Data-Space Relocation for High Data Cache Performance in MT-Sync Embedded Multi-Threaded Processor

Mahanama Wickramasinghe Hui Guo

School of Computer Science and Engineering
University of New South Wales, Australia
{mahanamaw,huig}@cse.unsw.edu.au

Technical Report
UNSW-CSE-TR-201612
August 2016

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

Multi-threaded processor execution is a design strategy for performance improvement and energy reduction. With multi-threaded execution, the processor pipeline's idle time of one thread execution can be hidden by executing other threads so that the overall execution time (hence the energy consumption) can be reduced. One typical issue with the multi-threaded processor design is the cache. Cache reduces long and power consuming memory accesses, and has become an essential component in modern processor systems. However, multi-threaded execution can interfere the cache access behaviour, potentially causing more cache misses, leading to degraded cache performance. This work proposes an off-line thread data-space relocation approach for our MT-Sync multi-threaded processor design to reduce such data cache misses. The approach does not introduce any performance or hardware overhead to the existing processor. The experiment results on a set of applications show that our design achieves 17.5 times more performance gain compared to baseline multi-threaded execution and saves 2.5 times more energy.

1 Introduction

It is often the case that when executing applications, a processor will go through frequent pipeline stalls due to long off-chip memory accesses. Memory accesses consume considerable power and pipeline stalls reduce processor throughput, which together greatly degrade the overall system performance and energy efficiency. Multi-threaded execution is a widely used method to hide the pipeline stalls. With the multi-threaded execution, the processor idle time caused by a pipeline stall during one thread execution can be used to execute other threads so that the processor throughput is improved. Cache is an essential component in the processor system to reduce memory accesses. It sits between the processor and memory to avoid repeat memory accesses for a same memory data. However, when both multi-threading and caching are implemented in the system, the interference introduced to the cache access behaviour by thread interleaving can negatively affect processor performance.

To observe the impact of multi-threaded execution on the data cache performance, we ran some experiments. Table 1.1 shows the data cache miss frequency for five applications under two execution modes: single-threaded (ST) and multi-threaded (MT), both with a direct mapped data cache of the same size. The experiment shows that the cache misses are increased and in some cases, more than doubled due to the multi-threaded execution.

Table 1.1: Data cache miss frequency under single-threaded and multi-threaded executions

Application	AES	DCT	FFT	MI	RS
ST	61368	24052	5744	22156	6292
MT	68128	63405	8289	27769	17918

Many researches have been carried out in the recent literature on cache miss reduction in embedded systems design. A majority of them focuses on searching for an optimal cache configuration for the single thread execution. Apart from that, there is a considerable amount of work being done in the area of cache sharing aware thread scheduling algorithms.

In this work, we aim to improve the data cache utilization **for a given cache configuration for multi-threaded execution**. Our design targets an application that provides embarrassing parallelism and its execution can be forked into several independent threads each working on a separate data set. For such applications, a design to improve the instruction cache performance has been proposed in [23]. This design is named **MT-Sync** and it facilitates multi-threaded execution with synchronized loop execution to increase locality to reduce instruction cache misses. MT-Sync is a purely hardware level thread execution control mechanism that is entirely distinguishable from operating system (OS) level thread schedulers. It utilizes a set of application-specific synchronization points to control thread execution and use instruction pre-fetching and cache locking to effectively minimize the number of main memory accesses during the frequent loop execution. Moreover, since it is oblivious to the data cache utilization, there is space to further enhance the MT-Sync design with an improved data cache. This work aims at addressing this problem and we propose a thread data space relocation approach such that the negative impact

of the multi-threaded execution on the data cache is mitigated or even turned to positive. The main contributions of this work are as follows.

- We address the data cache issue in the multi-threaded execution that has been thus far rarely studied.
- We propose a novel data placement strategy, where the data for a thread is moved in the memory in a step of the cache block size. In such a way, the cache performance in the multi-threaded execution can be estimated based on the memory access traces of individual threads. This enables an off-line search for the solution.
- We present a search algorithm for a given number of threads to relocate the data space of each thread in such a way that the thread competitions for the cache are evenly distributed over all cache sets.

Our approach is simple yet effective. We evaluate our design with a set of applications, which shows an average of 10% performance and energy efficiency improvement due to the reduced data cache misses.

The rest of this report is organized as follows. Section 2 discusses the work on data placement strategies in the current literature. Section 3 explains our data placement strategy. Experiments and results are presented in Section 4 and Section 5 concludes the paper.

2 Related Work

There are numerous possible approaches to reduce data cache misses, such as cache customization [4], data prefetching [15] and cache sharing aware thread scheduling [19]. A significant part of such efforts has been devoted to fast exploration for an optimal cache configuration.

Many researchers looked into effectively assigning tasks into processor cores sharing caches in multiprocessor systems to reduce the data misses. Tam et al. [19] devised an operating system level thread scheduler for multiprocessors where data access latency depends on the physical location of the caches. The cores have exclusive L1 caches whereas an L2 cache is shared between the processors on the same chip. The processors on separate chips communication by sharing a memory. The proposed scheduling algorithm assigns tasks to the chips to reduce the cross-chip cache accesses. Heavily communicating threads are scheduled to the same chip. However, Zhang et al. [24] show that irrespective of how threads are placed on the cores, performance remains almost the same for PARSEC benchmark suite. Such scheduling strategies can be implemented on top of the memory data placement approach we propose in a larger system we propose as we focus on threads assigned to a particular embedded processor and we do not rely on a thread scheduler that focuses on the data cache.

Work on the application data placement in the memory to improve cache utilization is limited, and most of the work is performed at a fine-grained level and focused on how to place individual variables and data items in the memory for low cache misses. Such a problem, Petrank and Rawitz in [16] has theoretically demonstrated, is an NP problem and difficult to solve. Therefore, the existing approaches are basically heuristic. Below are some of them.

In an early work, Calder et al. [7] propose a software based data placement technique. They use profiling to determine the data usage patterns. Those patterns are then applied in a heuristic algorithm. The algorithm employs a Temporal Relationship Graph (TRG) to calculate data placement solutions distinctively for global variables, local variables, heap and constants. A data placement optimizer resides in the compiler reorders the global data segment. For heap data optimization, customized data allocation routines are used at run-time.

Later, the focus is turned to reorganization of heterogeneous data structures in the memory for high data cache locality. Truong et al. [22] propose to assemble the fields with high spacial locality into a single cache line within a structure. And the identical fields in different instances of the same data structure are dynamically grouped by a software-level data allocation tool to prevent the fields of low usage from being loaded into the cache when a high usage data is cached. Chilimbi et al. [8] use the clustering and colouring techniques to reduce the conflict misses in the data cache for the data with tree structures. Clustering improves temporal and spatial locality by packing closely accessed elements of data structures to a cache block. Colouring maps data elements accessed concurrently into non-conflicting cache locations. In [13] Kistler and Franz present a profile based optimization technique to group sequentially accessed fields of the data structures into the same cache line and reorder them within the cache line for further improvements. The implementation of the technique is aided by an operating system based dynamic code generation infrastructure [12]. In a similar study, Shin et al. [18] propose to dynamically allocate data structures to improve cache locality. With their approach, the same fields in all structures are bundled together and placed consecutively in the memory to reduce misses in L1 and L2 caches. In [17], Rabbah and Palem propose a linear time data remapping algorithm for data structures. The global data objects are remapped off-line according to the mismatch between their current placement and access patterns. The dynamically allocated data objects are remapped on-line by data allocation routines.

Some works aim to specific cache organizations. Beg and Beek [5][11] target direct-mapped cache and propose a graph-theory based [11] approach to identify optimal placement solution for a given memory access trace. Lin and Chen [14] similarly use a graph method to model the data placement problem for fully associative cache. They demonstrate that the problem can be reduced to a graph partitioning problem [9].

Tinnefeld et al. [21] and Ghoting et al. [10] discuss the cache conscious data placement in a data traversal perspective for database and data mining applications.

The above placement approaches either use a software level online data allocation routine (hence consuming processor time and degrading the overall system performance) or target on specific type of cache configurations (hence imposing application limitations), and they mainly deal with the placement of individual data objects of programs.

In this paper, we want to improve the data cache utilization for a multi-threaded processor. We treat the data space of a thread as a single large data (a coarse level approach) and we try to move the thread data spaces around in the memory to reduce the cache competitions caused by the multi-thread execution.

3 Data Space Relocation

We target a multi-threaded processor for a given application that is of embarrassing parallelism and offers the possibility for multiple independent-threads to be performed on different data sets. We assume the processor has separate instruction cache and data cache, and the data cache is shared by all threads. In [23], the authors proposed a design (called **MT-Sync**) that synchronizes thread executions on frequent loops (referred to as **sync-loop**). With this design, the execution of a sync loop will not be interrupted by other threads, and once the instructions of the loop are cached by one thread, they are available in the cache to other threads, hence improving the instruction cache performance. Here **we base our work on the MT-Sync design to reduce the cache misses caused by the multi-threaded execution**.

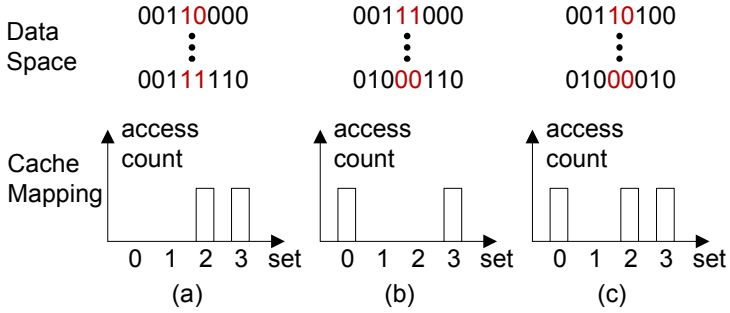

Figure 3.1: Example of Data Space Shift

We treat the data space for each thread as a whole. Our idea to reduce the cache misses caused by the interference of multi-threaded execution is relocating the data space for each thread. An example is given in Figure 3.1, where a data space is initially located in a 256-byte memory with the addresses from 00110000 to 00111110 (Figure 3.1(a)). For the direct mapped cache with block size of 4 words, each word of 2 bytes, its cache mapping is shown below the address space, namely, the data from the space can be cached in set 2 and set 3 in the cache. If the space is shifted by 1 block, to the location $00111000 - 01000110$, the cache mapping is changed to Figure 3.1(b). If the space is moved by 2 words, the cache mapping now becomes Figure 3.1(c).

As can be been, when data are moved in the memory, their cache locations can be different (as shown in both (b) and (c) in the example), and the distribution of the cache set access frequency may also be changed, as demonstrated in Figure 3.1(c) where the cache accesses are distributed over three sets (sets 0, 2, 3) instead of two sets with the initial space location.

Since we focus on the impact of the thread execution on cache, we want an individual thread data-space movement not to affect the distribution of the cache set access frequency of that thread. To this end, we propose to move the data space in the memory in a step that has the same size of a cache block.

For a given memory data placement and a cache configuration, if the frequently-accessed memory blocks map to only a small number of locations in the cache, the competitions for the cache among the threads, hence the cache miss rate, will be potentially high. We use the cache set access count (**AC**) to represent

such competitions. If a cache set has a high access count, chances that a cache block in the set will be evicted by other threads are very high.

For the MT-Sync loop execution, the cache accesses will not be interrupted by other threads. Therefore the related set accesses can be excluded in the consideration of cache competition. The rest of the accesses, we consider, are Subject to the Interference of Multi-thread execution (SIM) and we focus on the SIM access count (SIM AC) in our design. We use Figure 3.2(a) as an example. The shaded area represents the accesses from the sync-loop execution, they are removed in the SIM AC histogram, as shown in Figure 3.2(b).

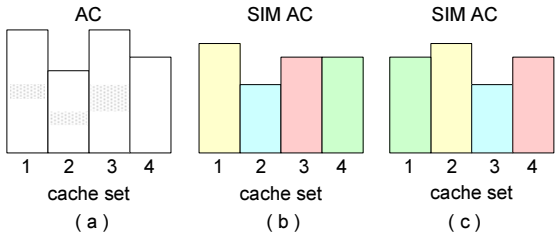

Figure 3.2: Set Access Count

We store the SIM access counts of all threads in a table, called **SIM AC table**. For m threads, there are m rows in the table. Each row lists the SIM AC for each cache set, as shown in Figure 3.3(a).

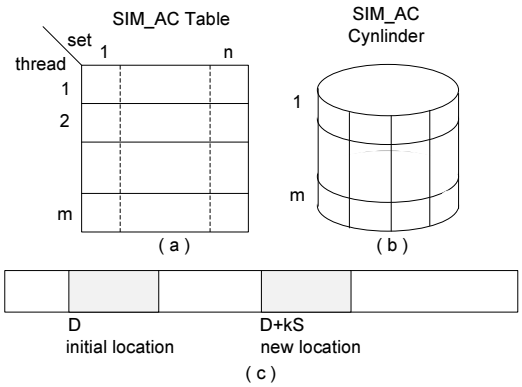

Figure 3.3: Thread SIM AC Representations (a) SIM AC Table (b) SIM AC Cylinder (c) Memory Data Relation

As has been mentioned above, we move the data space in the memory in (cache) blocks. Assume a cache block can hold S bytes. A thread data space, initially located at D, can be moved to a location, D+k*S, where k is a whole number, as illustrated in Figure 3.3(c). In other words, we move the data in their block addresses.

Assume the cache size is n sets, and the data block address in the memory is A, its set location in the cache can be determined by the mode operation: A%n. And for the data space movement, we have the following lemmas.

Lemma 1: If two locations A and B in a data space are mapped into the same cache set, after the space movement, they are still located in a same set.

Proof: According to the problem, A%n=B%n. Assume the data space is shifted by k blocks. Since k%n=k%n, based on the addition of the mode operation, we have (A+k)%n=(B+k)%n. Therefore, the two locations are still mapped to a same set in the cache after the data space shift.

Lemma 2: If memory locations A and B in a data space are not mapped into the same cache set, after the space movement, they still belong to different cache sets.

Proof: by contradiction. Assume they were mapped into a same cache set after the memory space shift (k blocks), namely, (A+k)%n =(B+k)%n. Since -k%n=-k%n, we have (A+k-k)%n=(B+k-k)%n, namely A%n=B%n, contradicting to the condition given in the problem (A%n!=B%n).

Based on Lemmas 1 and 2, we can conclude that for a thread, shifting its data space around will lead to its SIM AC distribution in the cache shifted and rotate-shifted due to the mode operation of memory address mapping. For example, if the data space is moved in the memory by one block, the set access count histogram of the thread is simply rotate-shifted by one set, as demonstrated in Figure 3.2(c) as compared to the initial SIM AC distribution in Figure 3.2(b).

When the data space is moved by k blocks in the memory, the related SIM AC histogram is rotate-shifted by k%n sets.

Therefore, for a single thread, the data space movement does not change the SIM AC distribution pattern; The total SIM access counts and cache misses of the thread remain the same. But for multi-threads, the access counts are accumulated, and moving their data spaces differently in the memory will change the total access count to each set. Hence the competition for a cache set by the multiple threads may be altered. Given this fact, we abstract our design problem as follows.

Since the SIM AC histogram is rotated over the cache sets when the data space is moved in the memory, we can bend the SIM AC table into a cylinder (we call it **SIM AC cylinder**). The cylinder contains m rings, each represent the access count for a thread over the cache sets. Moving the data space in the memory is equivalent to turning the ring. The total access count to a set by the m threads is sum of the related column values on the cylinder. We want to find a combination of the ring positions such that the column sums of the cylinder are as close as possible, namely, the total SIM accesses are evenly distributed over all sets in the cache – to reduce the cache competition, hence cache conflict misses.

We number the threads according to their relative data-space locations in the memory. The data space DS_i, of thread i has larger addresses than that of thread $i-1$. The shift distance of a data space is the offset of the new location to its initial location, as demonstrated in Figure 3.4. As can be seen, after the relocation, the total consecutive memory space size is increased. The gaps between the data spaces may cause the memory overhead if they cannot be used. Therefore, as a second search criteria, we want the total gap size as small as possible.

For the SIM AC cylinder of m threads and n-set cache, we position the rings in the order of their thread numbers. The top ring is associated with thread 1. We move rings for different thread data space placements based on the following rules:

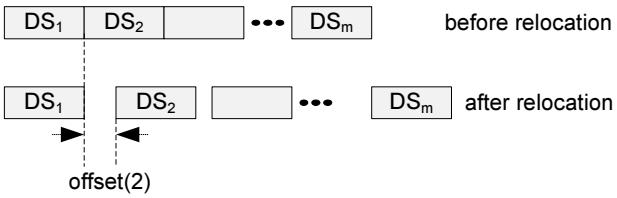

Figure 3.4: Thread Data Space Relocation

- For a thread, turn the ring rightwards to move its data space.
- For each position of a ring, i, all possible n positions of its lower ring, $i+1$, will be searched. When the last position of the low ring has been examined, the ring's position will be restored by turning it back to its initial position (the position with no gap between the data spaces of the two threads).
- For each upper ring movement, all rings below will be simultaneously turned the same distance to ensure the data spaces not overlapped.

Algorithm 1 Search for the best thread data space placement

```
/*for threads 2 to m, cache size: n sets*/
/*initialization*/
for i=2 to m do
   os(i)=-1; /*data space offset for thread i*/
   count(i)=0; /*number of offset positions searched for thread i*/
   gonext(i)=1; /*control turning ring i*/
end for
gonext(1)=0; /*control the completion of the space search*/
bst = Null; /*hold the best design after the search*/

/*design space search*/
while gonext(1)=0 do
   /*task 1: get a design*/
   /*get offset position for each thread – as the current design, D*/
   for i=2 to m do
      if gonext(i)=1 then
         /*moving rings j-m by one position*/
         for j=i to m do
            os(i)++;
         end for
         count(i)++;
         gonext(i)=0; /*current ring is halted to let the search of all positions
         of low rings*/
      end if
   end for
   /*task 2: save the current design, D, if it is the best so far*/
   if (dev(D)<dev(bst))OR
   ((dev(D)=dev(bst))AND(os(m)(D)<os(m)(bst))) then
      bst =D;
   end if
   /*task 3: prepare for the next round of search*/
   j=m;
   while count(j)=n do
      /*restore the position of ring j */
      count(j)=0;
      os(j) = os(j)-n;
      j–:
   end while
   /*ring j will be moved in the next round; If it is ring 1, the search is
   complete */
   gonext(j)=1;
end while
```

4 Experiments

We implement our data space relocation approach for the multi-thread processor with sync loop execution (MT-Sync) design discussed in [23]. The processor consists of a one-level instruction cache and a one-level data cache. The block size of both caches is 32 bytes.We adopt a similar experimental setting as proposed in [23]. Simplescalar [6] tool set is used for compiling and profiling and applications. Application Executions on the multi-thread processor are simulated with the Modelsim simulator [1] to obtain performance readings. Synopsys Design Compiler [2] is used to estimate the area and power costs of on-chip hardware components based on the TSMC Standard 65 nm Cell Library [3]. The critical path delay is used to determine the clock cycle time.

Designs with memories of different sizes are investigated. Table 4.1 shows the latency and dynamic energy consumption per memory access estimated by CACTI [20] for these memories. For the access latency (in ns), the equivalent number of CPU clock cycles (cc) are given in Columns 3 and 6 of the table. The instruction memory size ranges from 50 MB to 500 MB whereas the data memory size (available with CACTI) ranges from 50MB to 300MB. The memory access delays are incorporated into the processor hardware models for cycle accurate simulation.

Table 4.1: Latency and Dynamic Energy Consumption per Memory Access

Size(MB)	Instruction Memory			Data Memory		
	Access Latency		Energy	Access Latency		Energy
	(ns)	(cc)	(nJ)	(ns)	(cc)	(nJ)
50	9.19	5	0.89	10.21	5	0.52
100	11.98	6	1.10	15.74	8	0.74
150	14.30	7	1.50	20.74	10	1.06
200	17.59	8	1.54	25.17	12	0.81
250	19.75	9	1.81	29.59	14	0.97
300	21.94	10	2.13	33.95	16	1.15
350	24.82	12	1.61	-	-	-
400	27.01	13	1.76	-	-	-
450	29.23	14	1.92	-	-	-
500	31.42	15	2.08	-	-	-

In our experiment, we use ten kernel benchmarks as shown in Column 1 of Table 4.2. The abbreviations (Abr.) of the benchmark names, which will be used in the following result tables, are given in Column 2. The optimal instruction cache sizes are decided according to the frequent loop size of the benchmarks [23]. The data cache size is fixed to one sixteenth of the data space of each benchmark. Column 3 of Table 4.2 shows the code size (CS) of the benchmarks whereas Column 4 shows the selected instruction cache size (ICS). The selected data cache size (DCS) is given in Column 5.

The processor power and area costs significantly vary with its cache configurations. Since our focus is on the data cache, we limit our experiments to direct-mapped instruction caches. For data cache, we test direct-mapped (1-way), 2-way, and 4-way set associative caches.

Tables 4.3 and 4.4 respectively show the power consumption and area costs

Table 4.2: Benchmarks

Benchmark	Abr.	CS (KB)	ICS (B)	DCS (B)
Discrete cosine transform	DCT	1.09	128	512
Matrix multiplication	MM	0.39	128	512
Matrix inversion	MI	2.77	128	2048
LU matrix decomposition	LU	1.16	128	1024
Cholesky matrix decomposition	CHL	1.48	128	512
Gaussian elimination	GE	1.29	128	1024
Radix sorting	RS	2.77	128	256
Fast Fourier transform	FFT	1.80	256	512
Linear equation solving	LE	1.86	256	512
AES encryption	AES	7.48	512	2048

obtained by synthesis for the single-thread (ST) processor and baseline MT-Sync processor.

Table 4.3: Power Consumption and Area of the ST Processor with Different Cache Configurations

ICache (B)	DCache		Power (mW)	Area (mm^2)
	Size (B)	Assoc.		
512	512	1-way	9.40	0.2171
		2-way	9.43	0.2188
		4-way	9.44	0.2194
256	128	1-way	4.83	0.1508
		2-way	4.83	0.1513
		4-way	4.83	0.1513
128	512	1-way	6.87	0.1787
		2-way	6.90	0.1803
		4-way	6.91	0.1809
	256	1-way	4.92	0.1515
		2-way	4.93	0.1523
		4-way	4.94	0.1528
	128	1-way	4.00	0.1378
		2-way	4.01	0.1383
		4-way	4.01	0.1383
	64	1-way	3.48	0.1310
		2-way	3.48	0.1310

The execution data obtained from the simulation and the power/energy readings from the synthesis and memory models are used to calculate the overall energy consumption (E) of an execution based on the following formula:

$$E = P * T + E_i * a_i + E_d * a_d, \tag{4.1}$$

where P is the total power consumed by the on-chip components and T the application execution time. And E_i and E_d are the energy consumption per memory access respectively for instruction memory and data memory; a_i and a_d are the number of accesses made into the instruction and data memories.

Table 4.4: Power Consumption and Area of the MT-Sync Processor with Different Cache Configurations

ICache (B)	DCache		Power (mW)	Area (mm^2)
	Size (B)	Assoc.		
512	2048	1-way	23.28	0.4689
		2-way	23.36	0.4739
		4-way	23.36	0.4750
256	512	1-way	10.27	0.2802
		2-way	10.30	0.2808
		4-way	10.31	0.2827
128	2048	1-way	20.33	0.4293
		2-way	20.52	0.4343
		4-way	20.57	0.4373
	1024	1-way	12.85	0.3212
		2-way	11.96	0.3237
		4-way	12.92	0.3256
	512	1-way	9.18	0.2669
		2-way	9.21	0.2675
		4-way	9.22	0.2692
	256	1-way	7.23	0.2398
		2-way	7.24	0.2395
		4-way	7.25	0.2411

In this experiment, we first observe the impact of baseline multi-threaded execution on cache misses, performance and energy consumption. Then we show how MT-Sync with data space relocation mitigates negative effects or further improves the positive effects of multi-threading.

Given the 3 different data cache configurations, 10 different instruction memory sizes, and 6 data memory sizes, we run a total of 3*10*6*10=1800 experiments for 10 benchmarks for one execution type. Since we test ST, baseline MT and MT-Sync with data space relocation, the total number of experiments run is 1800*3=5400.

From these experiments we obtain data for cache misses and CPI of the executions and calculate the total energy consumption. For fairness, we compare these values of both baseline MT and MT-Sync with data space relocation with those values of ST execution. Obtained results are presented in two tables. Table 4.5 shows the average (avg) and standard deviation (dev) of the percentage reductions of data cache misses (Dmisses), instruction cache misses (Imisses), performance (CPI) and energy consumption (Energy) from baseline MT execution compared to ST execution. In a similar manner, Table 4.6 shows the same set of results for MT-Sync with data space relocation compared to ST execution. It should be noted that these values are obtained by considering all off-chip memory sizes and data cache configurations (60*3=180 values per application).

From Table 4.5 we can observe that baseline MT execution results in additional data misses for most applications except for AES, CHL and DCT and reduced instruction cache misses for all benchmarks except LU and MI. Moreover, it reduces performance and increases energy consumption on average for

a range of applications. These negative effects can be explained by the poor synergy between multi-threaded execution and caching.

The idea of MT-Sync with data space relocation is to mitigate these negative effects with respect to both data and instruction caches. From the results shown in Table 4.6, we can clearly observe that its every average reduction value is higher than its counterpart in Table 4.5. In most cases, the negative values have been turned to positive and the already positive values increased. On average, the negative impact on data cache misses is mitigated by a factor of about 9 whereas the instruction cache misses have been further reduced by a factor of 2. Similarly, average performance of ST have been increased 17.5 times more than MT by our design resulting in a 2.5 times more energy saving.

Table 4.5: Cache miss reduction, performance improvement and energy saving by baseline MT execution compared to ST

Benchmark	Dmisses (%)		Imisses (%)		CPI (%)		Energy (%)	
	avg	dev	avg	dev	avg	dev	avg	dev
AES	7.48	3.96	51.18	13.27	-10.97	15.38	16.07	12.65
CHL	4.64	2.65	95.89	2.45	-15.87	92.72	78.18	6.65
DCT	15.55	4.66	26.67	7.51	6.56	5.23	-79.34	32.41
FFT	-87.87	1.84	50.92	4.88	11.45	6.24	22.49	11.37
GE	-9.22	7.05	10.63	9.14	-29.11	12.40	40.43	7.13
LE	-25.70	7.71	44.71	15.22	28.30	9.04	46.10	9.11
LU	-4.60	2.66	-20.70	6.95	11.75	5.06	13.25	3.51
MI	-3.32	2.22	-26.56	6.49	-4.67	4.77	-41.79	5.06
MM	-6.86	2.18	25.25	6.46	-5.96	7.28	47.39	3.87
RS	-55.25	25.70	68.56	4.88	19.92	7.23	24.37	10.11
Average	-16.51	6.06	32.65	7.72	1.14	16.53	16.72	10.19

Table 4.6: Cache miss reduction, performance improvement and energy saving by MT-Sync with data-space relocation compared to ST

Benchmark	Dmisses (%)		Imisses (%)		CPI (%)		Energy (%)	
	avg	dev	avg	dev	avg	dev	avg	dev
AES	21.61	4.00	78.24	2.93	31.18	8.02	43.71	6.66
CHL	2.38	0.64	96.22	0.25	8.55	2.46	87.92	2.45
DCT	43.76	8.01	79.56	3.44	35.22	5.33	16.83	9.41
FFT	-86.11	1.59	51.06	4.43	15.40	5.33	25.94	11.46
GE	7.47	0.79	77.69	2.88	15.28	1.91	69.73	3.34
LE	11.52	4.38	48.02	11.59	28.65	7.69	59.37	6.60
LU	-4.48	1.25	-5.40	4.29	12.24	2.37	15.58	4.20
MI	-0.88	1.11	-21.69	6.30	3.56	7.08	-36.00	4.49
MM	6.28	1.68	84.70	0.70	24.56	4.87	80.74	2.63
RS	-22.41	13.13	76.93	4.02	25.67	8.57	42.45	6.27
Average	-2.09	3.66	56.53	4.08	20.03	5.36	40.63	5.75

As is discussed in Section 3, the data space relocation creates gaps between thread data spaces. Table 4.7 presents the average overhead in bytes for each

application and the relative value as compared to the application data space is given in the last column of the table.

Table 4.7: Memory space overhead from informed data placement

Benchmark	Data Cache Ways					
	1-way		2-way		4-way	
	bytes	%	bytes	%	bytes	%
AES	768	6.49	672	5.68	416	3.51
CHL	320	9.09	160	4.55	96	2.73
DCT	320	15.56	96	4.67	96	4.67
FFT	352	13.37	160	6.08	96	3.65
GE	576	13.00	384	8.66	192	4.33
LE	384	13.95	192	6.98	96	3.49
LU	384	4.84	160	2.02	96	1.21
MI	1536	16.28	544	5.77	96	1.02
MM	384	15.84	192	7.92	96	3.96
RS	160	11.90	96	7.14	32	2.38
Average	518.4	12.03	265.6	5.95	131.2	3.09

It can be seen from Table 4.7 that the percentage space overhead reduces as the number of ways of the cache increases. For a fixed cache size, the number of sets decreases with the increasing associativity. The less the number of sets, the less search space explored by Algorithm 1, which leads to a reduced data space shift range, hence the possible memory overhead.

5 Conclusions

Multi-threaded execution often adversely affects the cache performance.In this work we address this issue for the data cache of the MT-Sync multi-thread processor design we proposed in [23]. MT-Sync design is able to improve the instruction cache performance by synchronizing thread execution on frequent loops.

We presented a design time data-space relocation approach for each thread to reduce the data cache misses caused by the multi-threaded execution. Here we shift the thread data-space in the step of cache block size so that the interference of the multi-thread executions can be easily captured and the space relocation problem can be simplified.

The experiments on a set of applications and different data cache configurations demonstrated the effectiveness of our design approach. It mitigates the negative effects of multi-threaded execution even turning them into positive in some cases. On average, our design improves performance of single-threaded execution 17.5 times more than the improvement achieved by baseline multi-threaded execution resulting in a 2.5 times more energy saving.

Bibliography

[1] Modelsim Simulator. http://www.mentor.com/products/fv/modelsim.

[2] Synopsys Design Compiler. http://www.synopsys.com.

[3] TSMC 65nm GP Standard Cell Libraries - tcbn65gplus. https://www.cmc.ca/en/whatweoffer/products/cmc-00200-01411.aspx.

[4] Mazen AbuZaher, Bayan Alayoubi, Basma Alefeshat, and Abdelwadood Mesleh. *Innovations and Advances in Computer, Information, Systems Sciences, and Engineering*, chapter Dynamic Cache Miss-Rate Reduction, pages 199–204. Springer New York, New York, NY, 2013.

[5] Mirza Beg and Peter van Beek. A graph theoretic approach to cache-conscious placement of data for direct mapped caches. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 113–120, New York, NY, USA, 2010. ACM.

[6] Doug Burger and Todd M Austin. The simplescalar tool set, version 2.0. *ACM SIGARCH Computer Architecture News*, 25(3):13–25, 1997.

[7] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-conscious data placement. *SIGPLAN Not.*, 33(11):139–149, October 1998.

[8] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. *SIGPLAN Not.*, 34(5):1–12, May 1999.

[9] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1st edition, 1990.

[10] A. Ghoting, G. Buehrer, M. Goyder, S. Tatikonda, X. Zhang, S. Parthasarathy, T. Kurc, and J. Saltz. Knowledge and cache conscious algorithm design and systems support for data mining algorithms. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium, 2007. IPDPS 2007.*, pages 1–6, March 2007.

[11] Martin Charles Golumbic. *Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics, Vol 57)*, chapter Graph Theoretic Foundations.

[12] Thomas Kistler. Dynamic runtime optimization. In *Proceedings of the Joint Modular Languages Conference on Modular Programming Languages*, JMLC '97, pages 53–66, London, UK, UK, 1997. Springer-Verlag.

[13] Thomas Kistler and Michael Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Transactions on Programming Languages and Systems*, 22(3):490–505, May 2000.

[14] C. C. Lin and C. L. Chen. Object placement for fully associative cache. In *Proceedings of IEEE/IFIP International Conference on Embedded and Ubiquitous Computing, 2008. EUC '08.*, volume 2, pages 480–485, Dec 2008.

[15] T. Ono and M. R. Greenstreet. Cache prefetching and speculation on multi-threaded processors. In *Proceedings of 2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, pages 206–211, Aug 2013.

[16] Erez Petrank and Dror Rawitz. The hardness of cache conscious data placement. *Nordic Journal of Computing*, 12(3):275–307, June 2005.

[17] Rodric M. Rabbah and Krishna V. Palem. Data remapping for design space optimization of embedded memory systems. *ACM Transactions on Embedded Computer Systems*, 2(2):186–218, May 2003.

[18] Keoncheol Shin, Jungeun Kim, Seonggun Kim, and Hwansoo Han. Restructuring field layouts for embedded memory systems. In *Proceedings of Design, Automation and Test in Europe, 2006. DATE '06.*, volume 1, pages 1–6, March 2006.

[19] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: Sharing-aware scheduling on smp-cmp-smt multiprocessors. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 47–58, New York, NY, USA, 2007. ACM.

[20] S. Thoziyoor, Jung Ho Ahn, M. Monchiero, J.B. Brockman, and N.P. Jouppi. A Comprehensive Memory Modeling Tool and Its Application to the Design and Analysis of Future Memory Hierarchies. In *Proceesings of the 35th International Symposium on Computer Architecture, 2008. ISCA '08.*, pages 51–62, June 2008.

[21] Christian Tinnefeld, Alexander Zeier, and Hasso Plattner. Cache-conscious data placement in an in-memory key-value store. In *ACM International Conference Proceeding Series*, pages 134 – 142, Lisbon, Portugal, 2011.

[22] D. N. Truong, F. Bodin, and A. Seznec. Improving cache behavior of dynamically allocated data structures. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques.*, pages 322–329, Oct 1998.

[23] M. Wickramasinghe and H. Guo. Effective hardware-level thread synchronization for high performance and power efficiency in application specific multi-threaded embedded processors. In *Proceedings of the 2015 33rd IEEE International Conference on Computer Design (ICCD)*, pages 311–318, Oct 2015.

[24] Eddy Z. Zhang, Yunlian Jiang, and Xipeng Shen. Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 203–212, New York, NY, USA, 2010. ACM.