/ACM International Symposium on Microarchitecture (MICRO) (2015) 255 (DOI: 10.1145/2830772.2830827).
[5] D. Lee, et al.: “Adaptive-latency DRAM: optimizing DRAM timing for the common-case,” IEEE International Symposium on High Performance Computer Architecture (HPCA) (2015) 489 (DOI: 10.1109/HPCA.2015.7056057).
[6] W. Jang: “Enhancing lifetime of phase-change memory for video processor,” IEICE Electron. Express 14 (2017) (DOI: 10.1587/ elex.14.20170402).
[7] P. Chi, et al.: “PRIME: a novel processing-in-memory architecture for neural network computation in ReRAM-based main memory,” ACM/IEEE International Symposium on Computer Architecture (ISCA) (2016) 27 (DOI: 10.1109/ISCA.2016.13).
[8] W. Jang: “Unaligned burst-aware DRAM subsystem,” IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 27 (2019) 2387 (DOI: 10.1109/TVLSI.2019.2922621).
[9] W. Jang: “MLC STT-MRAM-aware memory subsystem for smart image applications,” IEEE Trans. Multimedia 22 (2020) 717 (DOI: 10.1109/TMM.2019.2930342).
[10] T. Park, et al.: “TLC STT-MRAM aware LLC for multicore processor,” IEICE Electron. Express 17 (2020) (DOI: 10.1587/elex.17.20200359).
[11] S. Mitra: “A survey of recent prefetching techniques for processor caches,” ACM Computing Surveys (CSUR) 49 (2016) 35 (DOI: 10.1145/2907071).
[12] J. Pierce, et al.: “Wrong-path instruction prefetching,” IEEE/ACM International Symposium on Microarchitecture (MICRO) (1996) 165 (DOI: 10.1109/MICRO.1996.566459).
[13] O. Mutlu, et al.: “An analysis of the performance impact of wrong-path memory references on out-of-order and runahead execution processors,” IEEE Trans. Comput. 54 (2005) 1556 (DOI: 10.1109/TC.2005.190).
[14] C. Kaynak, et al.: “Shift: shared history instruction fetch for lean-core server processors,” IEEE/ACM International Symposium on Microarchitecture (MICRO) (2013) 272 (DOI: 10.1145/2540708.2540732).
[15] Y. Wang, et al.: “Reducing DRAM latency via charge-level-aware look-ahead partial restoration,” IEEE/ACM International Symposium on Microarchitecture (MICRO) (2018) 298 (DOI: 10.1109/ MICRO.2018.00032).
[16] W. Shih, et al.: “NUAT: a non-uniform access time memory controller,” IEEE International Symposium on High Performance Computer Architecture (HPCA) (2014) 464 (DOI: 10.1109/HPCA.2014.6835956).
[17] H. Hassan, et al.: “ChargeCache: reducing DRAM latency by exploiting row access locality,” IEEE International Symposium on High Performance Computer Architecture (HPCA) (2016) 581 (DOI: 10.1109/HPCA.2016.7466096).
[18] JEDEC specifications: “DDR 1, 2, 3, and 4 SDRAM standards,” http://www.jedec.org, accessed.
[19] B. Jacob, et al.: Memory Systems: Cache, DRAM, Disk (Morgan Kaufmann, 2010).
[20] J. Liu, et al.: “RAIDR: retention-aware intelligent DRAM refresh,” ACM/IEEE International Symposium on Computer Architecture (ISCA) (2012) 1 (DOI: 10.1109/ISCA.2012.6237001).
[21] J. Liu, et al.: “An experimental study of data retention behavior in modern DRAM devices: implications for retention time profiling mechanisms,” ACM/IEEE International Symposium on Computer Architecture (ISCA) (2013) 60 (DOI: 10.1109/ISCA.2016.7446096).
[22] K.K. Chang, et al.: “Improving DRAM performance by parallelizing refreshes with accesses,” IEEE International Symposium on High Performance Computer Architecture (HPCA) (2014) 356 (DOI: 10.1109/HPCA.2014.6835946).
[23] I. Bhatia, et al.: “DRAM refresh mechanisms, penalties, and tradeoffs,” IEEE Trans. Comput. 65 (2016) 108 (DOI: 10.1109/TC.2015.2417540).
[24] E. Bhatia, et al.: “Perceptron-based prefetch filtering,” ACM/IEEE International Symposium on Computer Architecture (ISCA) (2019) 1 (DOI: 10.1145/3307650.3322207).
[25] Y. Kwon, et al.: “A disaggregated memory system for deep learning,” IEEE Micro 39 (2019) 82 (DOI: 10.1109/MM.2019.2929165).
[26] D. Chatzopoulos, et al.: “Mobile augmented reality survey: from where we are to where we go,” IEEE Access 5 (2017) 6917 (DOI: 10.1109/ACCESS.2017.2698164).
[27] G. Pastuszak and A. Abramowski: “Algorithm and architecture design of the H.265/HEVC intra encoder,” IEEE Trans. Circuits Syst. Video Technol. 26 (2016) 210 (DOI: 10.1109/TCSVT.2015.2428571).
[28] S. Srinath, et al.: “Feedback directed prefetching: improving the performance and bandwidth-efficiency of hardware prefetchers,” IEEE International Symposium on High Performance Computer Architecture (HPCA) (2007) 63 (DOI: 10.1109/HPCA.2007.346185).
[29] T. Vogelsang: “Understanding the energy consumption of dynamic
random access memories,” IEEE/ACM International Symposium on Microarchitecture (2010) 363 (DOI: 10.1109/MICRO.2010.42).

[30] Rambus: “DRAM power model” (2016) http://www.rambus.com/energy.

[31] A. Patel, et al.: “MARSS: a full system simulator for multicore x86 CPUs,” IEEE/ACM Design Automation Conference (DAC) (2011) 1050 (DOI: 10.1145/2024724.2024954).

[32] P. Rosenfeld, et al.: “DRAMSim2: a cycle accurate memory system simulator,” IEEE Comput. Archit. Lett. 10 (2011) 16 (DOI: 10.1109/L-CA.2011.4).

[33] C.D. Spradling: “SPEC CPU2006 benchmark tools,” ACM SIGARCH Computer Architecture News 35 (2007) 130 (DOI: 10.1145/1241601.1241625).