World Academy of Science, Engineering and Technology
International Journal of Electrical and Computer Engineering
Vol:10, No:4, 2016

Impact of Stack Caches: Locality Awareness and Cost Effectiveness

Abdulrahman K. Alshegaifi, Chun-Hsi Huang

***Abstract*—**Treating data based on its location in memory has received much attention in recent years due to its different properties, which offer important aspects for cache utilization. Stack data and non-stack data may interfere with each other's locality in the data cache. One of the important aspects of stack data is that it has high spatial and temporal locality. In this work, we simulate non-unified cache design that split data cache into stack and non-stack caches in order to maintain stack data and non-stack data separate in different caches. We observe that the overall hit rate of non-unified cache design is sensitive to the size of non-stack cache. Then, we investigate the appropriate size and associativity for stack cache to achieve high hit ratio especially when over 99% of accesses are directed to stack cache. The result shows that on average more than 99% of stack cache accuracy is achieved by using 2KB of capacity and 1-way associativity. Further, we analyze the improvement in hit rate when adding small, fixed, size of stack cache at level1 to unified cache architecture. The result shows that the overall hit rate of unified cache design with adding 1KB of stack cache is improved by approximately, on average, 3.9% for Rijndael benchmark. The stack cache is simulated by using SimpleScalar toolset.

***Keywords*—**Hit rate, Locality of program, Stack cache, and Stack data.

I. Introduction

THE new era of computer systems requires performance optimization to handle the huge amount of data. A major current focus in improving the performance is in the area of using parallelism. The goal of parallelism is to reduce the execution time of a program by executing multiple tasks simultaneously.

The well-known divide-and-conquer scheme is considered as a powerful technique for designing efficient algorithms. It has been proven that divide-and-conquer is a useful paradigm for sequential as well as parallel algorithms [1], [2]. Divide-and-conquer is a problem-solving technique that divides a problem into smaller sub-problems and then solves these sub-problems. Once the sub-problems have been solved, merging their solutions is started to construct a solution to the original problem. The divide-and-conquer method is naturally solved by using a recursion function. Upon each function call, a new stack frame is located in specific region of memory called stack segment. Hence, the stack segment is significantly used during function calls.

Abdulrahman Alshegaifi is with the Department of Computer Science and Engineering, University of Connecticut, Storrs, CT 06269, USA (corresponding phone: 860-771-0955; fax: 860-830-6271; e-mail: abdulrahman.alshegaifi@uconn.edu).

Chun-His Huang is with the Department of Computer Science and Engineering, University of Connecticut, Storrs, CT 06269, USA (e-mail: chunhsi.huang@uconn.edu).

The stack segment is one region of program's virtual memory and it is used to store dynamic variables with other certain information when function is called. It is clear that stack data that is retrieved form stack segment has different characteristics than non-stack data that is received from other segments as observed previously in [3], [4], [5], [8]. The stack data occupies contiguous locations in memory and thus it has a clear special and temporal locality. The stack data tends to be for a short period of time compared to non-stack data. Thus, keeping it in a separate cache, such as stack cache would maintain non-stack data from effecting and hence would improve the performance.

Studying the behavior of the stack is vital for two important reasons. First, researchers have shown that the frequency of stack accesses is approximately 33%-60% of memory accesses [4], [5]. This percentage of stack accesses is a considerable amount and indicating a need for this segment to be optimized. Second, the stack segment is fairly often used in sorting algorithms especially parallel sorting algorithms. Parallel sorting algorithms are based on divide-and-conquer scheme, which requires involving function calls. Sorting is considered as a fundamental component of numerous applications in computer systems. For example, sorting is used in indexing method for organizing data recode in database systems. Since then, parallel sorting algorithms have generated considerable recent research interest [7], [9]. Sorting is a core competent of high performance computing. Implying the stack segment is mostly used in these kinds of applications.

Based on the above reasons, we investigate the stack segment in more details. The approach we have used in this study aims to analyze the characteristics of stack data in terms of its effectiveness on other non-stack data. Also, this work aims to investigate the appropriate size and associativity to accommodate stack data, especially if the percentage of stack references is over 99% of memory references. In addition, we evaluate the cost effectiveness of adding one more cache at level 1 cache. By adding one more cache, there will, then, be three level-one caches, namely, the two common instruction and data caches, as well as the new stack cache which primarily handles stack data.

The rest of this paper is organized as follows: Section II presents related work, followed by a section that provides a background and describes stack cache implementation. In Section IV, experimental setup is introduced and in Section V the experimental result is evaluated. Section VI concludes the paper and presents the future work.

II. Related Work

Stack cache has generated considerable recent research interest. The properties of data representing stack segment are different than other memory segments, leading researchers to take advantage of these properties.

Olson et al. investigated energy efficiency by exploiting stack data characteristics [5]. They found that stack segment accesses have different behavior than others memory accesses in terms of *footprint*, *frequency* and *ratio of load and store*. They observed that on average, 40% of memory references are to the stack for SPEC 2006 workload. The ratio of writes to reads for stack accesses was higher than non-stack accesses. Based on these characteristics, they proposed implicit stack cache and explicit stack cache to reduce energy consumption. In implicit stack cache, specific ways of L1 data cache was reserved for storing only stack data while in explicit stack cache, different L1 cache was used to store the stack data. The result showed that implicit stack cache method minimized the dynamic energy of L1 data cache by 37% and explicit stack cache minimized the dynamic energy by 36% on average.

Hemsath et al. proposed stack cache that acts as a window in the current stack frame and predicted the recent used data to be maintained in the stack cache [4]. The stack cache is implemented as a circular buffer. In their proposed stack cache implementation, there is a *stack cache unit* that observes the modification on the stack pointer. When the stack pointer is modified, the Top and Bottom pointer of the circular buffer are adjusted to keep track of the new stack frame; valid data. There is another unit is called *stack cache prefill/spill unit*. This unit predicts either the stack segment is going to grow or to shrink and, thus, it keeps the useful data most of the time in the stack cache. The result showed improvement in the speed up by 1% to 4% for some of their workload set.

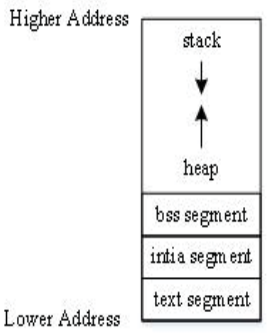
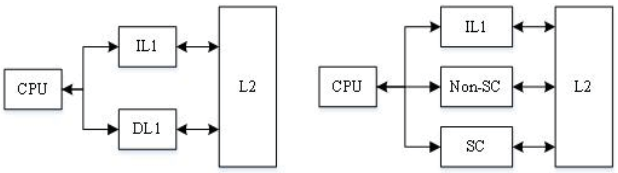


Fig. 1 Virtual memory space of a process



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

Fig. 2 Cache Architecture

Romansky and Lazarov proposed a new prefetching method to improve the locality and hence improve the performance [10]. Upon every procedure call, a procedure frame is reserved in memory. Their technique is to prefetch all or almost all of the stack data and caching them into stack cache memory. The result showed that 99% of accuracy was achieved by using their proposed stack cache memory.

TABLE I
SC and Non-SC Accuracy

Benchmarks	Non-SC Accesses	Non-SC Accuracy on Average			SC Accesses	SC Accuracy on Average		
		1 KB	2 KB	4 KB		1 KB	2 KB	4 KB
dijkstra	**93%**	0.898	0.927	0.966	**7%**	1	1	1
compr-ss95	**92%**	0.887	0.907	0.922	**8%**	1	1	1
sha	**14%**	0.948	0.948	0.991	**86%**	0.991	0.991	0.991
anagrm	**68%**	0.914	0.944	0.971	**32%**	1	1	1
perl	**70%**	0.810	0.860	0.907	**30%**	0.997	0.999	1
go	**67%**	0.607	0.733	0.847	**33%**	1	1	1
AES	**58%**	0.485	0.610	0.849	**42%**	1	1	1
gcc	**42%**	0.822	0.877	0.918	**58%**	0.981	0.991	0.996
	Avg	**0.796**	**0.851**	**0.921**	**Avg**	**0.996**	**0.998**	**0.998**

III. Background and Stack Cache Implementation

A program is a collection of instructions and data. When the program is lunched, it is loaded into memory via an operating system. Each program has its own virtual memory that is consists of several regions. The typical regions of memory include text segment, initialized data segment, uninitialized data segment (bss), heap and stack segment.

The stack segment, in x86 processor, is located at the top of virtual memory and grows towards the lower addresses as shown in Fig. 1. It is used to store local variables along with other certain information each time a procedure call is executed. Each segment of the virtual memory, stack or non-stack, has its own specific characteristics. In this paper, we investigated the characteristic of stack region because it shows that it has a predictable behavior.

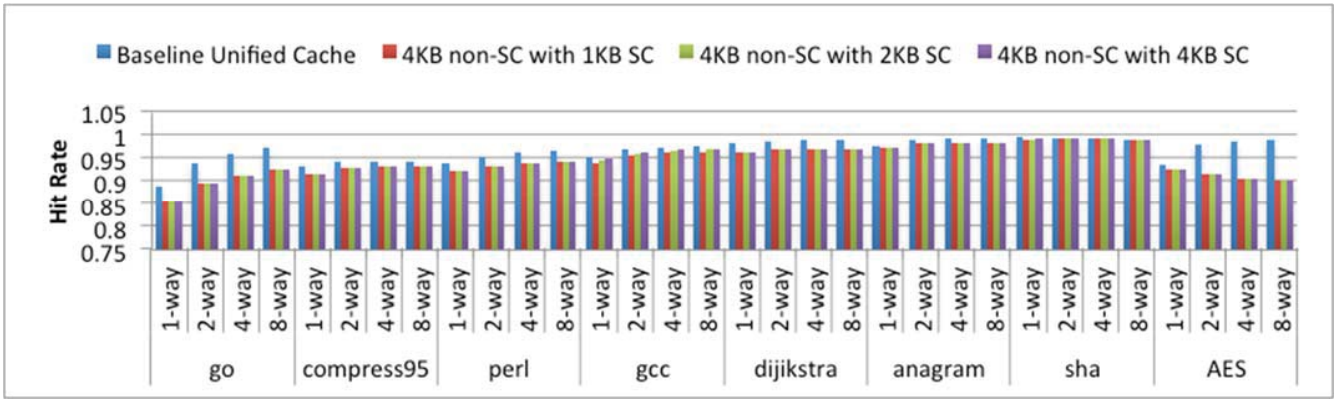


Fig. 3 Hit ratio of unified cache and non-unified cache architecture

For the purpose of analyzing the characteristics of stack data, we implemented stack cache as shown in Fig. 2 (b). The basic idea is to split the level 1 data cache as shown in Fig. 2 (a) into two caches; Stack Cache (SC) and non-Stack Cache (non-SC). The stack cache maintains, only, the data that is retrieved from the stack segment, whereas non-stack cache obtains the data that is retrieved from different memory segments, such as heap or bss segment. By using a separate cache for stack data, we would be able to study this part of memory (stack segment) in accurate manner.

The stack cache, in our approach, is implemented as the way of the data cache in the classical cache organization is implemented. So the stack cache could be organized as direct-mapped, set associative or fully associative cache.

All memory accesses are classified as stack accesses if they occur within a certain region of the virtual memory space. Simply, the N most significant bits of the top address of virtual memory that stack segment starts from are compared with the address generated by a processor. If they match, the access is classified as stack access; otherwise, classified as non-stack access. The number of bits (N) that are needed to classify memory accesses is 8 bits after experiments.

All stack accesses are directed to the stack cache and similarly, all non-stack accesses are directed to the non-stack cache. In case of hit, the data is supplied to the processor from either stack or no-stack cache; but never both. In case of miss in any of the two caches, the data is fetched from L2 cache or from the lower memory in the hierarchy.

IV. Experimental Setup

SimpleScalar toolset [6] was used to evaluate our approach. A modification was needed to simulate the stack cache for the non-unified cache architecture. The GCC compiler of SimpleScalar architecture was, also, used to generate simpleScalar benchmark binaries.

There were different workloads that we used in our experiments. Some of the workloads that we ran were selected from the SPECint95 sets. These benchmarks are gcc, go, anagram, compress95 and Perl. We used some other real applications such as Rijndael, Dijkstra and SHA algorithms. The Rijndael, also known as Advanced Encryption Standard (AES), is an encryption algorithm established by National Institute of Standard and Technology (NIST). The Dijkstra is an algorithm that is used to find the shortest route between two nodes in the graph. The Secure Hash Algorithm (SHA) is a cryptographic hash function and it is, also, published by NIST. Moreover, a quicksort workload was used. It is a well-known algorithm that uses divide-and-conquer technique for sorting. Since the quicksort is typically implemented by using a recursion functions, the stack segment is extremely involved. This variety of benchmarks that we analyzed would provide us more accurate observations for our analysis. Especially, our selected benchmarks cover almost all percentage of accesses spectrum distribution. In the result section, we provide a table classified the accesses of our benchmarks based on the percentage of accesses that are directed to either stack or non-stack cache.

In our baseline configuration for unified cache architecture, the level 1 data cache has capacity of 8KB, the size of cache line is 32B and the Least Recently Used (LRU) algorithm is used for replacement policy. For non-unified cache architecture, the size of both caches (stack and non-stack cache) is various while the size of cache line and the replacement policy remains the same as unified cache architecture.

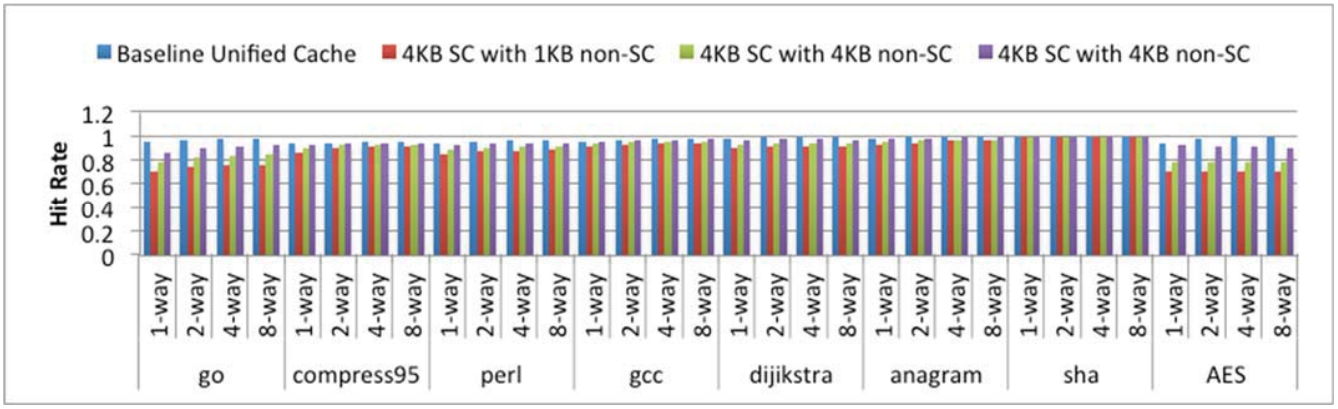


Fig. 4 Hit ratio of unified cache and non-unified cache architecture

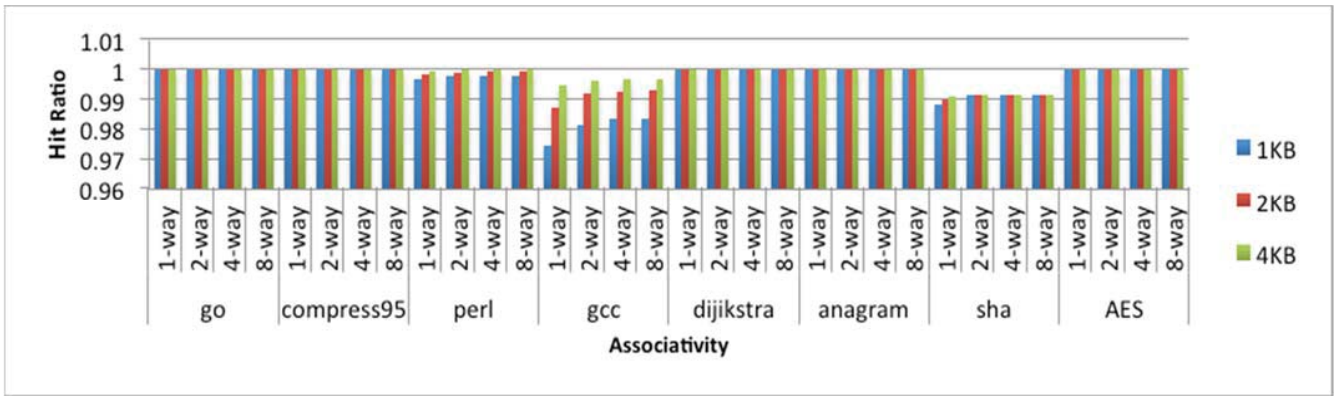


Fig. 5 Hit ratio of stack cache VS associativity

V. Results and Discussions

The total hit ratio of non-unified cache design is computed as follow:

First, the percentage of memory accesses that are directed to non-stack cache (β) is calculated by the following:

$$\beta = \frac{non\text{-}SC\text{-}Accesses}{non\text{-}SC\text{-}Accesses + SC\text{-}Accesses}$$

where *non-SC-Accesses* represent the total number of accesses that are directed to non-stack cache and *SC-Accesses* represents the total number of memory accesses that are directed to stack cache.

Then, the Total Hit Rate of non-unified cache architecture (THR) is computed by:

$$THR = \beta * Hit\ Rate(non\text{-}SC) + (1 - \beta) * Hit\ Rate(SC)$$

A. Unified Cache VS Non-Unified Cache Architecture

On both Figs. 3 and 4, we compared the hit ratio of the baseline of unified cache architecture with the total hit ratio of the baseline of non-unified cache architecture. For non-unified cache architecture, the size of two caches (stack and non-stack) varies from 1 KB to 4 KB and the total size of two caches is less than or equal 8 KB. It is limited to be not more than 8 KB for the simple reason that is to have a fair comparison between two caches architecture (unified and non-unified). The total size of both stack and non-stack caches in non-unified cache architecture is less than or equal to the size of level 1 data cache in unified cache architecture.

In Fig. 3, we fixed the size of non-stack cache to be 4KB and varies the size of the stack cache from 1KB to 4KB. We examined the total hit ratio of non-unified cache architecture for each size of stack cache. As can be observed form Fig. 3, splitting level 1 data cache into two caches (stack and non-stack cache) a slightly reduces the overall accuracy in almost all cases. This reduction on the accuracy is a consequence of an increase in non-stack cache misses due to a downsizing in the size of non-stack cache as illustrated in Table I. Also, Fig. 3 shows that the total hit rate of 4KB non-SC with size of 1KB, 2KB and 4KB of stack cache is identical in almost all benchmarks that we ran. Indicating the possibility of reducing the size of stack cache without sacrificing the accuracy.

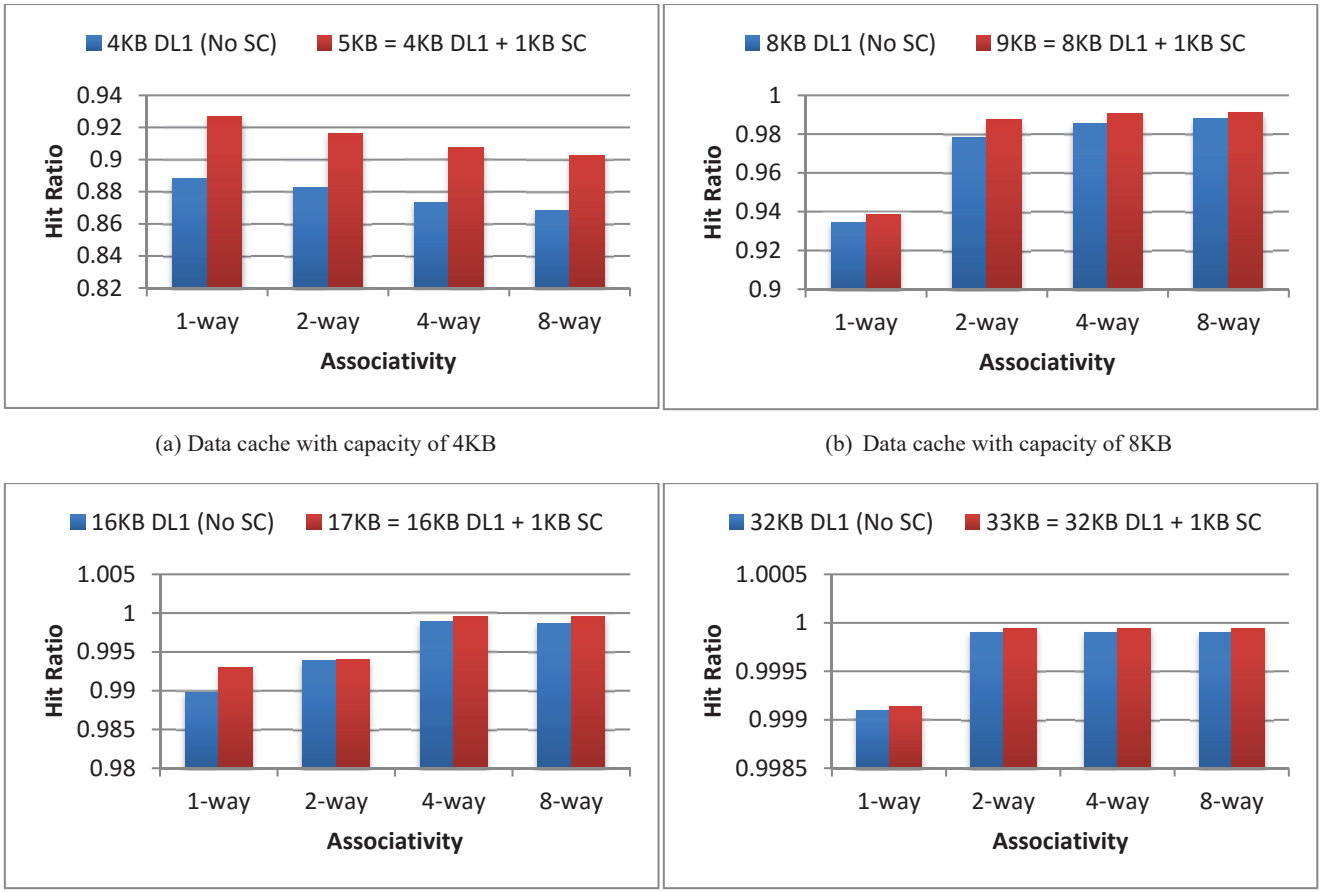
Lastly, Fig. 3 illustrates the effectiveness of the associativity on the hit ratio for both cache architectures. Generally speaking, increasing associativity would either increase the hit rate or maintain it the same for both cache design in 7 benchmarks except AES.

In Fig. 4, we examined the same aspects that are in Fig. 3. However, in this case we fixed the size of stack cache and change the size of non-stack cache instead. In overall, the unified cache design is still better in terms of the hit rate for almost all benchmarks than non-unified cache design. Fig. 4, also, shows that the hit rate of non-stack cache increases as its size increases. Implying that non-stack cache is more size

sensitive than stack cache. However, this is not the case of SHA algorithm. In case of SHA algorithm, increasing the size of non-stack cache does not help for obtaining higher hit rate and that is, clearly, a result of the percentage of accesses to non-stack cache is too small as listed in Table I. The percentage of accesses is around 14% of the total memory accesses in which turns out that the hit rate of non-stack cache of different sizes does not affect the overall hit rate of the non-unified cache.

TABLE II
QUICKSORT BENCHMARK

Number of Elements in an Array	Non-SC Accesses	Non-SC Accuracy on Average			SC Accesses	SC Accuracy on Average		
		1 KB	2 KB	4 KB		1 KB	2 KB	4 KB
1000	**9%**	0.972	0.972	0.972	**91%**	0.991	0.995	0.998
16000	**3%**	0.998	0.998	0.998	**97%**	0.989	0.993	0.996
128000	**1%**	1	1	1	**99%**	0.984	0.995	0.997
1280000	**0.06%**	1	1	1	**99.94%**	0.981	0.983	0.984
	Avg	**0.993**	**0.993**	**0.993**	**Avg**	**0.986**	**0.992**	**0.994**



(a) Data cache with capacity of 4KB

(b) Data cache with capacity of 8KB

(c) Data cache with capacity of 16KB

(d) Data cache with capacity of 32KB

Fig. 6 Hit ratio of unified cache VS a case of adding 1KB for stack cache

B. Appropriate Size and Associativity for Stack Cache

Table I represents the accuracy of different sizes of stack and non-stack cache. The benchmarks are divided into three categories based on the percentage of accesses that are directed to either stack or non-stack cache. For all benchmarks, we calculated the average hit rate of both caches (SC and non-SC) of different associativity that varies from 1-way to 8-way. For example, the hit rate of capacity 1KB of stack or non-stack cache is equal to the summation of hit rate of 1,2,4 and 8-way associativity divided by four.

The first category, in purple color, represents the benchmarks that have biased accesses to either stack or non-stack cache. As one can observe from that category, the first two benchmarks, which are dijikstra and compress95, have a majority of accesses to non-stack cache, while the last benchmark (sha) has the majority of accesses to stack cache. For the accuracy of non-stack cache in this first category, we see that increasing the size of non-stack cache a slightly improves the hit rate. However, for stack cache, the hit ratio is

significantly high and increasing its size doesn't really matter as much as to non-stack cache even though the majority of accesses are directed to stack cache as the case of *sha* benchmark.

The second category, in blue color, represents the benchmarks that have almost 70% of accesses to non-stack cache and approximately 30% of accesses to stack cache. We still observe almost the similar behavior of the first category. The non-stack cache is sensitive to the size while the stack cache is not.

The last category, in green color, represents a case of having approximately half of accesses to non-stack cache and the other half to stack cache. Our observations for the previous two categories remain true for this category as well.

We computed the average hit rate of each size (1KB, 2KB and 4KB) for all benchmarks of both stack and non-stack cache as reported in the last raw of Table I. Based on that computation, we would conclude that the optimal size for the stack cache that provides 99% of accuracy is 2KB as shown in Table I. However, For Non-stack cache, we expect to obtain a higher hit rate with the large size of cache.

Since we want to get more precise estimate of the optimal size of stack cache, we ran a quicksort benchmark that has extremely accesses to stack cache. We ran a quicksort on array of size 1000 elements, 16000 elements, 128000 elements and 1280000 elements. Table II shows the accuracy of different sizes of stack and non-stack cache when we ran the quicksort benchmark for different array sizes.

It is clearly seen from Table II that the majority of the accesses are to stack cache and the hit rate of stack cache is significantly high even with over 99% of accesses classified as stack accesses. Also, with this high percentage of accesses to stack cache, the size of 2KB of stack cache provides almost the same hit rate of size 4KB. We fairly could say that the difference between the hit rate of size 2KB and 4KB of stack cache is a negligible amount. This observation confirms our previous conclusion that the optimal size of stack cache that provides, on average, more than 99% of accuracy is 2KB.

Another source of evidence about our observation indicating 2KB is the efficient size of stack cache is shown in both Table I and Fig. 1. From Table I, we see the majority of accesses in case of sha algorithm are biased to stack cache. 86% of accesses are to stack cache while 14% of accesses are to non-stack cache. Hence, the hit rate of stack cache most likely determines the overall hit rate of non-unified cache architecture. From Fig. 1, in case of sha algorithm as well, it is evident from the result that the hit rate of non-unified cache architecture with size 2KB of stack cache is identical, in case of 2-way,4-way and 8-way, with the hit rate of size 8KB of the unified cache architecture. Implying that increasing the size of cache beyond 2KB for an application that has a majority of accesses to stack cache would not beneficial for getting higher hit rate.

Fig. 5 illustrates the hit rate of stack cache for different associativity. It is apparent that in the majority of benchmarks, direct-mapped (1-way associativity) is a suitable organization for stack cache and that is understandable since the stack data in some sense is located in contiguous locations in memory. For the case of perl and sha benchmarks, it seems that 2-way associativity is more adequate, however, the percentage improvement is negligible. In gcc benchmark, it is noticeably that increasing associativity beyond 1-way would improve the hit rate of stack cache. Nevertheless, we would conclude that 1-way is sufficient associativity for stack cache since it shows that increasing associativity more than 1-way is not beneficial for most of the benchmarks.

C. Cost Effectiveness of Adding Stack Cache at Level 1

In the previous sections, the data cache of unified cache architecture was split into two caches; stack and non-stack cache. The total size of two caches was limited to be not more than the size of data cache in unified-cache design. In this section, however, one more cache was added to the data cache instead. So it is similar to the non-unified cache architecture but in this case the size of stack cache is fixed and the total size of two caches is not limited to be the same size of level 1 data cache in unified-cache architecture. Since we have proven small size of stack cache would provide high accuracy, the cache added was small and acts as the stack cache. It maintains only the data that retrieved from the stack segment. In this experiment, only 1KB of stack cache was added and hence, the total capacity of level one cache is equal to the size of data cache plus 1KB of stack cache.

Since Rijndael (AES) benchmark shows the case of having almost 50% of memory accesses distributed among stack and non-stack segment, we decided to use Rijndael encryption algorithm as a case study for analyzing the cost effectiveness of adding small, fixed, size of stack cache to the data cache at level 1.

In Fig. 6, we compare the hit rate of level 1 data cache of unified cache architecture with the hit rate when adding 1KB of stack cache to the unified cache design. For different sizes of data cache varies from 4KB to 32KB, the result shows that small investment such as adding 1KB of stack cache improves the hit rate consistently.

From Fig. 6 (a), we see a noticeable improvement in hit rate when adding 1KB of stack cache at level 1. We think that it is often caused by the amount of data that needs to be processed is larger than the size of data cache. In this case, the improvement in hit ratio is about 3.9% on average of different associativity. This improvement we obtained strongly suggests that the investment of adding small size of stack cache to level 1 would considerably improve the hit rate. In case of data cache with capacity of 32KB, the cost of adding 1KB of stack cache is about 3%, which is very small. So, if the workload set is larger than the data cache of capacity 32KB, this small investment (3%) could be more beneficial for increasing the hit rate.

VI. Conclusion and Future Work

The stack segment is used to store dynamic variables with other certain information when function is called. In prior work, stack data and non-stack data are cached together; leading a possibility of an increase in conflict miss in data

cache. Treating stack data differently from non-stack data may prevent expulsion non-stack data from the data cache. In this study, we simulated non-unified cache architecture to keep stack data separate from non-stack data. Then, we examined the acceptable size and associativity for stack cache to achieve high hit ratio even with over 99% of memory accesses directed to stack cache. In addition, we analyzed the cost effectiveness of adding small size of stack cache at level 1 in unified cache design.

We observed that the overall hit rate of non-unified cache design was sensitive to the size of non-stack cache. Also, our results provide compelling evidence that the acceptable size and associativity for stack cache that provide, on average, more than 99% of accuracy is 2KB and 1-way associativity even with over 99% of memory accesses directed to stack cache. Furthermore, our result shows that adding 1KB of stack cache to unified cache design improves the overall hit rate by approximately, on average, 3.9% for Rijndael algorithm. A further observation about stack cache can be made for shared memory multiprocessor systems.

REFERENCES

[1] G. E. Blelloch and B. M. Maggs, “Parallel algorithms”, Algorithms and theory of computation handbook, Chapman & Hall/CRC, pp. 25–25, 2010.
[2] E. Horowitz and A. Zorat, “Divide-and-conquer for parallel processing”, IEEE Transactions on Computer, vol. C-32, no. 6, pp. 582–585, 1983.
[3] L. K. Melhus, “Analyzing contextual bias of program execution on modern CPUs”, M.S. Thesis, Norwegian University of Science and Technology, 2013
[4] A. Hemsath, R. Morton and J. Sjodin, “Implementing a stack cache”, Rice University, Advanced Microprocessor Architecture, http://www.owlnet.rice.edu/~elec525/projects/SCreport.pdf (2002).
[5] L. E. Olson, Y. Eckert, S. Manne and M. D. Hill, “Revisiting stack caches for energy efficiency”, University of Wisconsin, Technical Report. TR1813, 2014
[6] D. Burger and T. M. Austin, “The SimpleScalar tool set, version 2.0”, University of Wisconsin-Madison Computer Sciences Department, Technical Report. TR1342, 1997
[7] P. Kataria, "Parallel quicksort implementation using MPI and pthreads”, http://www.winlab.rutgers.edu/~pkataria/pdc.pdf (2008).
[8] M. A. Ertl, “Implementation of stack-based languages on register machines”, Ph.D. Thesis, Technische Universit\"at Wien, Austria, 1996
[9] M. F. Ionescu and K. E. Schauser. "Optimizing parallel bitonic sort", Parallel Processing Symposium, 1997. Proceedings., 11th International. IEEE, pp. 303-309, 1997.
[10] R. Romansky and Y. Lazarov, “Stack cache memory”, Journal of Information, Control and Management Systems, vol. 1, no. 2, pp. 29–37, 2003.