Towards an Energy-Efficient Cache Architecture for Extreme-Scale Systems

Abdulrahman Alshegaifi, Chun-Hsi Huang
Department of Computer Science and Engineering
University of Connecticut
Storrs, CT 06226, USA
{abdulrahman.alshegaifi, chunhsi.huang}@uconn.edu

***Abstract*—Energy consumption is the major limitation to achieving exascale computing systems. Caches are essential components that dissipate a large proportion of processor's energy. The use of smaller caches can reduce this effect; however, smaller caches result in performance degradation because they increase the number of misses. One way to compromise these conflict goals is to find a group of frequent references that can be served from the smaller cache with negligible misses. This smaller cache can serve these frequent references without the need to access a relatively large cache.**

Stack references (references that access the stack memory region) may facilitate the use of smaller caches and, thus, we suggest using stack cache toward extreme scale systems. We propose non-unified data cache design that maintains stack data in a separate cache and aims to, at least, maintain the performance (in terms of hit rate) as the same as the conventional L1 data cache, but with efficient energy. Because extreme scale systems require a massive number of cores to deliver high performance, minimizing the L1 caches in each core increases the number of cores that can be allocated on a single die. Hence, we examined the performance of a non-unified data cache design in comparison to that of a conventional data cache for different sizes of L1 data caches ranging from a quite large to small size. Our results show that through all different sizes of data cache, the non-unified design improved or maintained the same performance of the conventional cache in all applications tested, except in a very few cases, typically when the data cache was relatively large.

Keywords—Cache Design; Cache Memories; Stack Cache; Exascale Computing Systems

I. INTRODUCTION

Exascale computing systems have been researched extensively in recent years. Exascale computing refers to systems that can execute a thousand times as many operations per second as those of current petascale systems. However, the construction of practical exascale systems is limited by several factors, such as energy efficiency, interconnection technology, scalability, and resilience. Energy efficiency is considered one of the greatest impediment to achieving exascale computing systems [1][2][3][4]. Future exascale systems are constrained by a power budget of 20 MW [2]; a representative current supercomputer consumes 17.8 MW [2]. To address this issue, the energy efficiencies of different aspects of the system should be considered.

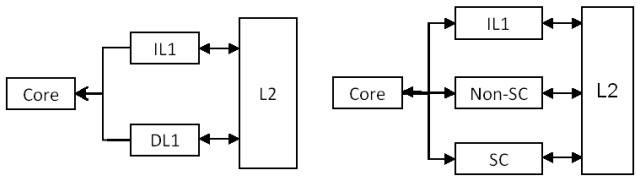
In a processor, power consumption is spread across multiple components, including caches, the clock, and the register. In particular, L1 caches are one major contributor to processor energy consumption. For example, caches account for 16% of the energy consumption in Alpha 21264 processors, and 30% in StrongARM processors [5]. Caches can dissipate more than 40% of the total energy consumed by a processor [6]. Therefore, several approaches have been proposed to alleviate this issue [7][8][9][10][11][12]. For extreme scale systems, the need for high performance and energy-efficient L1 cache design is increased. In this work, we focus on L1 data cache design.

L1 data cache is frequently accessed. The use of a single large cache consumes a greater amount of energy per access than the use of smaller caches. However, reducing the size of L1 data cache decreases its performance. To overcome this issue, one way is to find a group of frequent references that can be served from a smaller cache with negligible misses. This smaller cache can help to serve most of the references and, thus reduces the number of accesses to a large data cache. Thereby, it saves energy with efficient performance.

A promising approach is to take advantage of the unique characteristics of the stack memory region. The references to this memory segment seem to be frequent, and its data behavior differs from those of other memory regions; various researchers have attempted to exploit these properties, such as [13][14][15][16]. This region grows each time a function is called and shrinks when the result is returned. At any given time, only one stack frame is active. If the stack frame size is small, then a small specialized cache could provide a significantly high hit rate.

In this paper, we suggest the use of stack cache toward extreme scale systems and propose a non-unified data cache design that maintains stack data in a separate cache called a stack cache. Stack caches maintain only stack data that is retrieved from the stack memory region; other kinds of data, called non-stack data, are diverted to the original (non-stack) data cache. The aim of this study is to design L1 data cache that at, at least, maintains the performance as the same as the conventional L1 data cache, but with efficient energy.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE



(a) Unified Data Cache (b) Non-Unified Data Cache

Fig. 1. Cache Architecture

In the present study, we examine the influence of the stack cache on high-performance computing benchmarks and highly parallelizable applications, such as Divide-and-Conquer (D&C) algorithms. The D&C algorithms are naturally solved by a recursion function, implying that the stack memory region is frequently accessed. Subsequently, we evaluate the overall performance of the proposed non-unified data cache design in comparison to the original data cache architecture when the L1 data cache is minimized. The non-unified data cache architecture maintains stack and non-stack data in two separate specialized caches.

The remainder of this paper is organized as follows. Related work is discussed in section II. The proposed non-unified data cache architecture is detailed in section III. In section IV, our experimental setup is introduced, and in section V the non-unified design evaluation results are presented. Section VI contains our conclusions and the aims of future work.

II. Related Work

The unique characteristics of the memory stack region have generated significant recent research interest. Nielsen and Schoeberl [17] proposed four different stack cache implementations that allow the storage of stack and non-stack data in two separate caches, to improve the performance of embedded processors. These four implementations are known as the simple, window, and prefilling with and without tag stack caches. The simple and prefilling with tag stack caches were placed in parallel with the first-level data cache; the window and prefilling without stack caches were placed between the CPU and the DL1 data cache. The window and prefilling without tag stack caches improved the performance of the system by up to 3.5%.

Lee et al. [18] proposed a non-architectured register file, called the stack value file (SVF), that exploits the characteristics of the stack memory region to improve instruction-level parallelism and to reduce the latency of stack references, the demand on the first-level data cache, and memory traffic. The SVF is designed as a circular buffer and maintains data from the top of the stack. In addition, the SVF reduces data bus traffic by avoiding write-back of dirty blocks and the loading of invalid stack data located beyond the top of the stack.

Olson et al. investigated improving energy efficiency by exploiting stack data characteristics [19]. They found that stack segment accesses exhibited different behaviors to those of other memory accesses for a variety of different workloads for both x86 and ARM systems. To take advantage of these characteristics, the authors proposed implicit and explicit stack caches to reduce energy consumption. In the implicit stack cache, specific ways of the L1 data cache was reserved to store only stack data. In the explicit stack cache, a separate L1 cache was used to maintain stack data. Their results showed that the implicit stack cache reduced the dynamic energy of the L1 data cache by an average of 37%, and the explicit stack cache reduced the dynamic energy by an average of 36%.

Cho, Yew, and Lee [20] developed a data-decoupled architecture to provide high memory bandwidth in a wide-issue superscalar processor. This data-decoupled architecture splits all memory references into two streams. Each stream is diverted to a separate memory access queue in the pipeline and to a separate cache. They concluded that the data-decoupled scheme, in some cases, achieved higher performance than that obtained by increasing the number of ports to the L1 data cache. The authors also proposed fast data forwarding and combined access optimization. In their subsequent work [21], they proposed an access region predictor to predict which memory region (heap or stack) is accessed by a specific instruction. This enabled them to predict the relevant access region before the specific address was determined.

III. Non-Unified Data Cache Design

Non-unified data cache design is based on providing additional small caches dedicated to stack references, implemented in parallel to the first-level data cache (Fig. 1(b)). This design takes advantage of the unique characteristics of the stack segment to enable the same, or greater, hit rate of conventional data cache design but with less energy consumption. The conventional data cache design is called, here, unified data cache as shown in Fig. 1(a).

In our non-unified data cache design, the additional small cache is called the stack cache (SC). The non-stack cache (Non-SC) is the same as the conventional data cache, except that it only stores data that is retrieved from non-stack memory segments, such as the heap or bss. The SC implementation is equivalent to that of the data cache in a unified cache design. Thus, the SC can be organized as a direct-mapped, set associative, or fully associative cache.

Data memory accesses are classified as stack accesses if they occur within a certain region of the virtual memory space. For example, in an x86 processor, and the processor simulated in this study, the stack segment is located at the top of virtual memory and grows towards the lower addresses. If an address generated by a processor is located below the starting address of the stack segment, the *N* most significant bits of the starting address of the stack segment are compared with the address generated by the processor. If these bits match, then the access is classified as stack access; otherwise, it is classified as non-stack access. The number of bits (*N*) that are needed to classify memory accesses is 8 bits after experiments. In our non-unified cache architecture, all of the stack accesses are directed to the SC; the non-stack accesses are directed to the Non-SC. In the case of a hit, the data is supplied to the processor from either the SC or Non-SC, but never from both.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

TABLE I. BENCHMARKS

Benchmarks Suite	Benchmarks
Embedded Applications (MiBench)	bitcount (bc), susan.smoothing (ss), susan.edges (se), susan.corners(sc), patricia(pa), stringsearch (sts), rijndael.dec (rdec), rijndael.enc (renc), adpcm.rawcaudio (rc), adpcm.rawdaudio (rd), CRC32 (crc), fft, fft.inverse (fft.inv)
NAS Parallel Benchmarks (NPB)	cg, mg, ep, lu
Divand-Conquer Algorithms (D&C)	quicksort (qs), mergesort (ms), heapsort (hs)

In the case of a miss in either of the two caches, the data is fetched from the L2 cache or from lower memory in the hierarchy.

IV. EXPERIMENTS

A. Simulation Framework

To evaluate our non-unified data cache design, we integrated the SC using the SimpleScalar simulation toolset [24]. Because the performance of caches is measured by the hit rate factor, we used the sim-cache simulator. The SimpleScalar GCC compiler was used to generate SimpleScalar binaries in PISA format.

To ensure that the fair comparison was made between the non-unified and conventional cache architectures, a cacheline size of 32 B and a LRU replacement policy were used for each cache. The L1 data cache capacity and its associativity are introduced in later sections.

B. Benchmarks

Three different types of benchmarks were used in our experiments: MiBench [23], D&C algorithms, and NAS parallel supercomputer benchmarks (C version) [22]. Table I lists the selected applications from each benchmark suite. All of the benchmarks were run to completion.

MiBench is widely used for benchmarking embedded systems and includes a set of representative embedded application domains: Automotive and Industrial Control, Network, Security, Consumer Devices, Office Automation, and Telecommunications. We randomly selected applications from those domains. In addition, MiBench provides two input data sets: a small and a large set. We ran each benchmark using the large input data set.

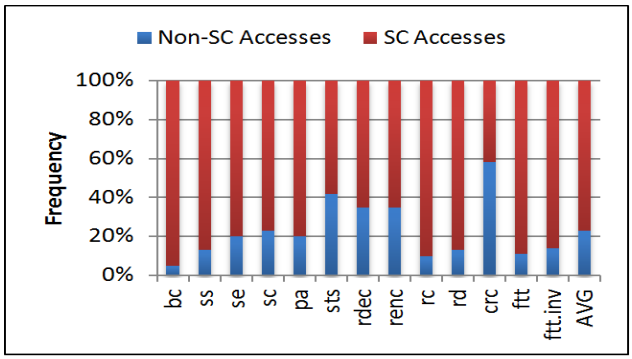
D&C algorithms, such as sorting algorithms, are naturally parallelizable algorithms. Therefore, they are the most suitable algorithms for parallel machines. They are naturally solved by recursion functions. This implies that the stack segment is heavily involved in the processing of these algorithms, which makes them appropriate for evaluating non-unified designs. We selected three different sorting algorithms to evaluate our SC-based design: Quick Sort, Merge Sort, and Heap Sort. Each of these algorithms was used to sort 1000, 16000, 128000, and 1280000 elements. We studied those sorting algorithms for both statically and dynamically allocated unsorted arrays. For statically allocated arrays, the unsorted array is defined as a static array inside a function; therefore, the compiler will allocate the array to the stack segment. Conversely, dynamically allocated arrays are defined as dynamic arrays by *malloc ()*; therefore, the array will be allocated to the heap segment.

The NAS Parallel Benchmarks (NPB) include several applications designed for parallel supercomputers. We selected four applications from this workload with a standard problem size (class A).

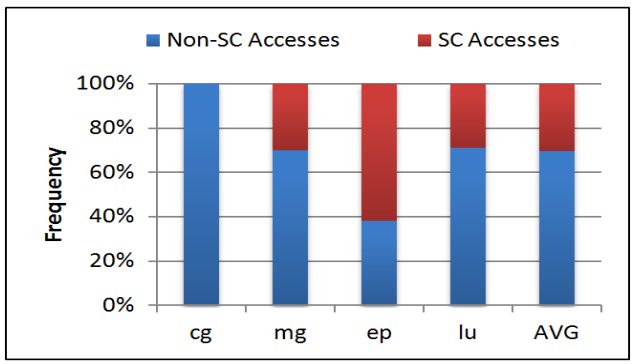
V. EVALUATION AND DISCUSSION

A. Distribution of Stack References

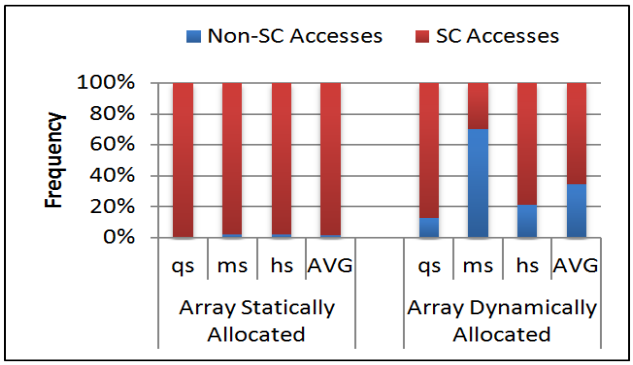
Fig. 2(a–c) show breakdowns of data memory accesses for different workloads. Fig. 2(a) represents applications from the



(a) MiBench Benchmarks



(b) NPB Benchmarks



(C) Divide-and-Conquer Algorithms

Fig. 2. Breakdown DL1 Memory References for Different Workloads

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

MiBench suite and shows that on average, 77% of data memory references were accessed from the stack memory region. For the NPB benchmarks, stack references were less frequent than for the MiBench workload. On average, 30% of the total NPB benchmark references were to the stack region; the maximum percentage of stack references was about 60% ("ep" application; Fig. 2(b)).

The stack references were extremely high for the D&C algorithms (Fig. 2(c)). As stated earlier, the D&C algorithms are solved using recursion functions; hence, the stack region is repeatedly accessed during each function call. Fig. 2(c) shows that on average, 65% of the total accesses were to stack memory when the array was dynamically allocated. The percentage of stack references was greater than 99% when the array was statically allocated. These mainly refer to program semantics. If the array is defined as a pointer (dynamically allocated), then, it will be allocated to the heap (non-stack). Conversely, if the array is defined as static inside a function (statically allocated), it will be allocated to the stack region.

Overall, these results show that the stack region is frequently accessed and represents most of the load/store references in the majority of applications.

B. Locality of Stack and Non-Stack Caches

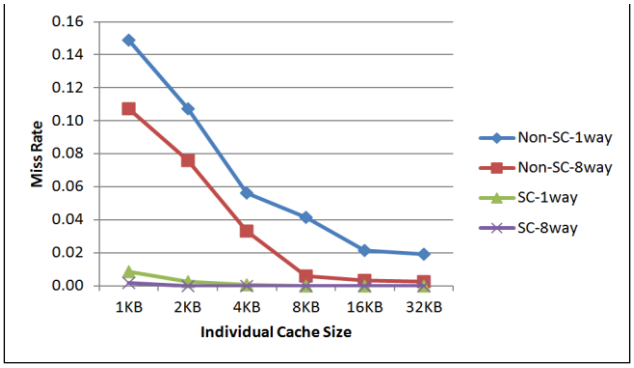
Several experiments were conducted to observe the locality of each individual cache (SC and Non-SC) in the non-unified data cache design. We measured the miss rate of the SC over its references, and that of Non-SCs with sizes of 1 to 32 KB with 1-way (direct-mapped) and 8-way associativity.

Because the majority of memory references represented by D&C algorithms are stack references, it would be unnecessary to compare the localities of the SC and Non-SC for this kind of application. Hence, we only performed locality experiments for MiBench and NPB workloads. Fig. 3(a) and (b) illustrate the average miss rate of the SC and Non-SC for the MiBench and NPB workloads, respectively. As expected, higher associativity resulted in a lower miss rate for each cache; this trend was more pronounced for the Non-SC. In addition, these results show that the SC consistently exhibited better locality than the Non-SC for all cache sizes tested in both benchmark sets. Moreover, regardless of the proportion of accesses, the SC achieved a hit rate greater than 99% with a relatively small cache. In contrast, the results for the Non-SC show that the miss rate decreased as the Non-SC size increased. This indicates that the localities of non-stack references are more sensitive to the cache size.

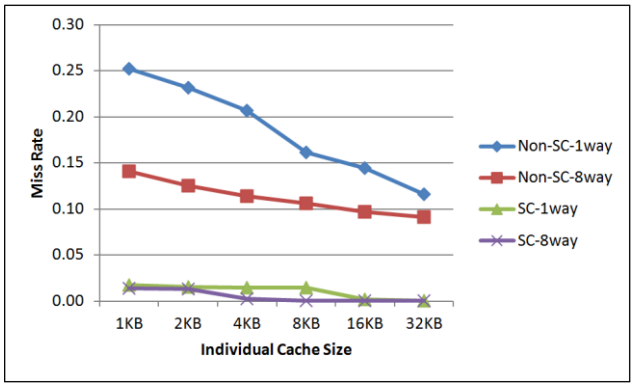
From these results, we conclude that the locality of the SC was significantly high, even with a very small cache size. Hence, by separating stack references from other types of references and directing them to an additional small cache, the SC would provide a efficient performance without significantly increasing the overall capacity of the first-level data cache.

C. Stack Cache Size

In our non-unified data cache design, the choice of a suitable size for the SC is critical because all of the stack



(a) MiBench Benchmarks



(b) NPB Benchmarks

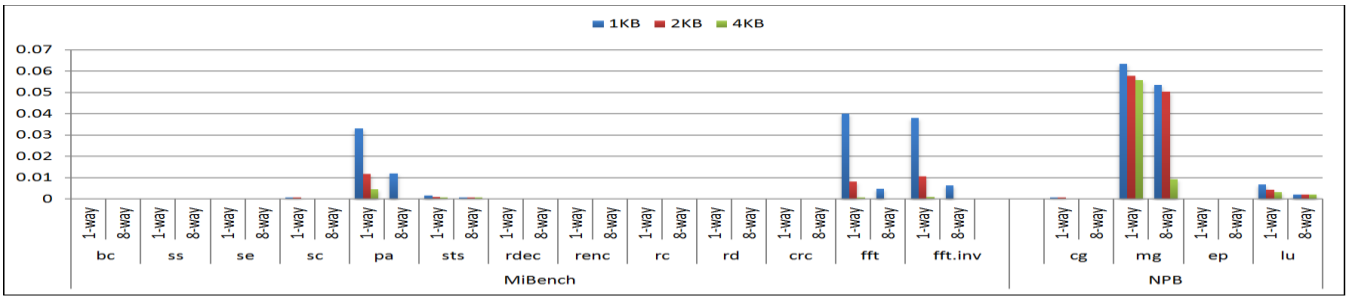
Fig. 3. Average Miss Rate of the Stack and Non-Stack Cache.

references will be restricted within that selected size. In addition, the SC should be as small as possible to maximize energy efficiency and to avoid significantly increasing the overall size of the L1 data cache.

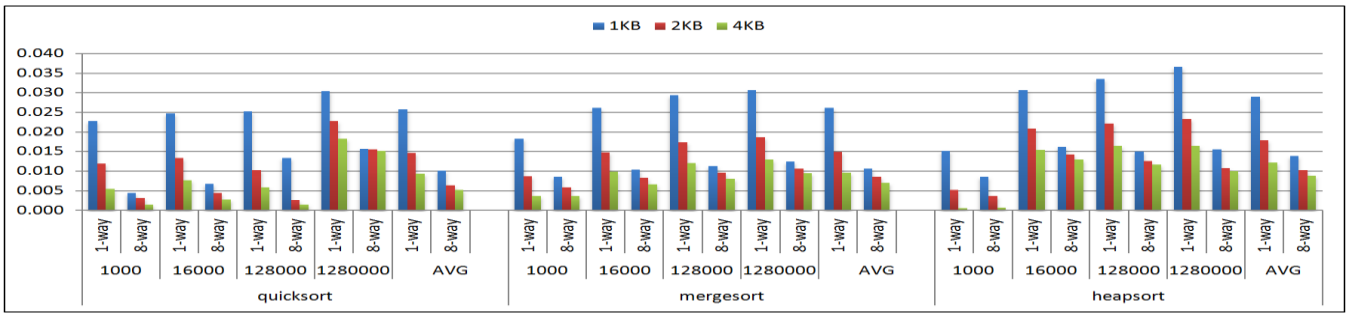
Fig. 4 shows the miss rates of the SC over its references for SC sizes from 1 to 4 KB with 1-way and 8-way associativity. Fig. 4(a) represents the SC miss rates for the MiBench and NPB benchmarks. As shown in Fig. 4(a), 1 KB was sufficient for the majority of the applications. However, in the "pa", "fft", "fft.inv", and "mg" applications, the 1 KB SC exhibited a significantly higher miss rate than either the 2 or 4 KB SC. For these applications, a 2 KB SC provided a hit rate of approximately 99% for 1-way associativity, except for the "mg" benchmark. In the case of 8-way associativity, 2 KB SC was significantly efficient and provided a hit rate of greater than 99.8%, except for the "mg" benchmark.

We also performed experiments on workloads that significantly use stack segments to carefully select the size of the SC. Fig. 4(b) and (c) show the miss rates of the SC for D&C algorithms for statically and dynamically allocated arrays, respectively. The size of the array was varied from 1000 elements to slightly over 1 million elements. For each array size, we measured the miss rates of SCs from 1 to 4 KB with 1-way and 8-way associativity. The SCs exhibited very low miss rates for both statically and dynamically allocated arrays. However, when the array was dynamically allocated, the SC miss rate was significantly lower than for the statically allocated array for all of the programs and array sizes tested.

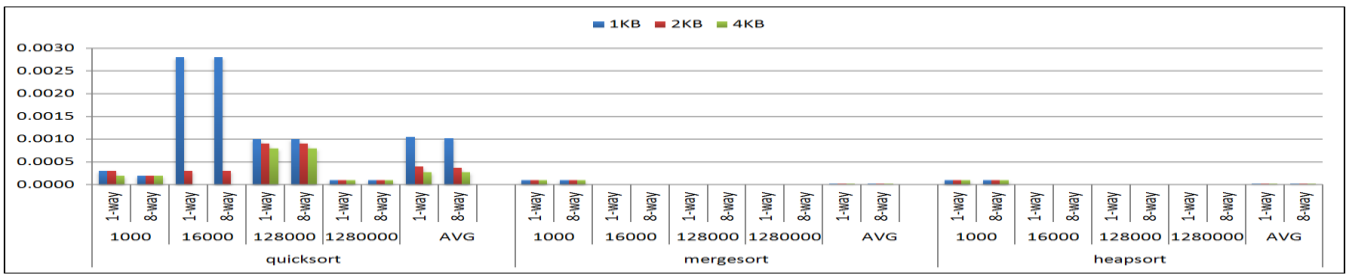
978-1-5386-3472-1/17/$31.00 ©2017 IEEE



(a) MiBench and NPB Benchmarks.



(b) Divide-and-Conquer When Array Statically Allocated.



(c) Divide-and-Conquer When Array Dynamically Allocated.

Fig. 4. Stack Cache Miss Rate for Various Sizes with 1-way and 8-way Associativity.

These results were attributed to the relative proportions of stack references. For statically allocated arrays, almost all of the references were to the stack region as shown in Fig. 2(c); however, in the dynamically allocated case, a larger proportion of the references were accessed from the non-stack region.

Based on these results, we selected an SC size of 2 KB. In most of the applications tested, the 2 KB SC provided a hit rate of greater than 99%. The 1 KB SC exhibited a significantly higher miss rate than the 2 and 4 KB SCs. The miss rate of the 2 KB was, also, larger than that of the 4 KB. However, in this case, the miss rate reduction was minor.

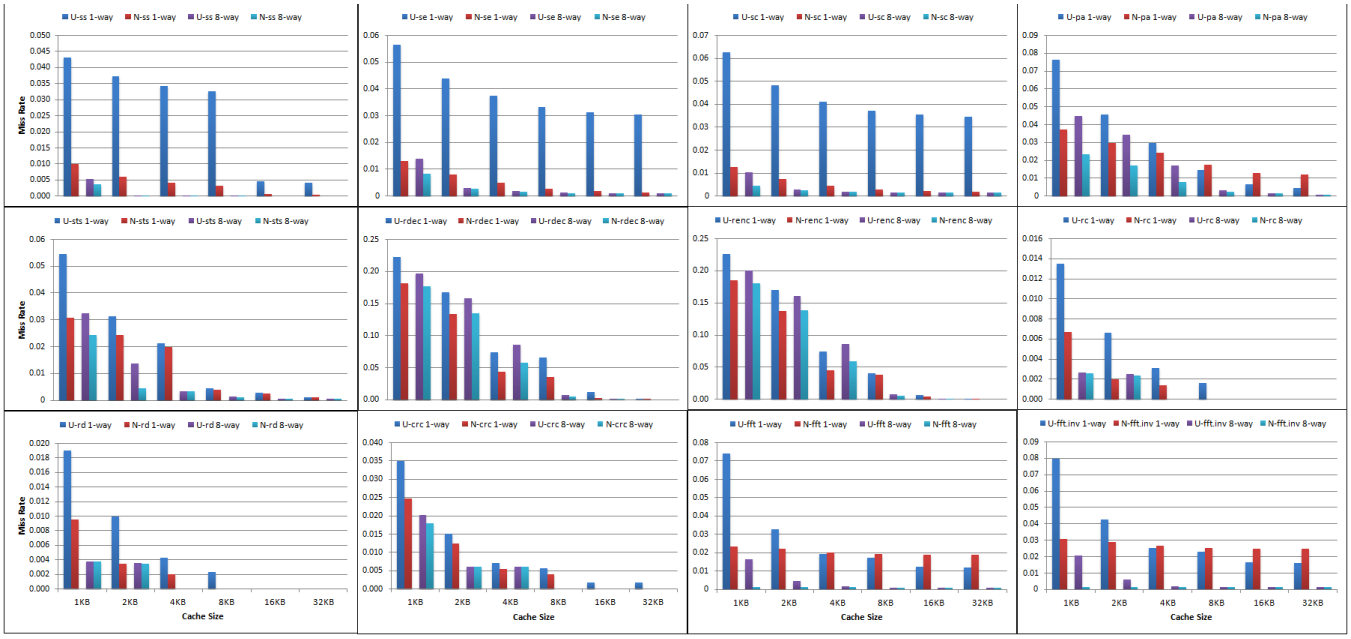
D. Non-Unified vs. Unified Data Cache Architecture

In this subsection, we discuss the miss rates of the non-unified data cache design compared to those of a conventional unified data cache architecture. The non-unified design adds a 2 KB SC to the first-level data cache. We examined the effect of this SC on the miss rate for data cache sizes from 1 to 32 KB with 1-way (direct-mapped) and 8-way associativity. All the caches had the same associativity in all conducted experiments. For example, when using a direct-mapped organization for the data cache, a direct-mapped organization was also applied to both the Non-SC and SC in the non-unified design.

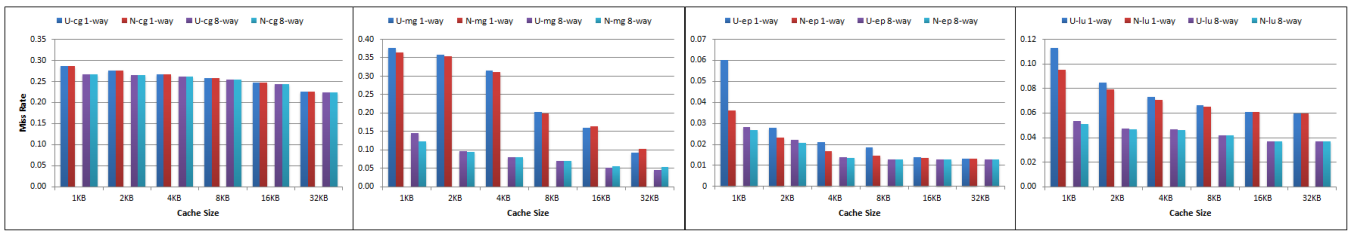
Because MiBench and NPB benchmarks exhibited more significant variations in their data memory reference patterns than D&C algorithms, we performed experiments using these benchmarks. Fig. 5(a) and (b) illustrate the miss rates of non-unified and unified architectures for different data cache sizes with a direct-mapped and 8-way associativity for MiBench and NPB workloads, respectively.

The results for the MiBench benchmarks show that for various data cache sizes, the non-unified design resulted in a significantly lower miss rate than the unified architecture for the majority of applications when using 1-way associativity (Fig. 5(a)). The “ss”, “se”, and “sc” benchmarks exhibited miss rate reductions of greater than 77%, for all of the cache sizes investigated, and the “rc” and “rd” benchmarks exhibited reductions of greater than 50% for cache sizes from 1 to 8 KB.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE



(a) MiBench Benchmarks



(b) NPB Benchmarks

Fig. 5. Unified VS Non-Unified Miss Rate for Different Data Cache sizes with 1-way and 8-way Associaitivity. U-ss and N-ss Indicate Unified and Non-unified Architecture for "ss" benchmark repectively.

In the case of larger associativity (8-way), we observed that the non-unified design only exhibited a reduced miss rate when the data cache was small. However, The miss rates of the non-unified architecture did not increase when larger cache sizes were used as illustrated in Fig. 5(a). The results for the "bc" application are not shown because 95% of its references are stack references, and a 1 KB SC provided a hit rate of almost 100% for both associativities (Fig. 4(a)). Hence, for applications such as "bc" and D&C algorithms, where stack references represent most of their accesses, a small cache is sufficient. Moreover, in these cases, a non-unified design would be significantly more energy efficient in contrast to a unified design that incorporates a single large cache for different needs of each applications.

For NPB workloads, the non-unified design generally exhibited the same miss rates as the unified design for both associativities and provided slightly better results for small cache sizes (Fig. 5(b)). The maximum miss rate reduction was about 40% for the "ep" benchmark, and 15% for the "mg" benchmark when using a 1 KB with 1-way and 8-way associativity, respectively. For the "cg" application, the miss rates were the same for both architectures for 1-way and 8-way associativity and all cache sizes; in this application, the most of the memory references are assigned to the non-stack region.

In very few cases, the non-unified design exhibited a slightly higher miss rate than the unified data cache architecture. For example, the "pa", "fft", and "fft.inv" applications from the MiBench workload in the case of 1-way associativity, and the "mg" application from the NPB workload for both associativities. However, these increased miss rates were observed when the data cache size (of the unified design) was relatively large. The increased miss rates were attributed to the limited size of the SC; and as the data cache size was increased, these applications were processed more effectively and the miss rate of the unified architecture decreased.

In conclusion, non-unified design consistently improves or maintains the same miss rate of unified data cache design on different data cache sizes with direct-mapped cache and 8-way associativity for the majority of applications of MiBench and NPB benchmarks.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

VI. Conclusions and Future Work

The goal of future exascale computing systems is to achieve 10^{18} operations per second with a similar, or lower, energy consumption than that of a current supercomputer. Caches are fundamental parts of computer systems, because they alleviate the speed disparity between the processor and main memory. However, existing L1 cache designs account for a large fraction of a chip's overall power consumption. Accessing a single large data cache consumes substantially more energy per access. We suggested a non-unified data cache architecture that maintains stack data in a small additional "stack cache".

We found that stack data are frequently referenced making them the best candidate to exploit. Our results showed that in high performance benchmarks, embedded benchmarks, and D&C algorithms, an average of 30%, 77%, and 99% of memory references, respectively, were stack references. In addition, for all of these different spectrums of stack references, small size of cache can provide a significant high hit rate. The experimental results showed that a 2 KB stack cache achieved a hit rate of greater than 99% for almost all of the benchmarks tested.

Moreover, we investigated the overall performance of a non-unified data cache design in comparison to a conventional unified data cache design for different sizes of L1 cache. We observed that for both a direct-mapped cache and one with 8-way associativity, the non-unified architecture attained the same or better performance for the majority of MiBench and NPB benchmarks with different data cache sizes.

Overall, the results obtained in this study demonstrate that the stack memory region is regularly referenced in most applications. A small stack cache was observed to consistently provide high hit rates for a wide range of applications. This implies that diverting stack references to a separate small cache is a promising approach to achieve a high-performance and energy-efficient design for L1 data cache. The next stage of this work is to further study the energy consumption of the proposed cache architecture.

References

[1] P. Kogge et al. "Exascale computing study: technology challenges in achieving exascale systems Technical," *DARPA IPTO*, 2008.

[2] DOE ASCAC Subcommittee, "Top ten exascale research challenges," U.S. Department of Energy (DOE), 2014.

[3] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *International Conference on High Performance Computing for Computational Science-VECPAR'10*, pp. 1-25, 2011.

[4] J. Torrellas, "Extreme-scale computer architecture energy efficiency from the ground up," in *Design, Automation and Test in Europe (DATE)*, pp. 1-5, 2014.

[5] S. Mittal, "A survey of architectural techniques for improving cache power efficiency," *Sustainable Computing.: Informatics and Systems*, vol. 4, no. 1, pp. 33-43, 2014.

[6] R. Gonzalez-Alberquilla, F. Castro, L. Pinuel, and F. Tirado, "Stack filter: reducing L1 data cache power consumption," *Journal of Systems Architecture*, vol. 56, no. 12, pp. 685-695, 2010.

[7] J. Kin, M. Gupta, and W. Mangione-Smith, "The filter cache: an energy efficient memory structure," in *30th International symposium on Microarchitecture (MICRO)*, pp. 184–193, 1997.

[8] P. Carazo, R. Apolloni, F. Castro, D. Chaver, L. Pinuel, and F. Tirado, "L1 data cache power reduction using a forwarding predictor," *Integrated Circuit and System Design, Power and Timing Modeling, Optimization, and Simulation*, pp. 116–125, 2011.

[9] J. Dai, M. Guan, and L. Wang, "Exploiting early tag access for reducing L1 data cache energy in embedded processors," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 22, no. 2, pp. 396-407, 2014.

[10] D. Nicolaescu, B. Salamat, A. Veidenbaum, and M. Valero, "Fast speculative address generation and way caching for reducing L1 data cache energy," in *International Conference on Computer Design (ICCD)*, pp. 101–107, 2006.

[11] A. Bardizbanyan, M. Själander, D. Whalley, P. Larsson-Edefors, "Speculative tag access for reduced energy dissipation in set-associative L1 data caches," in *International Conference on Computer Design (ICCD)*, pp. 302-308, 2013.

[12] A. Bardizbanyan, M. Själander, D. Whalley, P. Larsson-Edefors, "Reducing set-associative L1 data cache energy by early load data dependence detection (ELD^3)," *Proc. Conf. on Design, Automation and Test in Europe (DATE)* , 2014.

[13] S. Abbaspour, A. Jordan, and F. Brandner, "Lazy spilling for a time-predictable stack cache: implementation and analysis, " in *Int. Workshop on Worst-Case Execution Time Analysis*, pp. 83--92, 2014.

[14] J. Lu, K. Bai, and A. Shrivastava, "SSDM: Smart Stack Data Management for software managed multicores (SMMs)," *Proceedings of the Design Automation Conference*, pp. 1-8, 2013.

[15] A. Jordan, F. Brandner, and M. Schoeberl, "Static analysis of worst-case stack cache behavior," *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, pp. 55-64, 2013.

[16] S. Abbaspour, F. Brandner, and M. Schoeberl, "A time- predictable stack cache," 16th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC), pp. 1-8, 2013.

[17] C. Nielsen and M. Schoeberl, "Stack caching using split data caches," *Proceedings of IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, pp. 66-73, 2015.

[18] H. Lee, M. Smelyanskiy, C. Newburn, and G. Tyson, "Stack Value File: Custom microarchitecture for the stack," *Int'l Symposium on High-Performance Computer Architecture*, pp. 5-14, 2001.

[19] L. E. Olson, Y. Eckert, S. Manne, and M. D. Hill, "Revisiting stack caches for energy efficiency," Technical Report TR1813, University of Wisconsin, 2014.

[20] S. Cho, P. Yew, and G. Lee, "Decoupling local variable accesses in a wide-issue superscalar processor," *Int'l Symposium on Computer Architecture*, pp. 100-110, 1999.

[21] S. Cho, P. Yew, and G. Lee, "Access region locality for high-bandwidth processor memory system design," *Int'l Symposium on Microarchitecture*, pp. 136-146, 1999.

[22] An unofficial C version of the NAS Parallel Benchmarks OpenMP 3.0. 2014. Retrieved from: https://github.com/benchmark-subsetting/NPB3.0-omp-C

[23] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge and R. B. Brown, " MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of IEEE 4th Annual Workshop on Workload Characterization*, pp. 3-14, 2001.

[24] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," Technical Report TR1342, University of Wisconsin-Madison Computer Sciences Department, 1997.

978-1-5386-3472-1/17/$31.00 ©2017 IEEE