Automatic and Efficient Heap Data Management for Limited Local Memory Multicore Architectures

Ke Bai, Aviral Shrivastava
Compiler and Microarchitecture Laboratory
Arizona State University, Tempe, Arizona 85281, USA
Email: {Ke.Bai, Aviral.Shrivastava}@asu.edu

***Abstract*—Limited Local Memory (LLM) multi-core architectures substitute cache with scratch pad memories (SPM), and therefore have much lower power consumption. As they lack of automatic memory management, programming on such architectures becomes challenging, in the sense that it requires the programmer/compiler to efficiently manage the limited local memory. Managing heap data of the tasks executing in the cores of an LLM multi-core is an important problem. This paper presents a fully automated and efficient scheme for heap data management. Specifically, we propose i) code transformation for automation of heap management, with seamless support for multi-level pointers, and ii) improved data structures to more efficiently manage unlimited heap data. Experimental results on several benchmarks from MiBench demonstrate an average 43% performance improvement over previous approach [1].**

I. Introduction

As we transit from a few cores to many cores, scaling the memory architecture is a major challenge. Cache coherency protocols [2] do not scale well with the number of cores, and therefore maintaining the illusion of a single unified memory in hardware is becoming difficult. As a result, the distributed nature of the memory organization is being exposed to the software. One possible distributed memory architecture is the Non Coherent Cache architecture, which has been tried on the Intel SCC [2]. However, caches still consume a lot of power and die area. One option for a more power-efficient memory hierarchy is to use raw, "un-cached" memory (commonly known as scratch pad memory) in the cores. Scratch pad memory (SPM) consumes 30% less area and power [3] than a direct mapped cache of the same effective capacity. A multi-core architecture in which each core has a SPM instead of cache, and only the core can access its own SPM (and therefore, we refer to it as local memory) is called a Limited Local Memory (LLM) multi-core architecture. The IBM Cell BE [4] is a good example of LLM multi-core architectures.

LLM multi-core architecture is a truly distributed memory architecture on a chip. Consequently, applications have to be written as a bunch of interacting tasks or processes, and the tasks get mapped to the cores of the LLM architecture. Conventionally, *main task* executes on main core and creates *execution tasks*, which are then distributed and executed on execution cores. The global (main) memory is attached to main core, and execution cores only have a limited size of local memory. The execution core can only access its local memory, which implies that *load* and *store* instructions can only access the local memory. Global memory can only be accessed through a special DMA instruction. More importantly, the space of local memory has to be shared between code, stack, global and heap data of the task executing on the core. If the task can be fit into the local memory, then highly power-efficient execution is achieved. For example, power efficiency of the IBM Cell BE is around 5 GFlops per watt [4]. Contrarily, Intel i7 4-core Bloomfield 965 XE can only achieve 0.5 GFlops per watt [5], [6]. However, if memory footprint of the task is larger than the size of the local memory, then the data of the task must be explicitly managed through the use of special DMA instructions. To do this, the programmer must not only be aware of the local memory available in the architecture, but also be cognizant of the memory requirements of the task at every point in the execution of the program. This is very difficult for C/C++ programs, as even though the code and global data sizes are fixed after compilation, stack and heap sizes may be variable and data dependent. Therefore a task that works for a set of inputs may fail for another. On top of this, SPMs are unforgiving in the sense that SPMs just return the data at the requested location. There is no way of checking if the programmer forgot to fetch some data[1]. This is not a problem in cache-based architectures, as hardware brings the required data. Therefore debugging is significantly harder on such architectures. Programmers having to worry about data management has been the biggest roadblock in the success of otherwise very power-efficient LLM multi-core architectures. This burden on the programmers can be relieved through automatic data management solutions.

While management is needed for all code and data on the local memory, managing heap data is especially important, since it is dynamic in nature and may not be known at compile time. Although some works have been done towards managing heap data on SPM [7]–[9], they are not directly applicable to LLM architectures [1]. *In embedded systems, e.g., ARM processors, the SPM was in addition to the regular memory hierarchy, while in LLM multi-core architecture, SPM is an essential part of the memory hierarchy.* Therefore, previous schemes only managed frequently used heap data in SPM, in LLM multicores we need to manage all heap data in SPMs. The only work for heap data in LLM multi-core architectures was proposed by Bai et al. [1]. Since it requires users to manually use their management functions for each heap pointer access, it could be tedious and error-prone. In addition, the overhead of Bai's scheme can be reduced through a better management data structure.

This research was partially funded by grants from National Science Foundation CCF-0916652, IIP-0856090, and NSF I/UCRC for Embedded Systems.

[1]Of course there are mechanisms to make sure that the data you requested has arrived in the memory, but if the programmer does not check or forgets to fetch, then SPMs will just return the old data.

This paper proposes a fully automated and efficient heap data management for LLM multicore architectures. It consists of a modified GCC compiler (GCC 4.1.1) and an efficient heap data management runtime library. Through automatic code transformations, our automated framework unburdens the programmer of the task of API function insertion. Our automation is complete, and also works for multi-level pointers. To reduce the runtime overhead of heap data management, we examined several heap cache design parameters (block size, and associativity), and heap cache design options (victim buffer). Experimental results on several benchmarks from MiBench demonstrate that our work improves application performance by an average of 43% over previous scheme [1]. We found that for our set of benchmarks, a 4-way set associative heap cache without victim buffer is the best choice on average.

II. Previous Work

The local memory in each core of LLM multi-processors is a scratch pad memory or SPM, and is under software control. Although power-efficient, programmers must manage data on the SPM, since SPM lacks memory management logic in the hardware. To simplify programming on processors that feature SPM, several schemes have been proposed to manage code [10]–[19], global data [17], [18], [20]–[22], stack data [9], [15], [22]–[25], and heap data [7]–[9], [26]. However, these techniques cannot be directly applied to LLM multi-core architectures. This is because of the difference in the way SPMs have been traditionally used, and the way they are used in LLM multi-core architectures. In embedded systems, e.g., ARM processors, the SPM is present in addition to the regular cache hierarchy of the processor. Programs can be executed correctly without using SPM, but SPM can be used to optimize performance and power efficiency. To do this, programmers find the most frequently accessed data and map them to SPM. In contrast, the local memory (or SPM) is the only memory hierarchy in an LLM multi-core architecture and is essential, rather than optional. All code and data must go through it. Consequently, while the problem of using SPMs in embedded systems is that of optimization, the problem of using local memory in distributed memory multi-core processors is to enable the execution of applications. Previous research has focused on the question of "what to map" on the SPM, but it is not even an option for LLM multi-core processors.

Work [1] is the only heap data management scheme for LLM architecture, but it is semi-automatic. Basically, they established a mapping between global space and local space, and implemented a pair of functions, *_g2l(ga)* and *_l2g(la)* for address translation. On top of that, two existing functions, *malloc* and *free* were re-implemented. Every time when *malloc* is called, it ensures there is sufficient space in the local memory. If not, some older heap objects will be evicted to global memory. Either case will return a global address. *free* takes in global address and deallocates space in global memory. A heap management table (HMT) was used to maintain the status of heap objects, the object addresses, etc. While comprehensive and correct, previous approach [1] has quite high performance overhead in addition to its requirement of manual function insertion. The main reason is that it manages heap data in a highly associative data structure with LRU (Least Recently Used) replacement policy. The overhead of "heap cache" comprises of DMA overhead and extra instruction overhead to find if a DMA is needed. Although previous approach reduces the number of DMAs, it incurs high overhead in the number of instructions – and this happens at every access, rather than every miss. Furthermore, it has to manage both heap data and HMT, which only adds to the overhead.

III. Heap Data Management Framework

In this paper, we propose a fully automated and low-overhead heap data management scheme, which consists of a modified GCC compiler and an optimized runtime library. The modified compiler can alleviate the pain of library call insertions. Our runtime library includes three heap management functions, *_malloc(size)*, *_free(global_addr)*, and *_g2l(global_addr)*. *_malloc* only allocates space in global memory and returns a global address. The reason a global address is returned is that the mapping relation between global address and local address is "many-to-one". Using a local address is impossible to identify two or more distinct heap objects. *_free* takes in global address and deallocates space in global memory. *_g2l(global_addr)* gets *global_addr* and looks it up in our heap management data structure. If the heap object pointed by *global_addr* is not in the local memory, *_g2l* fetches it from *global_addr* and put it in our data structure. Either case will return the local address in our data structure.

Algorithm 1: Insertion of Function *_g2l*

```
Input: Basic block set $B$, statement list $S$ in GIMPLE IR
Output: GIMPLE IR with the insertion of _g2l
1  foreach basic block $bb \in B$ do
2      foreach statement $s \in S_i$, such that block statement list $S_i$ is for bb, and $S_i \subset S$, $\bigcup S_i = S$ do
3          if $s$ contains multi-level pointers then
4              break down to single level pointers with artificial variables, transform $s$ to statements with only single level pointers
5  foreach basic block $bb \in B$ do
6      foreach statement $s \in S_i$, such that block statement list $S_i$ is for bb, and $S_i \subset S$, $\bigcup S_i = S$ do
7          if $s$ is a modification expression then
8              analyzeStmt (s)
9  Function analyzeStmt(stmt)
10 $l \leftarrow$ getLeftOperand(stmt); $r \leftarrow$ getRightOperand(stmt)
   /* T is a single level ptr with the
      same type as the ptr in the stmt,
      and it is created by compiler     */
11 if TREE_CODE(r) is a reference then
12     create statement "T = _g2l(r)"
13     substitute $r$ with T in stmt
14 else if TREE_CODE(l) is a reference then
15     create statement "T = _g2l(l)"
16     substitute $l$ with T in stmt
17 insert new statement into statement list right before stmt
```

```
        C                          GIMPLE IR
val = **ptr;      ⟹        D.2348 = *ptr;
                           val = *D.2348;
```

Fig. 1. One example of the transformation from C code to GIMPLE IR

A. Compiler Implementation

Our extension of compiler is based on GCC 4.1.1. The compiler support is implemented as a pass at the GIMPLE level, since GIMPLE is a language independent IR and contains high level information, e.g., pointer information. GIMPLE is a three-address IR with tuples of no more than 3 operands (with some exceptions like function calls), and obtained by removing language specific construct from AST (Abstract Syntax Tree) [27].

1) Insertion of Function _g2l: Applications can have heap pointers, stack pointers, and function pointers. Stack pointers are pointers that point to their ancestor function frames. Differentiating a function pointer and a data pointer (stack pointer or heap pointer) is trivial and no details will be explained. We only mention that our framework will not insert *_g2l* for function pointers.

As shown in Algorithm 1, the pass traverses every statement in every basic block of the application. When a memory reference is detected at line 7, the function *analyzeStmt* will insert *_g2l*. As modification expressions in GIMPLE only have the form as "a=b" and only one of them could be a reference, *analyzeStmt* only needs to check if either one is a reference. If a reference is found, our pass creates a statement "T = _g2l(ptr)"(ptr may be l or r) and inserts the statement into the statement list right before *stmt*.

The correctness of the application will not be affected when all data pointers are inserted *_g2l*. *_g2l* itself can distinguish heap pointers from stack pointers, since stack pointers and heap pointers were initialized with local addresses and global addresses respectively. When *_g2l* takes in the parameter, it checks whether the parameter is in local address space or global address space. If it is a local address (which means a stack pointer), the function directly returns the original address. In addition, as our management granularity is at least at heap object level, our framework will not be affected if a heap object contains a function pointer as its element. For example, we consider a statement "H→func = testFunc;". The compiler will use "H" as the parameter of *_g2l* instead of "H→func", where *func* is a function pointer in the heap object H.

2) Multi-level Pointer Support: Previous scheme requires users to manually break down all multi-level pointers to single-level pointers. If the application is very small, this may be possible and intuitive. However, when it is large, this becomes formidable. Our compiler can process all multi-level pointers. This is achieved by breaking down multi-level pointers in *C* to operations containing only single-level pointers in GIMPLE IR, with artificial pointers generated by the compiler. An example of transformation from C to GIMPLE IR is shown in Fig. 1, in which *ptr* is a pointer-to-pointer in *C*. In the example, a pointer read statement is transformed to two statements in the GIMPLE IR, with an artificial pointer *D.2348* generated by compiler. By this transformation, every statement in the GIMPLE IR only has one single-level reference. One thing needs to be mentioned is that, although *D.2348* and *ptr* are both pointers, macro TREE_CODE of them return *var_decl* for *D.2348* but *indirect_ref* for the latter one. Therefore, library calls will only be added for *ptr*. After address translation, *D.2348* gets a local address, and therefore no function is required. TREE_CODE macro is a functionality provided by GCC which can tell what kind of node a particular tree is [27].

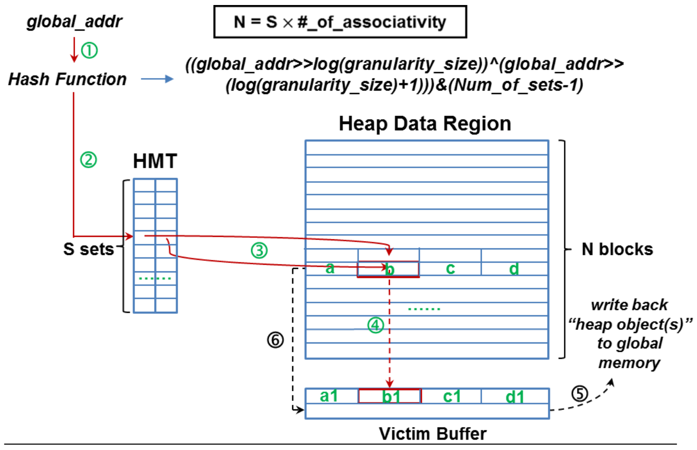

global_addr
①
Hash Function
N = S × #_of_associativity
((global_addr>>log(granularity_size))^(global_addr>>(log(granularity_size)+1)))&(Num_of_sets-1)
Heap Data Region
②
HMT
S sets
③
N blocks
a
b
c
d
......
④
⑥
write back "heap object(s)" to global memory
a1
b1
c1
d1
⑤
Victim Buffer

Fig. 2. Processes of looking up heap objects in 2-way associative heap cache: (1) step 4, 5, 6 might happen. (2) When an old heap block needs to be evicted, replacement runs in a round robin way.

B. Efficient Data Structure for Heap Data Management

1) Heap Cache Data Structure: Fig. 2 shows our heap cache data structure in the local memory. It has S sets, N heap blocks and a hash function. The data structure consists of an array of S entries in HMT (Heap Management Table) and an array of N heap blocks. As a set can contain several blocks, N is therefore equal to the number of sets (S) multiplies the number of associativity (A) (this figure shows a 2-way associative heap cache and we support A-way associative heap cache, where $A = 1, 2, 4, 8$). *granularity_size* and *Num_of_sets* in hash function are configured by users before compilation. Each entry in HMT contains a tag bit, a valid bit, and a modified bit and high bits of global addresses. There is a "one-to-one" static mapping between the entries in HMT and heap blocks. Therefore, it has the property that the number of entries in HMT is equal to the number of heap blocks (N), which means that the size of HMT is fixed. As a result, we do not need to manage HMT between global memory and local memory.

We also provide a victim buffer for heap cache in the local memory. It can be used to relieve the thrashing of heap objects. When a heap object needs to be swapped out of heap region, it will not directly be put to global memory but the victim buffer. Because of this, when there is a heap miss in heap cache, we might find the heap object is in the victim buffer and no slow fetch from global memory is needed.

2) Implementation of _g2l: Several steps involved for each heap access are shown in Fig. 2: (1) When *_g2l* function takes in *global_addr*, the hash function returns the set index corresponding to the requested global address. (2) After finding the set number i, the function directly goes to entry i in HMT, where tag status for set i is stored. Then the valid tags in the selected set are compared to the tag in the global address. (3) After comparison, the function knows which block the accessing heap object should be located[2]. Then, it can further know the object offset of the accessing heap object in the cache block from *global_addr*. In this example, we suppose the offset is 1. Finally, *_g2l* checks the status of the accessing object in the entry i of HMT to determine whether it is in the location b. If there is a valid matching entries in HMT, the request is a hit and the local address is calculated by adding the object offset to the local address of the heap block that corresponds to the matching entry. (4) If not, the miss handler is invoked accordingly. It goes through the fully associative victim buffer to find out whether the heap object is there. When heap object hits, its local address in the block of the victim buffer will be returned. (5) Otherwise, an old heap block following the predefined replacement policy will be selected and evicted out to victim buffer to make space for the coming one. Before overwriting heap block in the victim buffer, the modified bit of the block will be checked. If this block is dirty, it will be written back to the global memory. (6) Otherwise, we can directly overwrite this location. Then, it fetches the heap block that corresponds to the requested global address from global memory and places it in the evicted location.

3) Optimization with SIMD: Our runtime library provides N-way ($N = 1, 2, 4, 8$) associative heap cache. The tag comparisons for the implementation of 4-way associative heap cache and 8-way associative heap cache are performed in parallel with the SIMD (Single Instruction Multiple Data) comparison instruction supported by the execution core (or SPE) [4]. This SIMD programming operates on vector data types that are 128-bits (16-bytes) long. As one entry in HMT is a word long, and therefore 4 comparisons for a set in 4-way associative heap cache can be finished in one SIMD instruction. Accordingly, 8-way associative heap cache requires 2 SIMD instructions for its 8 comparisons of a set.

4) Round Robin Replacement Policy: When a new heap block needs to be brought into the N-way associative heap cache, an old block is chosen to be evicted in a round-robin fashion. Specifically, a counter is maintained for each set of the heap cache. It indicates the index of the next block to be evicted. Whenever a heap block is evicted, the counter is updated by adding one and then modulo the number of blocks in the set (e.g. 4 for 4-way associative heap cache).

IV. Performance Evaluation

A. Experimental Setup

We run several benchmarks on IBM Cell BE [4] with 6 accesses of the 8 SPEs. The benchmarks include applications from Mibench suite [28] and some other possible applications. The benchmarks from Mibench are those that can be executed on our platform and have heap data. Table I shows the details of heap size requirement of each benchmark. MIN is the minimum size needed by our framework, while MAX is the total amount of heap space required by the application (without management). *yes* of Exceed means the application needs to be managed, since it is larger than available space for heap data in the local memory. Here the available space means the subtraction of the size of local memory (i.e., 256KB) and sizes of stack data, code and global data required by the program. When comparing the number of DMAs and cache misses in Table III, we use SimpleScalar [29] to get cache misses.

[2]The block is the granularity of heap management, which means the smallest unit of data transfers.

TABLE I. Heap Requirement of All Benchmarks

Benchmarks	Heap Size (bytes)		Exceed
	MIN	MAX	
basicmath	0	0	no
DFS	16	16000	yes
dijkstra	16	24064	yes
fft	16	262208	yes
invfft	16	262208	yes
MST	16	24576	yes
rbTree	32	49152	yes
sha	0	0	no
stringsearch	16	4096	no

B. Supporting Limitless Heap Data

Our scheme can manage unlimited heap data of the task mapped to the local memory. Although results scale for all benchmarks, only *rbTree* is shown. This benchmark dynamically allocates space for each node and creates a red black tree. The size of the code and global data of the program is 15312 bytes in total and the remaining space is reserved for stack data and heap data. To demonstrate our heap management technique, we changed the number of nodes in the tree from 1 to 131072. As shown in Fig. 3, only 6800 number of nodes can be created in the tree without any heap management, larger than which the program will crash. The reason is that the heap data finally use up the total available space when more number of nodes are created in the tree.

When managed with heap data management techniques, the application is assigned with all available space in the local memory for its heap data and therefore there will be no DMA till the space gets filled. By doing this, we can fairly compare the runtime of application *with* and *without* heap data management. Both previous scheme and ours can enable the execution of the application with very large number of nodes, however, ours has better performance. Detailed comparison will be discussed in the next subsection.

C. Comparisons against Previous Approach

1) Overall Performance: In this experiment, we use 4-way associative heap region without victim buffer for our management, as this configuration is found to be the best choice for most benchmarks in section IV-D. Fig. 4 shows that the average improvement over all benchmarks is 43%. Benchmark *basicmath* and *sha* have no improvement, as they have no heap data. There are two main reasons for improvement. First, Bai et al. [1] implemented LRU (Least Recently Used) replacement for their heap data management. Performance of fully associative heap region is degraded by the sequential table walking to find the valid matching address for heap data request. In contrast, our scheme finds the corresponding set

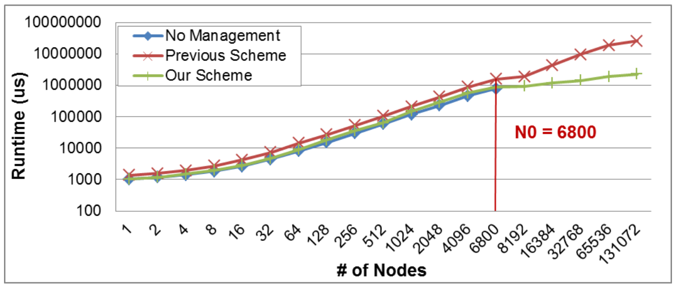


Fig. 3. Supporting Limitless Heap Data

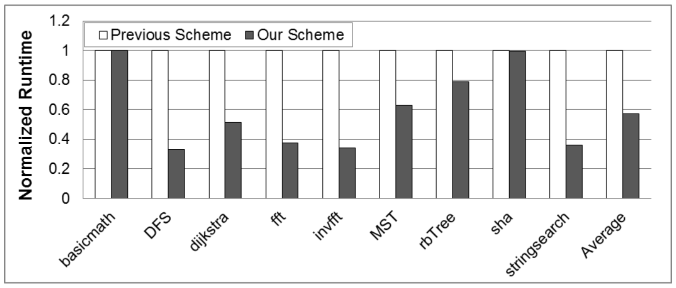


Fig. 4. Runtime comparison between previous scheme and our scheme

by hashing the global address of accessing heap object, which has much less time complexity than that of table looking up. In addition, our scheme uses low associativity for heap data management to further reduce the overhead. Second, in Bai's work [1], the Heap Management Table (HMT) increases as the number of heap objects increases. When the HMT becomes too large to reside in the local memory, expensive data management scheme will be employed to transfer part of HMT entries between global memory and local memory, which severely degraded the performance. To the contrary, the HMT size in our scheme does not change with the increase of heap objects, as our mapping scheme is between HMT entries and heap blocks, rather than between HMT entries and heap objects. In other words, the HMT in our scheme occupies constant space and can be fit into local memory. Therefore, the overhead incurred by HMT data transfers is avoided.

2) Management Overhead: The total overhead consists of the number of extra instructions and the number of data transfers (in terms of the number of DMAs) between global memory and local memory. Table II shows the average extra instructions incurred by each library function call, where we can see our scheme has much less extra instructions per call. Column *hit* for *_g2l* means the accessing heap object is residing in the local memory when *_g2l* is called, while *miss* means the accessing one is not in the local memory when *_g2l* is called. When *miss*, the function first writes back old data and then fetches the required data to the local memory by initiating DMA command. Table III shows the differences between the number of cache misses and the number of DMAs. But we must emphasize that the number of DMAs should not be fully counted as overhead, since penalty also exist for a data miss in the hardware cache. In SimpleScalar, we configured the cache size equaling to the size of heap region in the local memory and only count the misses of heap data. As shown in Table III, both schemes perform less number of DMAs compared to the cache based processors. It is because heap data are initiated in the local memory and DMA happens only when local memory is full. Another important reason is that the granularity is coarser with heap data management, but cache line size in cache based architecture cannot be too large. We also found most benchmarks have less number of DMAs with our scheme, except *fft*, *invfft* and *rbTree*. The overall DMAs are composed of DMAs for heap data and HMT entries. Our scheme has no DMAs for HMT entries. When our scheme performs better, it means the removal of DMAs for HMT entries can compensate more DMAs incurred by poorer temporal locality of heap data.

D. Impact of Heap Cache Parameters

1) Block Size: Fig. 5(a) shows the impact of block size for heap cache. In this experiment, there is no victim buffer

TABLE II. DYNAMIC INSTRUCTIONS PER FUNCTION

	_malloc	**_free**	**_g2l**		**_l2g**
			hit	**miss**	
Previous Scheme	948	50	280	373	243
Our Scheme	60	20	51	117	0

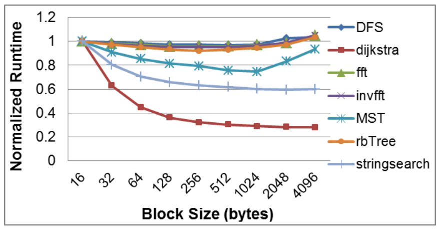
TABLE III. DMAS VS CACHE MISSES

	cache misses	**DMAs**	
		Previous Scheme	**Our Scheme**
basicmath	0	0	0
DFS	26519	1987	250
dijkstra	434867	2656	154
fft	5579	27	238
invfft	5599	27	246
MST	240117	325885	396
rbTree	26519	3958	4322
sha	0	0	0
stringsearch	656	0	0

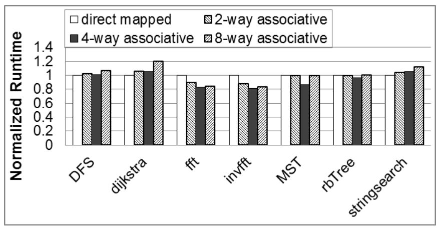
and heap objects are managed in 4-way associative way. We varied block size from 16 bytes to 4096 bytes and made heap cache use all available space in the local memory. The block size is also the minimum data transfer unit between the global memory and the local memory (can be termed as granularity). A larger block size provides the functionality of prefetching as it brings in a few nearby elements together to the local memory. We normalized the runtime of application with other block sizes to that with block size equaling to 16 bytes. As shown in Fig. 5(a), the performance of *dijkstra* and *stringsearch* can be improved by simply increasing the block size. Because they access data sequentially in nature, a large block size takes advantage of data prefetching and higher data transfer bandwidth. On the other hand, other benchmarks can get the best performance with block size between 256 bytes and 512 bytes, since there is a trade-off between transfer granularity and data locality. When block size is small, increasing it can increase the reuse of data. After some time, when increasing block size, data locality is not increased too much but the overhead introduced by the larger transfer size increases a lot.

2) Set Associativity: In this experiment, we explored from direct mapped to 8-way associative heap cache without victim buffer. The heap region size is changed from the minimum size to all the available space in the local memory, and the block size is changed from 16 bytes to 4096 bytes. We normalized the average runtime of N-way (N=2,4,8) associative management to that of direct mapped management. Fig. 5(b) shows 4-way associative heap cache can improve performances of benchmark *DFS*, *fft*, *invfft* and *MST*. For benchmark *dijkstra* and *stringsearch*, the performances are even worse with the higher associativity. The purpose of using a heap region with a higher associativity is to decrease the miss rates and reduce the number of DMA transfers, however, a higher associativity will incur higher computations, such as the computation spent on looking up the management data structure. If the benefit brought by high hit ratio beats the computation overhead, higher associativity is better. Otherwise, higher associativity will degrade the performance.

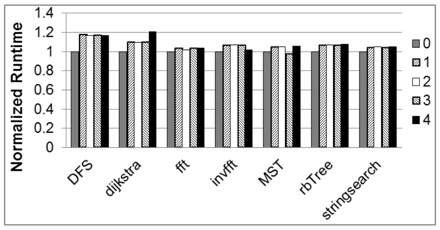
3) Victim Buffer: In order to relieve thrashing of heap objects, we implemented a victim buffer for heap data management. In this experiment, we made heap region use all available space in the local memory, varied the number of blocks in the victim buffer, and changed the block size from 16 bytes to 4096 bytes. In addition, we manage heap objects in 4-way associative way, as we found it is the best associativity in the previous section. Fig. 5(c) shows the average exploration results. We normalized all results with other number of blocks



(a) Impact of Block Size (in Bytes)



(b) Impact of Set Associativity



(c) Impact of Victim Buffer (# of Blocks)

Fig. 5. Impact of Heap Cache Parameters

of victim buffer to the result with 0 block. Each bar means a different number of blocks in the victim buffer, e.g. 0 means heap cache has no victim buffer and 1 means victim buffer has one heap cache block. We can observe that most of the benchmarks have the best performance without victim buffer, except that benchmark *MST* has the best performance when victim buffer can accommodate three heap cache blocks. Victim buffer is designed for thrashing of heap objects, which is heap access pattern dependent. When an application does not have thrashing access pattern, the implementation complexity will add more overhead. In this sense, the performance degradation by the extra instructions for victim buffer implementation outperforms the benefit obtaining from less number DMAs. One the other side, when an application has lots of thrashing access pattern, the filter buffer can reduce the number of conflict misses and improve its performance.

V. Conclusion

In this paper, we propose a compilation and runtime system to manage unlimited size of heap data for limited local memory multi-core architectures. Compared to previous technique, our infrastructure is fully automatic, supports multi-level heap pointers, and has better performance. Experimental results on several benchmarks demonstrate that we can achieve 43% better performance on average over previous approach. In addition, we explored design and parameter space for heap data structure. We explored different block sizes, set associativities and victim buffer sizes. Our results indicate that a 4-way associative heap data structure without victim buffer is the best overall design.

References

[1] K. Bai and A. Shrivastava, “Heap Data Management for Limited Local Memory (LLM) Multi-core Processors,” in *Proc. CODES+ISSS*, 2010, pp. 317–326.

[2] “The SCC Programmer’s Guide,” Tech. Rep.

[3] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, “Scratchpad Memory: Design Alternative for Cache on-chip Memory in Embedded Systems,” in *Proc. CODES+ISSS*, 2002, pp. 73–78.

[4] B. Flachs at el., “The Microarchitecture of the Synergistic Processor for A Cell Processor,” *IEEE Solid-state circuits*, vol. 41, no. 1, pp. 63–70, 2006.

[5] T. Hardware, “Raw Performance: SiSoftware Sandra 2010 Pro (GFLOPS).”

[6] “Intel Core i7 Processor Extreme Edition and Intel Core i7 Processor Datasheet, Volume 1,” in *White paper*. Intel.

[7] A. Dominguez, S. Udayakumaran, and R. Barua, “Heap Data Allocation to Scratch-pad memory in Embedded Systems,” *J. Embedded Comput.*, vol. 1, no. 4, pp. 521–540, 2005.

[8] R. McIlroy, P. Dickman, and J. Sventek, “Efficient Dynamic Heap Allocation of Scratch-pad Memory,” in *Proc. ISMM*, 2008, pp. 31–40.

[9] F. Poletti, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and J. M. Mendias, “An Integrated Hardware/Software Approach for Run-time Scratchpad Management,” in *Proc. DAC*, 2004, pp. 238–243.

[10] S. c. Jung, A. Shrivastava, and K. Bai, “Dynamic Code Mapping for Limited Local Memory Systems,” in *Proc. ASAP*, 2010, pp. 13–20.

[11] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. Olivieri, “A Post-compiler Approach to Scratchpad Mapping of Code,” in *Proc. CASES*, 2004, pp. 259–267.

[12] B. Egger, C. Kim, C. Jang, Y. Nam, J. Lee, and S. L. Min, “A Dynamic Code Placement Technique for Scratchpad Memory Using Postpass Optimization,” in *Proc. CASES*, 2006, pp. 223–233.

[13] A. Janapsatya, A. Ignjatović, and S. Parameswaran, “A Novel Instruction Scratchpad Memory Optimization Method Based on Concomitance Metric,” in *Proc. ASP-DAC*, 2006, pp. 612–617.

[14] M. T. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, “Dynamic Management of Scratch-Pad Memory Space,” in *Proc. DAC*, 2001, pp. 690–695.

[15] N. Nguyen, A. Dominguez, and R. Barua, “Memory Allocation for Embedded Systems with A Compile-time-unknown Scratch-pad Size,” in *Proc. CASES*, 2005, pp. 115–125.

[16] S. Steinke at el., “Reducing Energy Consumption by Dynamic Copying of Instructions onto On-chip Memory,” in *Proc. ISSS*, 2002, pp. 213–218.

[17] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel, “Assigning Program and Data Objects to Scratchpad for Energy Reduction,” in *Proc. DATE*, 2002, p. 409.

[18] M. Verma and P. Marwedel, “Overlay Techniques for Scratchpad Memories in Low Power Embedded Processors,” *IEEE VLSI*, vol. 14, no. 8, pp. 802–815, 2006.

[19] M. Verma, L. Wehmeyer, and P. Marwedel, “Cache-Aware Scratchpad Allocation Algorithm,” in *Proc. DATE*, 2004, p. 21264.

[20] P. Panda, N. D. Dutt, and A. Nicolau, “On-chip vs. Off-chip Memory: the Data Partitioning Problem in Embedded Processor-based Systems,” in *ACM TODAES*, 2000, pp. 682–704.

[21] M. T. Kandemir, J. Ramanujam, and A. N. Choudhary, “Exploiting Shared Scratch Pad Memory Space in Embedded Multiprocessor Systems,” in *Proc. DAC*, 2002, pp. 219–224.

[22] L. Li, L. Gao, and J. Xue, “Memory Coloring: A Compiler Approach for Scratchpad Memory Management,” in *Proceedings of PACT*, 2005, pp. 329–338.

[23] K. Bai, A. Shrivastava, and S. Kudchadker, “Stack Data Management for Limited Local Memory (LLM) Multi-core Processors,” in *Proc. ASP-DAC*, 2011, pp. 231–234.

[24] A. Kannan, A. Shrivastava, A. Pabalkar, and J.-e. Lee, “A Software Solution for Dynamic Stack Management on Scratch Pad Memory,” in *Proc. ASP-DAC*, 2009, pp. 612–617.

[25] M. Mamidipaka and N. Dutt, “On-chip Stack Based Memory Organization for Low Power Embedded Architectures,” in *Proc. DATE*, 2003, pp. 1082–1087.

[26] K. Bai, D. Lu, and A. Shrivastava, “Vector Class on Limited Local Memory (LLM) Multi-core Processors,” in *Proc. CASES*, 2011, pp. 215–224.

[27] *“GCC Internals”*. http://gcc.gnu.org/onlinedocs/gccint/.

[28] 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 *Proc. Workload Characterization*, 2001, pp. 3–14.

[29] T. Austin, E. Larson, and D. Ernst, “SimpleScalar: An Infrastructure for Computer System Modeling,” *Computer*, vol. 35, no. 2, pp. 59–67, Feb. 2002.