Superoptimizing Memory Subsystems for Multiple Objectives

Joseph G. Wingbermuehle(✉), Ron K. Cytron, and Roger D. Chamberlain

Department of Computer Science and Engineering,
Washington University in St. Louis, St. Louis, USA
{wingbej,cytron,roger}@wustl.edu

Abstract. We consider the automatic determination of application-specific memory subsystems via superoptimization, with the goals of reducing memory access time and of minimizing writes. The latter goal is of concern for memories with limited write endurance. Our subsystems out-perform general-purpose memory subsystems in terms of performance, number of writes, or both.

1 Introduction

Due to the large disparity in performance between main memory and processor cores, large cache hierarchies are a necessary feature of modern computer systems. By exploiting locality in memory references, these cache hierarchies attempt to reduce the amount of time an application spends waiting on memory accesses. Although cache hierarchies are most common, one can extend this notion to a generalized on-chip memory subsystem, which is interposed between the computation elements and off-chip main memory.

For general-purpose computers, memory subsystems are designed to have good average-case performance across a large range of applications. However, due to the general-purpose nature of these memory subsystems, they may not be optimal for a particular application. Because of the potential performance benefit with a custom memory subsystem, we propose the use of memory subsystems tailored to a particular application. Such custom memory subsystems have been used for years for applications deployed on ASICs and FPGAs [18]. Further, it is conceivable that general-purpose computer systems may one day be equipped with a more configurable memory subsystem if the configurability provides enough of a performance advantage.

Although DRAM is the most popular choice for main memory in modern computer systems today, there are several disadvantages to DRAM technology (e.g., volatility and scaling), leading researchers to seek other technologies. Because DRAM is volatile, it can be power-hungry since it requires power just to retain information. This is particularly apparent when used in a setting with infrequent main memory accesses. However, when used in a setting with frequent memory accesses, the refresh requirement for a large DRAM can greatly reduce application performance [19]. There are significant challenges to scaling down

S. Hunold et al. (Eds.): Euro-Par 2015 Workshops, LNCS 9523, pp. 352–363, 2015.
DOI: 10.1007/978-3-319-27308-2_29

DRAM cells [13] since bits are stored as charge on a capacitor, which limits energy efficiency and performance.

Several alternative main memory technologies have been proposed, including PCM [12] and STT-RAM [11]. Although there are many possible main memory technologies that could be considered, a common theme for many proposed technologies is an aversion to writes. For PCM, there is a limited write endurance, making it beneficial to avoid writes to extend the lifetime of the device. Further, on PCM devices writes are slow and energy-hungry. For STT-RAM, although writes do not limit the lifetime of the device, writes are much slower than reads and consume more energy.

Because writes are often costly with respect to energy, performance, and endurance, here we seek to determine if it is possible to modify the memory subsystem to reduce the number of writes to main memory. We are particularly interested in the possibility of reducing the number of writes beyond what a memory subsystem optimized for performance would provide.

We are pursuing this investigation using superoptimization techniques which originated in the compiler literature. The concept of superoptimization was introduced with the goal of finding the smallest instruction sequence to implement a function [14]. This differs from traditional program optimization in that superoptimization attempts to find the best sequence at the expense of a potentially long search process rather than simply improving code. In a similar vein, we are interested in finding the best memory subsystem at the expense of a potentially long search process rather than a generic memory subsystem.

Previously, we have used superoptimization techniques to improve the execution times of applications [21,22]. In this paper, we expand the scope to include minimization of memory writes. We will show how the superoptimization technique is beneficial in terms of write minimization, and we will also show how one can use superoptimization in a multi-objective context.

To evaluate our superoptimized memory designs, we target an ASIC with an external LPDDR main memory. The ASIC assumes a 45 nm process, clocked at 1 GHz, with 1 mm^2 available for the deployment of our custom memory subsystems. The external LPDDR is a 512 Mib device clocked at 400 MHz. All on-chip memory subsystems share access to the external LPDDR memory device.

2 Related Work

Superoptimization was originally introduced in [14]. In that work, exhaustive search was used to find the smallest sequence of instructions to implement a function. This is in contrast with traditional code optimization where pre-defined transformations are used in an attempt to improve performance. Note that traditional code optimization is not truly optimization in the classical sense, but instead simply code improvement. Superoptimization, on the other hand, does produce an optimal result when applied in this manner.

Since its introduction, superoptimization has been successfully used in compilers such as GCC, peephole optimizers [1], and binary translators [2].

General design space exploration has been applied to many fields, such as system-on-chip (SoC) communication architectures and integrated circuit design. Although a single objective, such as performance or energy, is often used, design space exploration for multiple objectives is also common [17]. Of particular interest to us is design space exploration applied to memory subsystems. Design space exploration has been used extensively to find optimal cache parameters [6,7]. This line of work has been extended to consider a cache and scratchpad together [3]. However, the ability to change completely the memory subsystem for a specific application and main memory subsystem distinguishes this work from previous work.

Many non-traditional memory subsystems have been proposed. These structures are often intended to be general-purpose in nature, but to take advantage of some aspect of application behavior that is common across many applications. However, there are also many non-traditional memory subsystems designed for particular applications, usually with much effort. Such designs are a common practice for applications deployed on FPGAs and ASICs [4].

Although performance is perhaps the most common objective, non-traditional memory subsystems optimized for other objectives have also been considered. For example, the filter cache [10] was introduced to reduce energy consumption with a modest performance penalty.

The combination of multiple memory subsystem components has also been considered to various degrees. For example, the combination of a scratchpad and cache has been considered [18]. Further, the combination of multiple caching techniques including split caches has been considered [15].

3 Method

The superoptimization of a memory subsystem involves several items. First we require a memory address trace from the application. This trace allows us to simulate the performance of the application with different memory subsystems. Next, we perform the superoptimization, which involves generating proposal memory subsystems and simulating them to determine their performance.

In order to evaluate the performance of a particular memory subsystem for an application, we use address traces. To gather the address traces, we use a modified version of the Valgrind [16] *lackey* tool. This allows us to obtain concise address traces for applications that contain only data accesses (reads, writes, and modifies). We ignore instruction accesses. All of the address traces contain virtual (instead of physical) addresses and are gathered for 32-bit versions of the benchmark applications.

To evaluate the performance of the memory subsystems proposed by the superoptimizer, we use a custom memory simulator. We use a custom simulator for three reasons. First, we need to simulate complex memory subsystems beyond simple caches. Second, rather than the number of cache misses, we are interested in total memory access time. Finally, the simulator must be fast enough to simulate large traces many thousands of repetitions in a reasonable amount of time.

Table 1. Memory subsystem components

Component	Description	Parameters ($n \in \mathbb{Z}_+$)
Cache	Parameterizable cache	Line size (2^n) Line count (2^n) Associativity ($1 \dots \mathit{line\_count}$) Replacement policy Write policy
FIFO	Stream buffer	Depth (2^n)
Offset	Address offset	Value ($\pm n$)
Prefetch	Stride prefetcher	Stride ($\pm n$)
Rotate	Rotate address transform	Value ($\pm n$)
Scratchpad	Scratchpad memory	Size (2^n)
Split	Split memory	Location (n)
XOR	XOR address transform	Value (n)

The memory subsystem superoptimizer is capable of simulating the memory subsystem components shown in Table 1. The CACTI tool [20] is used to determine latencies for ASIC targets.

For caches, the simulator supports four replacement policies: least-recently used (LRU), most-recently used (MRU), first-in first-out (FIFO), and pseudo-least-recently used (PLRU). The PLRU policy approximates the LRU policy by using a single *age* bit per cache way rather than $\lg n$ age bits, where n is the associativity of the cache.

The `offset`, `rotate`, and `xor` components in Table 1 are address transformations. The `offset` component adds the specified value to the address. The `rotate` component rotates the bits of the address that select the word left by the specified amount (the bits that select the byte within the word remain unchanged). Note that for a 32-bit address with a 4-byte word, $32 - \lg 4 = 30$ bits are used to select the word. Finally, the `xor` component inverts the selected bits of the address.

Other supported components include `prefetch` and `split`. The `prefetch` component performs an additional memory access after every memory read to do the prefetch. This additional access reads the word with the specified distance from the original word that was accessed. Finally, the `split` component divides memory accesses between two memory subsystems based on address: accesses with addresses above a threshold go to a separate memory subsystem from addresses below the threshold. Accesses that are not resolved within the split are sent to the next memory subsystem or main memory.

The communication between each of the memory components as well as the communication between the application and main memory is performed using 4-byte words. The bytes within the word are selected using a 4-bit mask to allow byte-addressing. The address bus is 30 bits, providing a 32-bit address space.

Table 2. Main memory parameters.

Parameter	Description	Value
Frequency	DRAM I/O frequency	400 MHz
CAS	Cycles to select a column	5
RCD	Cycles from open to access	5
RP	Cycles required for precharge	5
Page size	Size of a page in bytes	1024
Page count	Number of pages per bank	65536
Width	Channel width in bytes	8
Burst size	Number of columns per access	4
Page mode	Open or closed page mode	open
DDR	Double data rate	true

For the results presented here, the main memory is assumed to be a DRAM device. We use a DDR3-800D memory, whose properties are shown in Table 2.

To guide the optimization process, we use a variant of *threshold acceptance* [5] called *old bachelor acceptance* [9]. Old bachelor acceptance is a Markov-chain Monte-Carlo (MCMC) stochastic hill-climbing technique. Old bachelor acceptance provides a compromise between search space exploration and hill climbing.

Using stochastic hill-climbing, one typically selects an initial state, $s_t = s_0$, and then generates a *proposal* state, s^*, in the neighborhood of the current state. The state is then either accepted, becoming s_{t+1}, or rejected. With threshold acceptance, the difference in cost between the current state, s_t, and the proposal state, s^*, is compared to a threshold, T_t, to determine if the proposal state should be accepted. Thus, we get the following expression for determining the next state:

$$s_{t+1} = \begin{cases} s^* & \text{if } c(s^*) < c(s_t) + T_t \\ s_t & \text{otherwise} \end{cases}$$

For our purposes, the state is a candidate memory subsystem and the cost function, $c(\cdot)$, is described below to reflect the multiple objectives used in the optimization process.

With threshold acceptance, the threshold is initialized to some relatively high value, $T_t = T_0$. The threshold is then lowered according a cooling schedule. The recommended schedule in [5] is $T_{t+1} = T_t - \Delta T_t$ where $\Delta \in (0, 1)$. Old bachelor acceptance generalizes this, allowing the threshold to be lowered when a state is accepted and raised when a state is not accepted. This allows the algorithm to escape areas of local optimality more easily. For our experiments, we used the following schedule:

$$T_{t+1} = \begin{cases} T_t - \Delta T_t & \text{if } c(s^*) < c(s_t) + T_t \\ T_t + \Delta T_t & \text{otherwise} \end{cases}$$

To reduce the time required for superoptimization, we employ two techniques to speed up the process. First, we memoize the results of each state evaluation so that when revisiting a state we do not need to simulate the memory trace again. The second improvement is that we allow multiple superoptimization processes to run simultaneously sharing results using a database, thereby allowing us to exploit multiple processor cores.

Our memory subsystem optimizer is capable of proposing candidate memory subsystems comprised of the structures shown in Table 1. These components can be combined in arbitrary ways leading to a huge search space limited only by the constraints. For the ASIC target, the constraint is the area as reported from the CACTI tool [20].

Given a state, s_t, we compute a proposal state s^* by performing one of the following actions:

1. Insert a new memory component to a random position,
2. Remove a memory component from a random position, or
3. Change a parameter of the memory component at a random position.

We showed, in [21], that the above neighborhood generation technique is ergodic. To ensure that any discovered memory subsystem is valid, we reject any memory subsystem that exceeds the constraints. However, there are other ways a memory subsystem may be invalid. First, because we support splitting between memory components by address, any address transformation occurring in a split must be inverted before leaving the split. To handle this, we always insert (or remove) both the transform and its inverse when inserting (or removing) an address transformation.

4 Benchmarks

We use a collection of four benchmarks from the MiBench benchmark suite [8] as well as two synthetic kernels for evaluation purposes. The MiBench suite contains benchmarks for the embedded space that target a variety of application areas. For some benchmarks, the MiBench suite contains large and small versions. We chose the large version in the interest of obtaining larger memory traces.

The locally developed synthetic kernels include a kernel that inserts and then removes items from a binary heap (`heap`) and a kernel that sorts an array of integers using the quicksort algorithm (`qsort`). Rather than implement an application to perform these operations and use Valgrind to capture the address trace, the addresses traces are generated directly during a simulation run, which allows us to avoid processing large trace files for the kernels.

Because we are superoptimizing the memory subsystem, the amount of memory accessed by each benchmark is important. If a particular benchmark accesses less memory than is available to the on-chip memory subsystem, then it should be possible to have all memory accesses occur in on-chip memory, though such a design may require clever address transformations.

For the 45 nm ASIC process with an area constraint of 1 mm^2, we can store a total of 379,392 bytes in a scratchpad according to our CACTI model. This means that both the `bitcount` and `dijkstra` benchmarks are small enough to be mapped into a scratchpad, but all of the remaining benchmarks access too much memory for their footprint to fit on chip.

5 Results

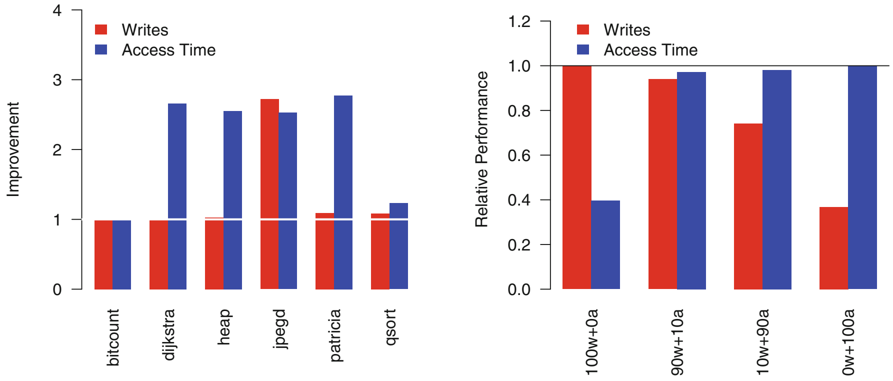
5.1 Minimizing Writes

Our previous work focused on reducing total memory access time [21]. However, the superoptimization technique is generic and, therefore, can be used to optimize for other objectives. Here we investigate minimizing the number of writes. The cost function used to guide the superoptimization process is the total writes to main memory for the complete execution of the benchmark application. Thus, we are not optimizing for performance, but exclusively for a reduction in writes to main memory.

To determine if the memory subsystem superoptimized for writes is actually any better at reducing writes than a memory subsystem superoptimized for total access time, we compare each of the memory subsystems. The first column of Fig. 1(a) shows the improvement that the memory subsystems superoptimized for writes have over the memory subsystems superoptimized for total access time (that is, W_t/W_w where W_w is the total number of writes to main memory when using the memory subsystem superoptimized for writes and W_t is the total number of writes to main memory when using the memory subsystem superoptimized for access time). The second column shows the improvement that the memory subsystems superoptimized for total access time have over the memory subsystems superoptimized for writes (that is, T_w/T_t, where T_t is the total access time when using the memory subsystem superoptimized for access time and T_w is the total access time when using the memory subsystem superoptimized for writes). Thus, bars above one indicate an advantage of one superoptimized memory subsystem over another.

Here we expect all bars to be at least one, indicating that the memory subsystem superoptimized for a particular objective is at least as good for that objective as a memory subsystem superoptimized for the other objective. Indeed, all bars in Fig. 1(a) are one or greater. In some cases, there is little or no difference between the superoptimized memory subsystems. For example, for the `bitcount` benchmark, both memory subsystems reduce the number of writes to zero and the memory subsystem superoptimized for total access time provides only a slight improvement in total access time over the memory subsystem superoptimized for writes. However, in most cases, a different memory subsystem is able to provide the best results for either objective.

The memory subsystem superoptimized for writes for the `bitcount` benchmark is shown in Fig. 2(a) and the memory subsystem superoptimized for total access time for the `bitcount` benchmark is shown in Fig. 2(b). For this benchmark, the memory subsystem superoptimized for total access time provides only

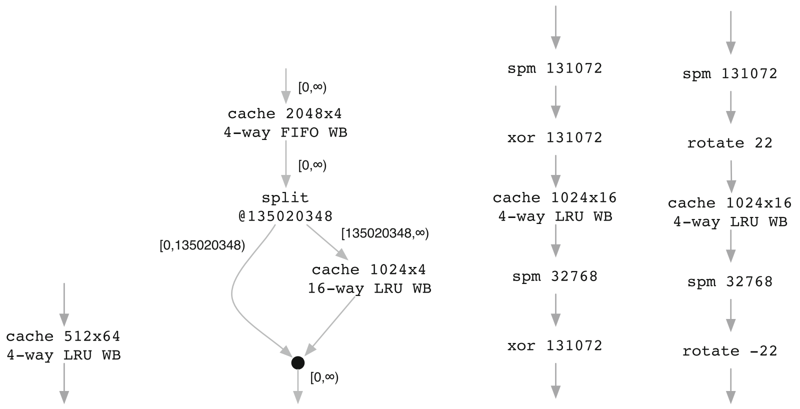


(a) (b)

Fig. 1. (a) Write and access time improvement, relative to one another; (b) Multi-objective superoptimization (in Sect. 5.2).

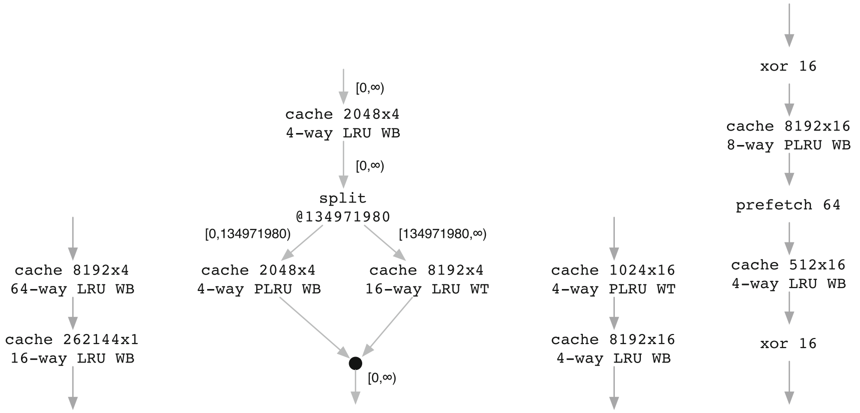
a small advantage over the simpler memory subsystem that was discovered to reduce writes to main memory.

The next memory subsystems we consider are those superoptimized for the heap kernel. The memory subsystem superoptimized to minimize writes is shown in Fig. 2(c) and the memory subsystem superoptimized to minimize total access time is shown in Fig. 2(d). Interestingly, these memory subsystems are very similar with the only difference being the address transformation. Despite the similar appearance, the each of the memory subsystems is able to provide a benefit over the other.



(a) bitcount writes (b) bitcount access time (c) heap writes (d) heap access time

Fig. 2. Superoptimized memory subsystems for bitcount and heap.



(a) `patricia` writes (b) `patricia` access time (c) `qsort` writes (d) `qsort` access time

Fig. 3. Superoptimized memory subsystems for `patricia` and `qsort`.

The memory subsystem superoptimized for writes for the `patricia` benchmark is shown in Fig. 3(a) and the subsystem superoptimized for total access time is shown in Fig. 3(b). An interesting observation is the large and highly-associative caches that are used when minimizing writes is the objective. These caches are effective at eliminating writes, but they are quite slow.

Finally, we consider the memory subsystems for the `qsort` benchmark. Figure 3(c) shows the memory subsystem superoptimized for writes and Fig. 3(d) shows the memory subsystem superoptimized for total access time. One notable difference between these subsystems is the presence of the prefetch component in the memory subsystem superoptimized for total access time.

Overall, memory subsystems superoptimized to minimize total access time appear to be capable of large reductions in total access time over memory subsystems superoptimized to minimize writes. On the other hand, while a memory subsystem superoptimized for writes is often able to reduce the number of writes compared to a memory subsystem superoptimized for total access time, the improvement is usually less pronounced.

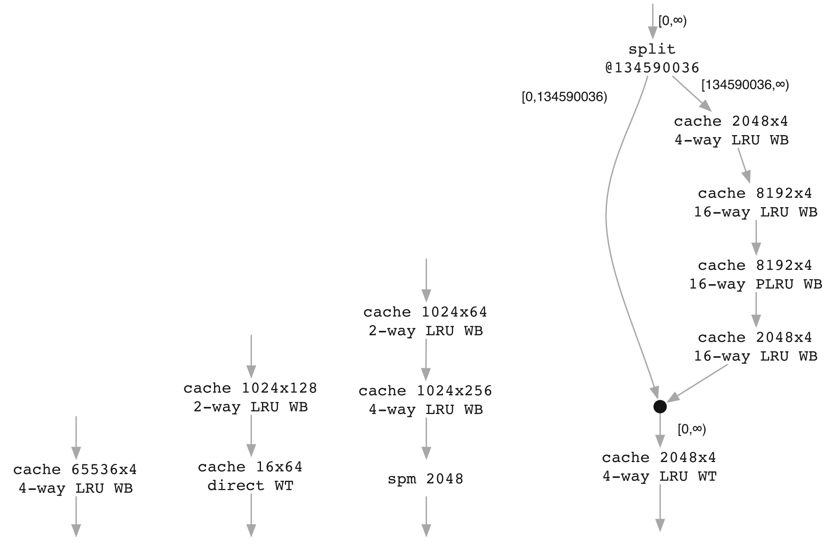
Another observation is that the memory subsystems superoptimized for writes are usually simpler than those superoptimized for total access time. Although a large cache will typically eliminate writes, the large cache will likely be slow. This implies that a large cache may be sufficient if we only care about writes, but something more exotic will likely provide better results if we want to minimize total access time.

5.2 Multi-objective Superoptimization

Here we investigate multi-objective superoptimization. From the previous section, we note that the memory subsystems that are superoptimized to minimize total access time are fairly good at reducing the number of writes to main memory, however, the memory subsystems superoptimized to minimize writes usually do better. On the other hand, the memory subsystems that are superoptimized to minimize writes often perform poorly with respect to total access time. Thus, one might wonder if it is possible to optimize for both objectives.

We use the weighted sum method to combine the objective functions to minimize writes and total access time. Figure 1(b) shows the improvement possible for various objectives for the `jpegd` benchmark with objective weights ranging from 100 %-writes, 0 %-access time through 0 %-writes, 100 %-access time. The graph shows uses the performance relative to the best result for writes and total access time. For the bars on the left, the graph shows W_w/W_m, where W_m is the number of writes to the main memory when using the memory subsystem superoptimized for multiple objectives and W_w is the number of writes to the main memory when using the memory subsystem superoptimized to minimize writes. For the bars on the right, the graph shows T_t/T_m, where T_m is the total access time when using the memory subsystem superoptimized for multiple objectives and T_t is the total access time when using the memory subsystem superoptimized to minimize total access time. Thus, higher values (closer to 1) indicate better results.

As can be seen in the graph, the largest differences in how good the memory subsystems perform for each objective occur when only a single objective is considered. When multiple objectives are considered, although there is some



(a) $100w + 0a$ (b) $90w + 10a$ (c) $10w + 90a$ (d) $0w + 100a$

Fig. 4. Memory subsystems for `jpegd`.

difference in how good the memory subsystems are, the result is very close to the best for all mixtures.

Figure 4 shows the memory subsystems for each mixture. When minimizing writes is most important, we see that a simple cache suffices. However, when minimizing total access time is also important, the large cache is separated into two caches, which makes sense since smaller caches are faster. Finally, when writes are no longer considered, a very complex memory subsystem is discovered, which does little to minimize writes, but provides the lowest total access time of all the memory subsystems considered.

6 Conclusions

We have shown that it is possible to superoptimize memory subsystems for specific applications that out-perform a general-purpose memory subsystem in terms of performance, writes, or a mixture. Unlike previous work, the memory subsystems that our superoptimizer discovers can be arbitrarily complex and contain components other than simple caches. To superoptimize a memory subsystem, we use old bachelor acceptance, which is a form of threshold acceptance.

While many benchmarks do not see a substantial improvement in write optimization (relative to access time optimization), for the `jpegd` benchmark individually optimizing for either goal provides substantial gains when measured against that goal. In addition, it is possible to co-optimize for both objectives and realize a memory system that performs quite well for each measure.

References

1. Bansal, S., Aiken, A.: Automatic generation of peephole superoptimizers. In: ACM SIGPLAN Notices, vol. 41, pp. 394–403. ACM (2006)
2. Bansal, S., Aiken, A.: Binary translation using peephole superoptimizers. In: Proceedings of 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI), vol. 8, pp. 177–192, December 2008
3. Chen, Y.T., Cong, J., Reinman, G.: HC-Sim: a fast and exact L1 cache simulator with scratchpad memory co-simulation support. In: Proceedings of 9th International Conference on Hardware/Software Codesign and System Synthesis, pp. 295–304. IEEE (2011)
4. Choi, Y.k., Cong, J., Wu, D.: FPGA implementation of EM algorithm for 3D CT reconstruction. In: Proceedings of 22nd Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 157–160. IEEE (2014)
5. Dueck, G., Scheuer, T.: Threshold accepting: a general purpose optimization algorithm appearing superior to simulated annealing. J. Comput. Phys. **90**(1), 161–175 (1990)
6. Ghosh, A., Givargis, T.: Cache optimization for embedded processor cores: an analytical approach. ACM Trans. Des. Autom. Electron. Syst. **9**(4), 419–440 (2004)
7. Gordon-Ross, A., Vahid, F., Dutt, N.: Automatic tuning of two-level caches to embedded applications. In: Proceedings of the Conference on Design, Automation and Test in Europe, p. 10208 (2004)

8. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: MiBench: A free, commercially representative embedded benchmark suite. In: Proceedings of 4th International Workshop on Workload Characterization, pp. 3–14 (2001)
9. Hu, T.C., Kahng, A.B., Tsao, C.W.A.: Old bachelor acceptance: a new class of non-monotone threshold accepting methods. ORSA J. Comput. **7**(4), 417–425 (1995)
10. Kin, J., Gupta, M., Mangione-Smith, W.H.: The filter cache: an energy efficient memory structure. In: Proceedings of 30th ACM/IEEE International Symposium on Microarchitecture, pp. 184–193. IEEE (1997)
11. Kultursay, E., Kandemir, M., Sivasubramaniam, A., Mutlu, O.: Evaluating STT-RAM as an energy-efficient main memory alternative. In: Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software, pp. 256–267. IEEE (2013)
12. Lee, B.C., Ipek, E., Mutlu, O., Burger, D.: Architecting phase change memory as a scalable DRAM alternative. ACM SIGARCH Comput. Archit. News **37**(3), 2–13 (2009)
13. Mandelman, J.A., Dennard, R.H., Bronner, G.B., DeBrosse, J.K., Divakaruni, R., Li, Y., Radens, C.J.: Challenges and future directions for the scaling of dynamic random-access memory (DRAM). IBM J. Res. Dev. **46**(2.3), 187–212 (2002)
14. Massalin, H.: Superoptimizer: a look at the smallest program. In: Proceedings of 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 122–126 (1987)
15. Naz, A.: Split array and scalar data caches: a comprehensive study of data cache organization. Ph.D. thesis, Univ. of North Texas (2007)
16. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 89–100 (2007)
17. Palermo, G., Silvano, C., Zaccaria, V.: Discrete particle swarm optimization for multi-objective design space exploration. In: Proceedings of 11th Conference on Digital System Design Architectures, Methods and Tools, pp. 641–644. IEEE (2008)
18. Panda, R.P., Dutt, N.D., Nicolau, A., Catthoor, F., Vandecappelle, A., Brockmeyer, E., Kulkarni, C., De Greef, E.: Data memory organization and optimizations in application-specific systems. IEEE Des. Test Comput. **18**(3), 56–68 (2001)
19. Stuecheli, J., Kaseridis, D., Hunter, H.C., John, L.K.: Elastic refresh: Techniques to mitigate refresh penalties in high density memory. In: Proceedings of 43rd IEEE/ACM International Symposium on Microarchitecture, pp. 375–384 (2010)
20. Thoziyoor, S., Muralimanohar, N., Ahn, J.H., Jouppi, N.P.: CACTI 5.1. HP Laboratories 2, April 2008
21. Wingbermuehle, J.G., Cytron, R.K., Chamberlain, R.D.: Superoptimization of memory subsystems. In: Proceedings of 15th Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), pp. 145–154, June 2014
22. Wingbermuehle, J.G., Cytron, R.K., Chamberlain, R.D.: Superoptimized memory subsystems for streaming applications. In: Proceedings of 23rd ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp. 126–135, February 2015