Efficient Heap Data Management on Software Managed Manycore Architectures

1st Jinn-Pean Lin
Computing, Informatics, and Decision Systems Engineering
Arizona State University
Tempe, CA, USA
jlin62@asu.edu

2nd Jing Lu
Computing, Informatics, and Decision Systems Engineering
Arizona State University
Tempe, CA, USA
jinglu1@asu.edu

3rd Jian Cai
Computing, Informatics, and Decision Systems Engineering
Arizona State University
Tempe, CA, USA
jcai19@asu.edu

4th Aviral Shrivastava
Computing, Informatics, and Decision Systems Engineering
Arizona State University
Tempe, CA, USA
Aviral.Shrivastava@asu.edu

***Abstract*—Software Managed Manycore (SMM) architectures have been proposed as a solution for scaling the memory architecture. In a typical SMM architecture, Scratch Pad Memories (SPM) are used instead of caches, and data must be explicitly managed in software. While all code and data need to be managed, heap management on SMMs is especially challenging due to the highly dynamic nature of heap data access. Existing techniques spend over 90% of execution time on heap data management, which largely compromised the power efficiency of SMM architectures. This paper presents compiler-based efficient techniques that reduce heap management overhead. Experimental results on benchmarks from MiBench [1] executing on an SMM processor modeled in Gem5 demonstrate that our approach implemented in LLVM 3.8 can improve execution time by an average of 80%, compared to the state-of-the-art [2].**

I. Introduction

As we scale the number of cores in a processor, cache-based memory hierarchy poses a serious limitation due to the rapidly increased demand of area and power for coherence maintenance. On one hand, caches consume significant amount of silicon area and energy[3] and the cost of maintaining cache coherence increases rapidly with the number of cores[4], [5], [6], [7]. On the other hand, cache-based systems are hard to use in real-time systems, since the execution time analysis for cache-based systems is quite complex[8]. For these reasons, some processor vendors have opted to remove caches and use only ScratchPad Memories (SPMs), or allow the caches to be configured as SPMs. An SPM is raw memory that stores only data, without the complex circuitry in a cache to implement automatic movement of data between the lower-level and upper-level memories, replacement policies and coherence. As a result, SPMs consume about 40% less area and energy per access [9]. Processors with only SPMs have been used for high performance computing [10], [11], gaming and multimedia processing [12], digital signal processing [13], and networking [14]. There are also academic researches to design SPM-based processors for various purposes [15].

The trade-off of using SPMs instead of caches is that data movements in and out of the local SPM on each core must be managed explicitly by special instructions (i.e., DMA instruction). For this reason, we refer to such an SPM-only manycore architecture as Software Managed Manycore (SMM) architecture. A lot of techniques have been proposed to mange code [16], [2], stack data [17], [18], and heap data[2], [19]. Among all these data types, heap data is particularly difficult to manage, due to its dynamic nature. However, since heap accesses may account for a significant fraction of all the memory accesses that the application makes, it is extremely important not only to manage heap data, but in an efficient way. This paper only focuses on heap data management, assuming code, global and stack data have been managed efficiently.

The state-of-the-art heap data management [2] enables managing heap data of any task on any SPM size by emulating a 4-way set-associative software cache on an SPM. However, many optimization can be conducted: **i) adjusting the granularity of management by tuning the software cache configurations , and ii) reducing management overhead by not performing management when not absolutely needed**. Experimental results on benchmarks from MiBenc demonstrate that our approach implemented in LLVM 3.8 can improve execution time by an average of 80%.

II. Background and State-of-the-Art

A lot of research has been done on heap data management on scrachpad memories in software [20], [21], [22], [23], [23], [24]. Those techniques, however, are orthogonal to our research, since they are not applicable for SMM cores. In a traditonal embedded cores, the scratchpad memory is in addition to the cache hierarchy, which implies that programs can be executed on the cores without using the scratchpad. On SMM architectures, however, scratchpad memory is the only

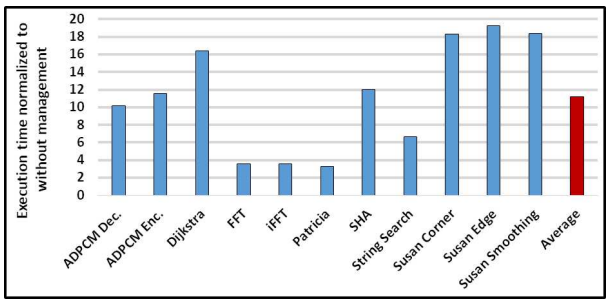


Fig. 1. Performance overhead with the state-of-the-art heap management.

memory type on each of the cores, which implies that data management in software is the only choice.

The state of the art heap management [2] emulates a 4-way set-associative cache on an SPM. The SPM is partitioned into a data region and a heap management table. The data region stores the actual heap data in fixed-sized blocks, while the management table stores a tags, a modified bit, and a valid bit for each block in the data region, i.e. there is a one-to-one mapping between each block in the data region and each entry in the management table. Every 4 entries in the management table forms a set, with a victim index for round-robin replacement policy.

The $g2l$ function implemented in the state of the art [2] takes a main memory address as input, and checks if the given address is in heap. The input address is immediately returned if it is not in heap region. Otherwise, the set index of the input main memory address is calculated. A sequential search is done to compare the tag of the input address with the tags saved in the entries of the corresponding set in the management table. If a match happens and the status of the matching entry is valid, a hit happens. Otherwise, if a miss happens, the enclosing data block of the input address will be copied from the main memory into the SPM.

Although the state of the art [2] correctly manages the heap data of an application, it incurs high performance overhead. Figure 1 shows its management overhead on some typical embedded applications. It is important to note that the heap management technique not only significantly increases the execution time of applications, but also inflicts high overhead on the benchmarks without any heap accesses, `Adpcm Decode`, `Adpcm Encode`, `SHA`, and `String Search`. The high overhead is caused by two main reasons:

i) Too many calls of heap management function $g2l$**.** $g2l$ is called before each memory address (including those are not to heap) to filter non-heap accesses at runtime, which introduces not only overhead at every memory access, but also branch operations, and potentially more memory operations.

ii) High instruction overhead due to the complexity of $g2l$**.** This is because the state of the art implements $g2l$ in a set associative manner. The function has to sequentially search all the entries in the set at every heap access. It also complicates the calculation of the set index, and the translation of a main memory address to the corresponding local SPM address. The set index of the input main memory address is calculated with Equation (1), where mem_addr is the input main memory address, $block\_size$ is the size of a data block, and set_num is the number of sets. The SPM address is then calculated with Equation (2), where spm_base is the start address of the data region, set_assoc is the set associativity (4 in this case), and $entry\_index$ is the index of the entry in the set specified by set_index. The complexity of the calculations translates to significant instruction overhead.

$$set\_index = ((mem\_addr >> log(block\_size)) \wedge (mem\_addr >> (log(block\_size)+1)))\&(set\_num-1) \quad (1)$$

$$spm\_addr = (set\_index * set\_assoc + entry\_index) * block\_size + spm\_base + mem\_addr\%block\_size \quad (2)$$

```
struct { int *a, *b; } s;
int main() {
  int i, *p;
  p = &i;
  *p = 5;
  s->a = malloc(20);
  s->b = s->a + 4;
  *(s->b + 8) = 10;
}
```
(a) Original code

```
int main() {
  int i, *p;
  *g2l(&p) = &i;
  *g2l(*g2l(&p)) = 5;
  *g2l(&(g2l(&s)->a)) = malloc(20);
  *g2l(&(g2l(&s)->b)) =
     *g2l(&(g2l(&s)->a)) + 4;
  *g2l(*g2l(&(g2l(&s)->b)) + 8) = 10;
}
```
(b) Transformed code with the prior work

```
int main() {
  int i, *p;
  p = &i;
  *p = 5;
  s->a = malloc(20);
  s->b = s->a + 4;
  *g2l(&(s->b + 8)) = 10;
}
```
(c) Transformed code with identifying heap accesses statically

Fig. 2. The previous approach inserts $g2l$ before every memory access, while ours tries to identify heap accesses statically and skip unnecessary $g2l$s.

III. Overview of Our Approach

To greatly reduce the overhead of heap management on SMM architectures in the state-of-the-art, a series of optimizations are proposed:

i) statically detecting heap accesses. This optimization identifies heap access at compile-time and eliminates heap management function $g2l$ when the memory is definitely not a heap accesses, and significantly reduces the number of unnecessary management calls at runtime. It also eliminates the runtime checking within the management function, if the memory access is determined to be a heap data access.

ii) simplifying management framework. A direct-mapped cache on SPM is implemented, where it is no longer required to sequentially go through different entries and search for the requested data block for each heap access. In addition, it simplifies the calculation of set index and the SPM address in the management functions. Therefore, this optimization can effectively reduce the number of instructions in each management function.

iii) inline and combine management calls. Inserted $g2l$ functions are inlined and common management instructions are executed before all management calls. This optimization is particularly beneficial, when management functions are called within loop nests, as the common operations are hoisted outside of the loops.

iv) adjusting block size. All the aforementioned optimizations are generic, and thus are useful for all applications. However, in embedded systems, where profiling information can be useful, heap data management can be further optimized. Depending on the type of cache misses an application suffers from, the block can be statically adjusted to avoid these misses. Given the size and set associativity of a software cache, adjusting block size will change the mapping between main memory locations and SPM memory locations.

IV. Details of Our Approach

A. Statically Detecting Heap Accesses

This optimization identifies heap accesses at compile-time, so that the management function *g2l* can be avoided at memory accesses that are definitely not to heap. Figure 2 illustrates the effect of this optimization. The original program defines a structure, which consists of two integer pointers `a` and `b`. It then creates a global variable `s` as an instance of the structure, and assigns `s->a` with an heap object created by a call to the `malloc` function. The program then points `s->b` to the fourth integer element starting from the address in `s->a`. Later `s->b` is used to access the heap object. The program also defines a pointer `p` that refers to a stack variable. Even though only `s->a` and `s->b` points to heap data in this program, the previous heap management technique [2] will insert a *g2l* call at every memory access unnecessarily as in Figure 2(b), including memory accesses via `p` and `s` (not `s->a` or `s->b`), which are to stack and global data respectively. On the other hand, with static detection heap accesses, we only insert *g2l* before the memory instructions via these two pointers.

To find out heap accesses, we first identify all the the heap pointers. Algorithm 1 explains the method we use to identify heap pointers, which includes both the pointers that directly points to heap objects created by memory allocators (e.g., *malloc* or *calloc*), and their aliases. The analysis starts at `getHeapPtr`. In this procedure, the analysis first executes `getAlloc` procedure, taking *main* function as a input (line 2). The `getAlloc` procedure identifies all the invocation of memory allocators in the input function `F`, and records the pointers that are used to store the created heap objects (line 8 and 9). If `F` calls any other functions `F'`, `getAlloc` recursively accesses and identifies the memory allocations in `F'` (line 11 and 12). Once all the heap pointers that store the heap objects created by memory allocations are identified, the analysis continues to identify all the possible alias of these heap pointers by executing the `getAlias` procedure on *main* function (line 4). The `getAlias` procedure goes through each instruction in the input function `F`, and recognizes any instruction that performs pointer arithmetic on a heap pointer and assigns the result to another pointer. The destination pointer of such an instruction is identified as an alias of the heap pointer. Similar to the `getAlloc` procedure, in case `F` calls any other function `F'`, the `getAlias` procedure recursively calls itself on `F'` to identify aliases created in `F'`. Since each iteration of the `getAlias` procedure may recognize new aliases, this procedure is repeated until no new aliases can be recognize (line 3 to 5).

Once all heap pointers are recognized, we can identify heap accesses and insert *g2l* function as follow. All the memory access (i.e. loads ans stores) via any of the heap pointers identified in Algorithm 1 are considered as potential heap accesses. A *g2l* function is inserted right before the memory instructions to first translate the memory address to an SPM address. The SPM address is then used to substitute for the original memory address in the instructions.

Algorithm 1 Identify heap pointers

```
1: function GETHEAPPTR
2:     getAlloc(main)
3:     repeat
4:         getAlias(main)
5:     until cannot find new aliases
6: function GETALLOC(Function F)
7:     for each instruction inst in F do
8:         if inst is a call to any memory allocator then
9:             Record destination pointer P as a heap pointer
10:        else
11:            if inst is a call to any user function F' then
12:                getAlloc(F')
13: function GETALIAS(Function F)
14:    for each instruction inst in F do
15:        if inst is an assignment statement with one operand P be a heap pointer then
16:            Record destination pointer P' as an alias of P
17:        else
18:            if inst is a call to any user function F' then
19:                getAlias(F')
```

```
int main() {
  int a[10], *b, *c;
  b = malloc(40);
  c = (rand() % 2) ? b : &a;
  b[3] = 20;
  c[4] = 15;
}
```

(a) Sample code

```
int main() {
  int a[10], *b, *c;
  b = malloc(40);
  c = (rand() % 2) ? b : &a;
  *g2l(&b[3]) = 20;
  *g2l_rc(&c[4]) = 15;
}
```

(b) Insert g2l at definite heap accesses. Otherwise insert g2l_rc to first check if an access is to heap at runtime

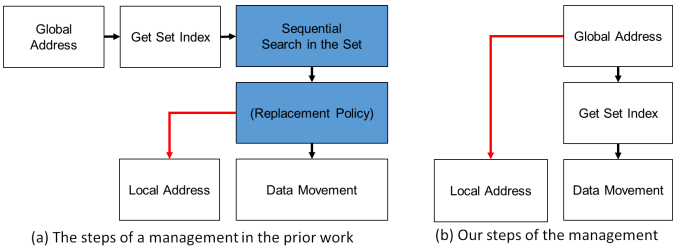
Fig. 3. When a memory access may be to heap but is not for certain, we check at runtime before managing the access.

There are cases when the compiler cannot determine whether a pointer refers to heap data. In Figure 3(a), the pointer `c` can either refer to heap data or stack data, depending on the outcome of the call to `rand` function, which returns a random number. A new management function called *g2l_rc* that checks at runtime and sees if the memory address is in heap, similar to the previous work, is introduced. When an access is assured to heap, the *g2l* function is called, which does not have any runtime checking. If an access may be to heap, *g2l_rc* is called instead. Otherwise, if an access is determined definitely not to heap, no heap management function will be invoked. Figure 3(b) shows the transformed code with heap management function. *g2l* is called before accessing the data referred by the pointer `b`, because it is in heap. *g2l_rc* is invoked before accessing `c`, because it may refer to heap data, but are not for sure. No heap management function is added when accessing `a` due to its access to stack data.

B. Simplifying Management Framework

Whenever a memory access happens, a software-cache based approach has to first calculate the set index of the memory address. The software cache will then sequentially access the entries in the set and compare the tag of the target address with the tags in the entries. Once the data block that contains the target address is located, either already in the SPM in a hit, or first copied from the main memory in a miss, the final SPM address is generated and used to replace the original memory address in the memory access.

Since this process happens within each management function call, it is performance critical to speed up this process. With a direct-mapped cache on software, this process can



(a) The steps of a management in the prior work (b) Our steps of the management

Fig. 4. (a) The steps of a management in the previous work (b) The steps of a management function in our approach The steps of a management function in the previous work and our approach.

be noticeably simplified to execute much less instructions at runtime, compared to a set-associative cache. Figure 4(a) and Figure 4(b) show two examples using the previous approach and our approach respectively. The edge in both figures specify dependence between two steps. The previous approach as in Figure 4(a) calculate set index with Equation (1). The software cache then searches the corresponding set for the requested data block. Only after the data block is found (either after a hit or after a miss), can then the SPM address be generated Equation (2). Notice this equations required both the index of the set and the index of the entry in the set, which explains the dependence of the calculation of the SPM address on the sequential searching in Figure 4(a). On the other hand, our approach in Figure 4(b) simplifies the calculation of the set index of a memory address into $set\_index = global\_addr >> log(block\_size)\%set\_num$. Since each set has only one entry, sequential searching is not necessary. The software can simply go ahead and calculate the final SPM address as $spm\_addr = spm\_base + mem\_addr\%(set\_num * block\_size)$. In addition, the calculation of SPM does not depend on any previous steps. Elimination of such dependence may allow the compiler to have more parallelism when generating and scheduling the machine instructions for the management functions.

C. Inlining and Combining Management Calls

Once $g2l$ function is inserted after identifying heap accesses statically, we can reduce the management overhead by inlining the management functions, which enables further optimization. In Section II we explained the previous approach divided SPM into two memory regions for heap management table and data region. Our approach makes similar usage of SPM space. Every $g2l$ thus has to load the start address of the heap management table and data region at the beginning of its execution, before executing any other call-specific instructions. Therefore, we can move these common instructions outside of the $g2l$ function and execute it once before any $g2l$ calls, so that all the subsequent $g2l$ calls can reuse the results, similar to common subexpression elimination.

Figure 5 illustrates the idea. Figure 5(a) shows the original code. Figure 5(b) is the transformed code before inlining. Each $g2l$ call first executes the common instructions redundantly, and then execute specific instructions for that call. We represent the common instructions and specific instructions in a $g2l$ with function calls $g2l\_common$ and $g2l\_specific$ respective in the example, but they are plain instructions in the actual implementation. In Figure 5(c), we inline the $g2l$ calls, move and execute the common instructions at the beginning of

```
spm_addr g2l(mem_addr):
  g2l_common();
  spm_addr = g2l_specific(mem_addr);
  return spm_addr;
```

```
int main() {
  heap[0] = 2;
  heap[1] = 4;
  for(i = 2; i < n;
i++)
    heap[i] = i;
}
```

(a) Original code

```
int main() {
  *g2l(&heap[0]) = 2;
  *g2l(&heap[1]) = 4;
  for(i = 2; i < n; i++)
    *g2l(&heap[i]) = i;
}
```

(b) Management calls with prior technique

```
int main() {
  g2l_common();
  *g2l_specific(&heap[0]) = 2;
  *g2l_specific(&heap[1]) = 4;
  for(i = 2; i < n; i++)
    *g2l_specific(&heap[i]) = i;
}
```

(c) Inlining management calls and eliminate common operations

Fig. 5. Inlining management calls and move common operations to the beginning of the caller function.

the caller function. After the optimization, only call-specific instructions are executed at where a $g2l$ was called. While this optimization should definitely improve performance, its importance is maximized when $g2l$ was originally called within loop nests, as this example shows —instead of repeatedly and excessively executing the common steps in a loop nest, moving these common instructions to be outside can significantly reduce such overhead.

In addition, at compile time, the modified compiler goes through every function in the program, inlines $g2l$ calls with call-specific instructions, and moves the common instructions to the beginning of the function.

V. Adjusting Block Size for Embedded Applications

When the capacity and associativity of a cache are given, the size of block size decides the number of sets. Different choices of block size may end up causing drastically different performance. We can therefore analyze the access pattern and find a block size that can achieve good performance. When a program is susceptible to cache thrashing, we can decreases block size to lower the chance of such undesirable situation. Cache thrashing refers to excessive conflict cache misses that happen when multiple main memory locations competing for the same cache blocks. It may happen when more than two heap objects with aggregate types (e.g., arrays) are accessed within the same loop. On the other hand, we can increasing block size to improve spatial locality under certain circumstances.

We proposed a heuristic that goes through all innermost loops in a program and adjusts block size based on profiling. Whenever it identifies more than two heap objects are accessed within the loop, it reduces the block size to increase the number of sets for avoiding cache thrashing; otherwise, it increases the block size to increase spatial locality. This analysis is statically done. Therefore, this optimization is the most effective for embedded applications using representative input.

VI. Experiments

A. Experimental Setup

Both the state-of-the-art [2] and our technique are implemented as intermediate representation (IR) passes on LLVM

TABLE I
MAXIMUM HEAP USAGE OF BENCHMARKS

Benchmark	Heap Size (KB)	Benchmark	Heap Size (KB)
Adpcm Decode	0	SHA	0
Adpcm Encode	0	String Search	0
Dijkstra	6.43	Susan Corner	92.16
FFT	32	Susan Edge	42.81
iFFT	32	Susan Smoothing	17.35
Patricia	766	Typeset	32

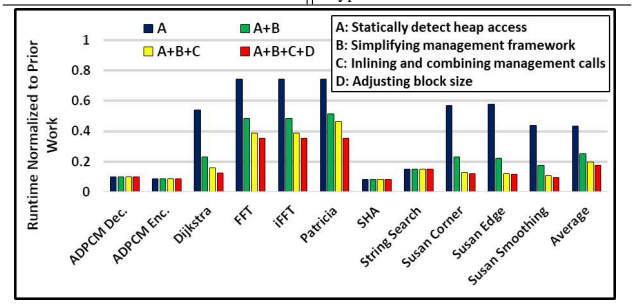


Fig. 6. The execution time of our approach normalized to the previous work with optimizations incrementally added.

3.8 [25] respectively. The same benchmarks are compiled with different heap management techniques, and the executable code is ran on Gem5 [26]. The block size in the software cache is set to 64 bytes in both techniques by default, and is only varied in the fourth optimization to reduce cache misses.

SMM architecture is emulated on Gem5, by modifying the linker script and reserving part of the memory address space as the SPM. A DMA instruction is implemented to copy data between the SPM and the main memory. DMA cost is modeled as a constant startup time and the time for actual data movement. The startup time is set to 291 cycles, and the rate for transferring data is set to 0.24 cycles/byte. The CPU frequency is set to 3.2 GHz. All these parameters are based on the IBM Cell processor [27].

The proposed technique is evaluated on Mibench benchmark suite [1]. Table I lists the maximum usage of heap data in the benchmarks, i.e., the maximum sum of sizes of heap objects at any moment. The benchmarks that have zero heap usage do not have any heap accesses.

B. Significantly Reduces Execution Time

Figure 6 shows the execution time of our approach normalized to the previous work, when each of the optimization is incrementally introduced. Overall, our approach reduces execution time by 80% on average with the first three generic optimizations, i.e., without adjusting block size. When we apply all four optimizations, the execution time is reduced by 83% on average.

As shown in Figure 6, statically detecting heap accesses contributes the largest reduction of execution time, especially in benchmarks that do not have any heap accesses, i.e., `Adpcm Decode`, `Adpcm Encode`, `SHA`, and `String Search`. Overall, it reduces the execution time by 57% on average, due to reduced management calls and less executed instructions in each call. Table II shows the number of calls to the $g2l$ function before and after statically detecting heap accesses in the previous work. The calls are significantly reduced in all the benchmarks, and they are completed eliminated if the benchmarks do not have any heap access.

Statically detecting heap accesses also eliminates runtime checking at $g2l$s, and thus reduces the number of instructions executed in each $g2l$. Table III shows the average number of instructions each $g2l$ executes under different cases, after we incrementally introduce the optimizations. There are 3 possible cases when a $g2l$ function is called: a cache hit, a cache miss with an unmodified data block is chosen to be evicted, and a cache miss with a dirty data block is chosen to be evicted. The memory access may either be a read access or a write access, so there are 6 different cases overall that may happen when calling a $g2l$ function. The table clearly shows there is a constant difference of 6 instructions between the Previous Work column and the Statically Detecting Heap Accesses column in any case.

Simplifying management framework, by implementing a direct-mapped software cache instead of a 4-way set-associative cache, reduces execution time by 42% on average (on top of statically detecting heap accesses). This is because average dynamic instruction count of $g2l$ calls in all the cases of Table III is significant reduced. For example, the average instructions executed in the sixth case is reduced from 166 to 58 after simplifying management framework. Since a direct-mapped cache causes more cache misses compared to a 4-way set-associative cache, we also compare the benefit (reduced cycles) due to less management instructions to the penalty (increased cycles) due to increases cache misses. Figure 7 shows the reduced CPU cycles thanks to less management instructions normalized to the increased CPU cycles because of more cache misses. The simplification of management framework improves the performance of a benchmark, as long as the quotient of that benchmark is greater than 1. For example, in `Patricia`, the reduced cycles are more than 10000000 times than the increased cycles. The figure shows that the increased cycles almost are ignorable compared to the reduced cycles, in all the benchmarks.

Inlining and combing management calls can further reduce

TABLE II
NUMBER OF $g2l$ CALLS CALLED BEFORE AND AFTER IDENTIFYING HEAP ACCESS STATICALLY WITH THE PREVIOUS TECHNIQUE

Benchmark	Unoptimized	Optimized
Adpcm Decode	116702082	0
Adpcm Encode	10211280	0
Dijkstra	149209166	19077784
FFT	336608	90188
iFFT	336671	90204
Patricia	3114668	893184
SHA	8350153	0
String Search	2198090	0
Susan Corner	1238553	273717
Susan Edge	2628207	579221
Susan Smoothing	37252034	4891730
Typeset	274118	3826

TABLE III
INSTRUCTIONS EXECUTED PER $g2l$ UNDER DIFFERENT CASES

Case	Previous Work	Statically Detect Heap Accesses	Simplify $g2l$	Inline and Combine $g2l$
read hit	52	46	19	8
write hit	59	53	23	10
read miss w/o WB	145	139	41	36
write miss w/o WB	145	139	44	37
read miss w/ WB	172	166	58	45
write miss w/ WB	172	166	58	45

note: WB means write-back.

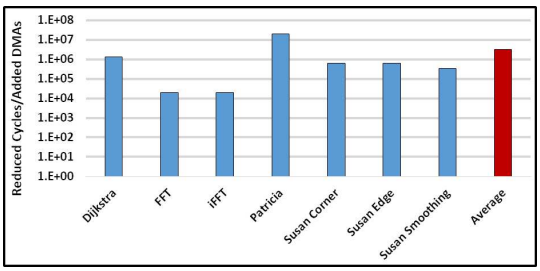


Fig. 7. Implementing a direct-mapped cache other than a 4-way set-associative cache reduces more execution time thanks to simplified management functions, compared to the extra time introduced due to increased cache misses.

execution time by 21% (on top of statically detecting heap accesses and simplifying management framework), thanks to the removed function calls and redundant operations. For example, as Table III shows, the average instructions executed in the sixth case is reduced from 166 to 58 after simplifying management framework, and is further reduced from 58 to 45 after inlining and combing management calls. Notice we apply this optimization after statically detecting heap accesses. So if heap management calls are all eliminated after that step, inlining and combining management calls will not improve performance. For example, the management calls of `Adpcm Decode`, `Adpcm Encode`, `SHA`, and `String Search` are reduced to 0 after the compiler statically finds out there are no heap accesses in these benchmarks. The performance is therefore not further improved after the first optimization.

The block size was set to 64 bytes by default. When analyzing the effectiveness of optimization by adjusting block size, we analyzed programs and adjusted the block size to 16 bytes when it needed to be decreased, and to 1024 bytes when it needed to be increased. The decision on block size was based on profiling information. Adjusting block size could further reduce execution time by 11% (on top of the previous three optimizations).

VII. Acknowledgement

This work was partially supported by funding from NIST Award 70NANB16H305, and by National Science Foundation grants, CAREER CCF-0916652, CNS 1525855, CPS 1645578, and CCF 172346 - the NSF/Intel joint research center for Computer Assisted Programming for Heterogeneous Architectures (CAPA).

VIII. Conclusion

Due to the expense of caches, SPM-based processors have been widely used in various areas. However, the data management must be explicitly done on SPM. This paper presents an efficient heap management that consists of three generic optimizations (statically detecting heap accesses, simplifying management framework, and inlining and combining management calls). The experimental results show that the execution time is reduced by 80% on average with the three general optimizations compared to the state of the art, while it can be reduced by 83% on average with all four optimizations.

References

[1] 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 *International Workshop on Workload Characterization*, 2001.

[2] K. Bai and A. Shrivastava, "Automatic and Efficient Heap Data Management for Limited Local Memory Multicore Architectures," in *Proc. of DATE*, 2013.

[3] S. Niar, S. Meftali, and J. L. Dekeyser, "Power Consumption Awareness in Cache Memory Design with SystemC," in *Proc. of ICM*, 2004.

[4] G. Bournoutian and A. Orailoglu, "Dynamic, Multi-core Cache Coherence Architecture for Power-sensitive Mobile Processors," in *Proc. of CODES+ISSS*, 2011.

[5] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou, "DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism," in *Proc. of PACT*, 2011.

[6] A. Garcia-Guirado, R. Fernandez-Pascual, A. Ros, and J. Garcia, "Energy-Efficient Cache Coherence Protocols in Chip-Multiprocessors for Server Consolidation," in *Proc. of ICPP*, 2011, pp. 51–62.

[7] Y. Xu, Y. Du, Y. Zhang, and J. Yang, "A Composite and Scalable Cache Coherence Protocol for Large Scale CMPs," in *Proc. of ICS*, 2011.

[8] R. Wilhelm and et al., "The Worst-case Execution-time Problem&Mdash;Overview of Methods and Survey of Tools," *ACM Trans. Embed. Comput. Syst.*, 2008.

[9] 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. of CODES*, 2002.

[10] N. P. Carter and et al., "Runnemede: An Architecture for Ubiquitous High-Performance Computing," in *Proc. of HPCA*, 2013.

[11] REX Computing, Inc., "THE NEO CHIP," http://rexcomputing.com/, 2014.

[12] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic Processing in Cell's Multicore Architecture," *IEEE Micro*, vol. 26, 2006.

[13] Texas Instrument, "TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor (Rev. E)," http://www.ti.com, 2014.

[14] A. Olofsson, "Epiphany-V: A 1024 processor 64-bit RISC System-On-Chip," *CoRR*, 2016.

[15] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, "SODA: A Low-power Architecture For Software Radio," *SIGARCH Comput. Archit. News*, 2006.

[16] S. C. Jung, A. Shrivastava, and K. Bai, "Dynamic Code Mapping for Limited Local Memory Systems," in *Proc. of ASAP*, 2010, pp. 13–20.

[17] J. Lu, K. Bai, and A. Shrivastava, "SSDM: Smart Stack Data Management for Software Managed Multicores (SMMs)," in *Proc. of DAC*, 2013, pp. 149–156.

[18] K. Bai, A. Shrivastava, and S. Kudchadker, "Stack Data Management for Limited Local Memory (LLM) Multi-core Processors," in *Proc. of ASAP*, Sept 2011, pp. 231–234.

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

[20] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, *Dynamic Storage Allocation: A Survey and Critical Review*, 1995.

[21] R. McIlroy, P. Dickman, and J. Sventek, "Efficient Dynamic Heap Allocation of Scratch-pad Memory," in *Proc. of ISMM*, 2008.

[22] A. Dominguez, S. Udayakumaran, and R. Barua, "Heap data allocation to scratch-pad memory in embedded systems," *J. Embedded Comput.*, 2005.

[23] E. G. Hallnor and S. K. Reinhardt, "A Fully Associative Software-managed Cache Design," in *Proc. of ISCA*, 2000.

[24] P. Chakraborty and P. R. Panda, "Integrating Software Caches with Scratch Pad Memory," in *Proc. of CASES*, 2012.

[25] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proc. of CGO*, 2004.

[26] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, 2011.

[27] M. Kistler, M. Perrone, and F. Petrini, "Cell Multiprocessor Communication Network: Built for Speed," *IEEE Micro*, 2006.