Optimizing Heap Data Management on Software Managed Manycore Architectures

by

Jinn-Pean Lin

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved June 2017 by the
Graduate Supervisory Committee:

Aviral Shrivastava, Chair
Fengbo Ren
Umit Ogras

ARIZONA STATE UNIVERSITY

August 2017

ABSTRACT

Caches pose a serious limitation in scaling many-core architectures since the demand of area and power for maintaining cache coherence increases rapidly with the number of cores. Scratch-Pad Memories (SPMs) provide a cheaper and lower power alternative that can be used to build a more scalable many-core architecture. The trade-off of substituting SPMs for caches is however that the data must be explicitly managed in software. Heap management on SPM poses a major challenge due to the highly dynamic nature of of heap data access. Most existing heap management techniques implement a software caching scheme on SPM, emulating the behavior of hardware caches. The state-of-the-art heap management scheme implements a 4-way set-associative software cache on SPM for a single program running with one thread on one core. While the technique works correctly, it suffers from significant performance overhead. This paper presents a series of compiler-based efficient heap management approaches that reduces heap management overhead through several optimization techniques. Experimental results on benchmarks from MiBenchGuthaus *et al.* (2001) executed on an SMM processor modeled in gem5Binkert *et al.* (2011) demonstrate that our approach (implemented in llvm v3.8Lattner and Adve (2004)) can improve execution time by 80% on average compared to the previous state-of-the-art.

TABLE OF CONTENTS

LIST OF TABLES

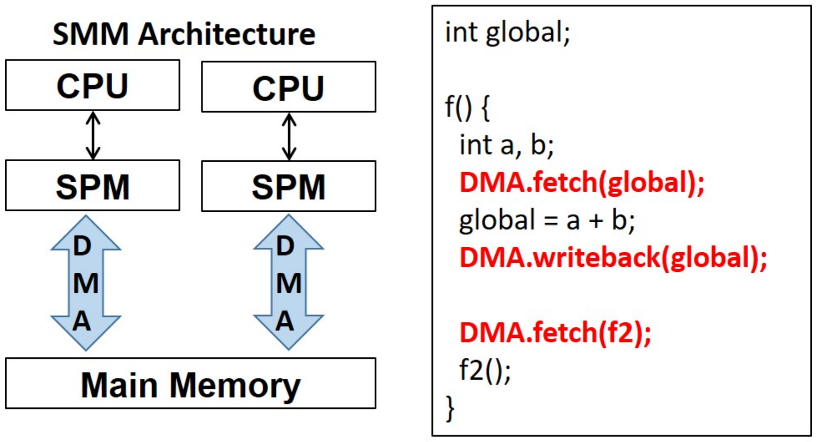
LIST OF FIGURES

Figure Page

Chapter 1

INTRODUCTION

Cache hierarchy is critical to the performance of modern processors, as it significantly reduces memory access latency. However, caches suffer from a string of disadvantages. They consume significant amount of silicon area and energyNiar *et al.* (2004) and the cost of maintaining cache coherence increases rapidly with the number of coresBournoutian and Orailoglu (2011); Choi *et al.* (2011); Garcia-Guirado *et al.* (2011); Xu *et al.* (2011). In addition, cache-based systems are hard to use in real-time systems, since the execution time analysis for cache-based systems is quite complexWilhelm *et al.* (2008). For these reasons, some processor vendors have opted to remove caches and use only ScratchPad Memories (SPMs), or provide the ability for 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



```
int global;

f() {
  int a, b;
  DMA.fetch(global);
  global = a + b;
  DMA.writeback(global);

  DMA.fetch(f2);
  f2();
}
```

Figure 1.1: Diagram of a 2-core SMM Architecture, Showing That Each Core Has a Local SPM. To Make a Program Run on a SPM-based Processor, Explicit Direct Memory Access (DMA) Instructions to Move Data Between the SPM and the Main Memory Must Be Inserted.

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 Banakar *et al.* (2002). Processors with only SPMs have been used for high performance computing Carter *et al.* (2013); REX Computing, Inc. (2014), gaming and multimedia processing Gschwind *et al.* (2006), digital signal processing Texas Instrument (2014), and networking Olofsson (2016). There are also academic researches to design SPM-based processors for various purposes Lin *et al.* (2006).

The trade-off of using SPMs instead of caches is that the data movements in and out of the SPM must now be managed explicitly by software. For this reason, we refer to such an SPM-only manycore architecture as Software Managed Manycore (SMM) architecture. Figure 1.1 shows a diagram of a 2-core SMM architecture, in which each core is connected to an SPM. Programs that run on SMM architecture must have explicit instructions inserted in them to move data into and out of the SPM in order to enable correct execution. While a programmer can be entrusted to insert these data movement instructions, the task is not only difficult and time consuming, but it can also be error-prone. Therefore, much research has gone into developing compiler techniques that can automatically insert these data movement instructions into the program, and ensure correct and efficient execution of the program. Although all data (heap data, stack data and global data) of the application needs to be managed (i.e., data movement instructions inserted in the program), when we want to execute an application on an SMM architecture, efficient heap data management is particularly important, since heap accesses may account for a significant fraction of all the memory accesses that the application makes. Simply allocating and directly accessing these heap data from the main memory without using SPM could cause extremely high overhead.

Previous approaches to manage heap data on SPM can be classified into static approaches, quasi-static approaches, and dynamic approaches. Static approaches like Wilson *et al.* (1995); McIlroy *et al.* (2008) only map a fraction of the heap data (typically most accessed) to the SPM, and the application can directly access it from there. However, since the SPM may be able to hold only a very small amount of heap data, this approach may not be effective and scalable. Quasi-static approachesDominguez *et al.* (2005) break the program execution into time intervals, and in each interval, they apply static heap management. Dynamic approachesHallnor and Reinhardt (2000); Moritz *et al.* (2001); Chakraborty and Panda (2012); Bai and Shrivastava (2013) are the most versatile and scalable. Depending on the program requirements, dynamic techniques not only keep changing the set of data objects that is mapped onto the SPM, but also the location of each data object at runtime. This allows the dynamic heap management techniques to make the most efficient use of SPM space.

Dynamic heap management requires modifications of user code. By default, all heap data is accessed using the address of the heap data in the main memory in programs (that run correctly on cache-based processors). However, when we bring

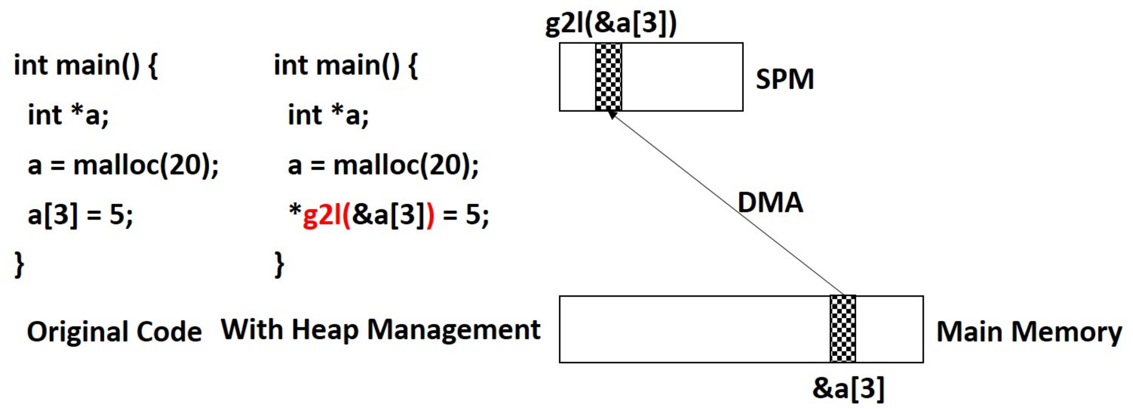

```
int main() {
  int *a;
  a = malloc(20);
  a[3] = 5;
}
```
```
int main() {
  int *a;
  a = malloc(20);
  *g2l(&a[3]) = 5;
}
```

Figure 1.2: The Heap Management Function *g2l* Takes a Memory Address as Input and Maps It to a Location in SPM. If the Data in the Memory Location Is Not Already Present in the SPM Location, *g2l* Function Will Also Issue a DMA to Copy the Data.

the heap data to the SPM, it must be accessed using the SPM address [1] , which in general will be different from the main memory address. Figure 1.2 (a) shows that in the original code, `malloc` function is called to allocate a heap object in the main memory. The start address of the heap object is stored in the pointer `a`. The store to `a[3]` directly access the main memory address. However, for an SPM-based processor, we must fetch `a[3]` from the main memory, and then access the SPM location. This is achieved by calling a heap management function, that we refer as *g2l* in the rest of the paper [2] , on the memory address of `a[3]` before we access its value. The function *g2l* first looks if the data is already present in its heap data structure in the SPM. If not, it will fetch it from the main memory, and return the SPM address of the data. Note that, since SPM has fixed capacity, it may need to evict some other existing heap data from the SPM to the main memory. All dynamic heap management techniques must implement *g2l*-like management functions.

The implementation of the *g2l* function in the state-of-the-art dynamic heap management technique on SMM architecturesBai and Shrivastava (2013) is correct, and enables the execution of single threaded programs with heap data on a core of SPM-based processors, but causes high overhead. There are two main reasons for the overhead: Firstly, their implementation checks whether a memory access is to heap at runtime in *g2l*, by comparing the target address to the address range of heap. Therefore, *g2l* has to be called before all the memory accesses, even when they are not to heap. Secondly, their implementation emulates a 4-way set-associative software cache with round-robin replacement policy, whose complexity causes significant instruction overhead.

[1]The SPM is physically addressed

[2]terminology borrowed from Bai and Shrivastava (2013)

To solve the problems, we propose three generic optimization techniques and one additional optimization technique for embedded applications. We first i) statically detect heap accesses at compile-time, so that *g2l* is avoided before definite non-heap accesses. We also ii) implement a direct-mapped cache like data structure to manage the heap data in SPM, which greatly simplifies the logic and thus multiplicatively reduces the instruction overhead of *g2l*, which allows us to iii) inline the *g2l* calls and remove redundant operations. Finally, we iv) adjust block size depending on the types of cache misses for embedded applications.

We implement the proposed techniques on LLVM 3.8 Lattner and Adve (2004), and evaluate them on Gem5 CPU simulator Binkert *et al.* (2011). The benchmarks used for evaluation are from Mibench suite Guthaus *et al.* (2001). Experimental results show that compared to the state-of-the-art, our techniques can effectively reduce the number of management calls, and the number of instructions executed within each management call. Even though using a direct-mapped software cache increases cache misses compared to a 4-way set-associative software cache, the experimental results show the performance boost from reduced instruction overhead is significantly more than the performance penalty caused by increased misses. As a result, our three generic optimization techniques reduce execution time by 80% on average with the first three optimizations. With all four optimizations, we can reduce execution time by 83% on average.

Chapter 2

RELATED WORK

Heap management is very important to application performance, since heap accesses may account for a significant portion of overall memory accesses. Figure 2.1 shows the percentage of memory accesses to heap out of all the data accesses in MiBench benchmarks. While heap accesses may not be present in all the programs, it is dominant in some, with more than 90% in `Susan Smoothing`.

Heap management on SPM can be generally divided into static approaches, quasi-static approaches and dynamic approaches. Static approaches treat an SPM as a heap, and implement efficient memory allocator to manage heap data on the SPM Wilson *et al.* (1995); McIlroy *et al.* (2008). Such methods avoid run-time overhead at every memory access. However, such methods may be forced to allocate heap objects to main memory when there are not enough space on SPM. Notice that static approaches

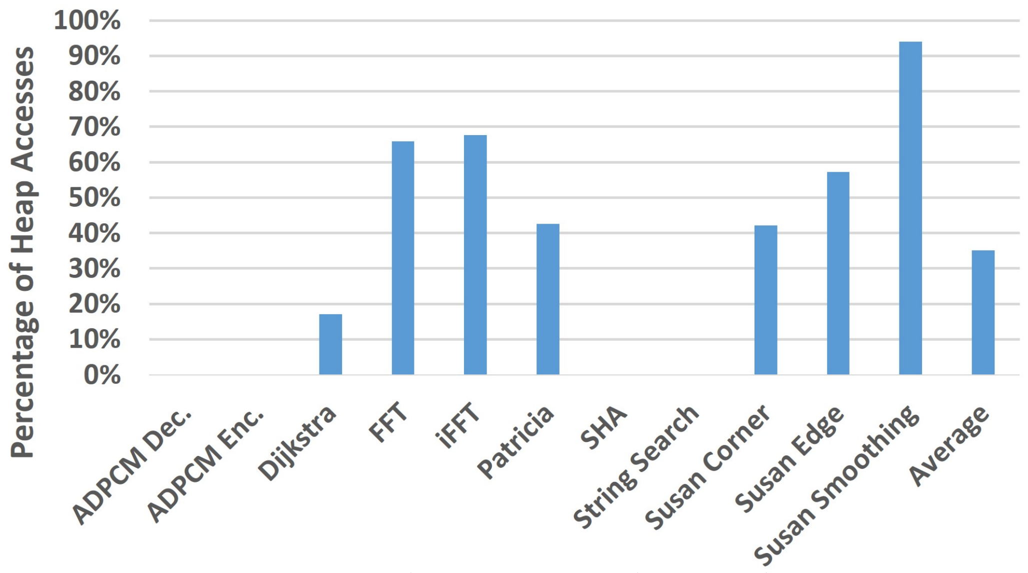


Figure 2.1: Percentage of Heap Accesses among All Data Accesses.

allocate/deallocate heap objects dynamically at runtime. The addresses of the heap data, however, is never changed until it is deallocated, and is therefore considered as static approaches.

Quasi-static approaches divide execution into time intervals, and bring the most frequently used data of each interval into the SPM at its beginning Dominguez *et al.* (2005). Within the interval, locations of heap data are fixed (thus the name quasi-static), either in SPM or in the main memory. Such approaches usually rely on profiling, and can be extremely efficient when representative input is known in advance, e.g., in embedded applications. However, profiling can be inaccurate in general, when application behaviors vary significantly as the input changes. On the contrary, most of our approach do not rely on profiling. The only optimization in our approach (namely, adjusting block size to reduce cache misses) that relies on profiling is for embedded applications, where profiling is acceptable.

Dynamic approaches change the set of data objects and the location of each object in SPM as program executes, and are the most flexible. Dynamic approaches for heap management, including ours, are mostly based on software caching, due to the high dynamism of heap data. Our approach is however different from the previous approaches. For example, Hallnor et al. proposed a approach that runs the replacement policy of level-two caches on software in a cache-based memory subsystems, while leaving miss handling to the hardware Hallnor and Reinhardt (2000). In contrast, our technique targets a system without any caches. Moritz et al. implemented a compiler-based software cache on a raw SRAM Moritz *et al.* (2001). The compiler statically groups memory accesses to data aggregates (arrays and data structures) into so called hot page sets, so that all the accesses to the same hot page set may share the same translation from a memory address to an SRAM address. The fast translation however relies on additional registers for each hot page set, while our technique does not require additional hardware. Chakraborty et al. proposed a compiler-based array management technique on SPM Chakraborty and Panda (2012). The technique statically analyzes a program and decides whether an array should be entirely copied into the SPM or dynamically managed by a software cache. The decision relies on the knowledge of array sizes, while our technique does not require such knowledge, and can be used to manage any data aggregates, such as linked lists or trees.

The state of the art dynamic heap management for SMM architectures from Bai et al. Bai and Shrivastava (2013) is the most relevant work to our technique. The state of the art implemented a 4-way set-associative software cache with first-in-first-out (FIFO) replacement policy on SPM. The details of the technique will be explained

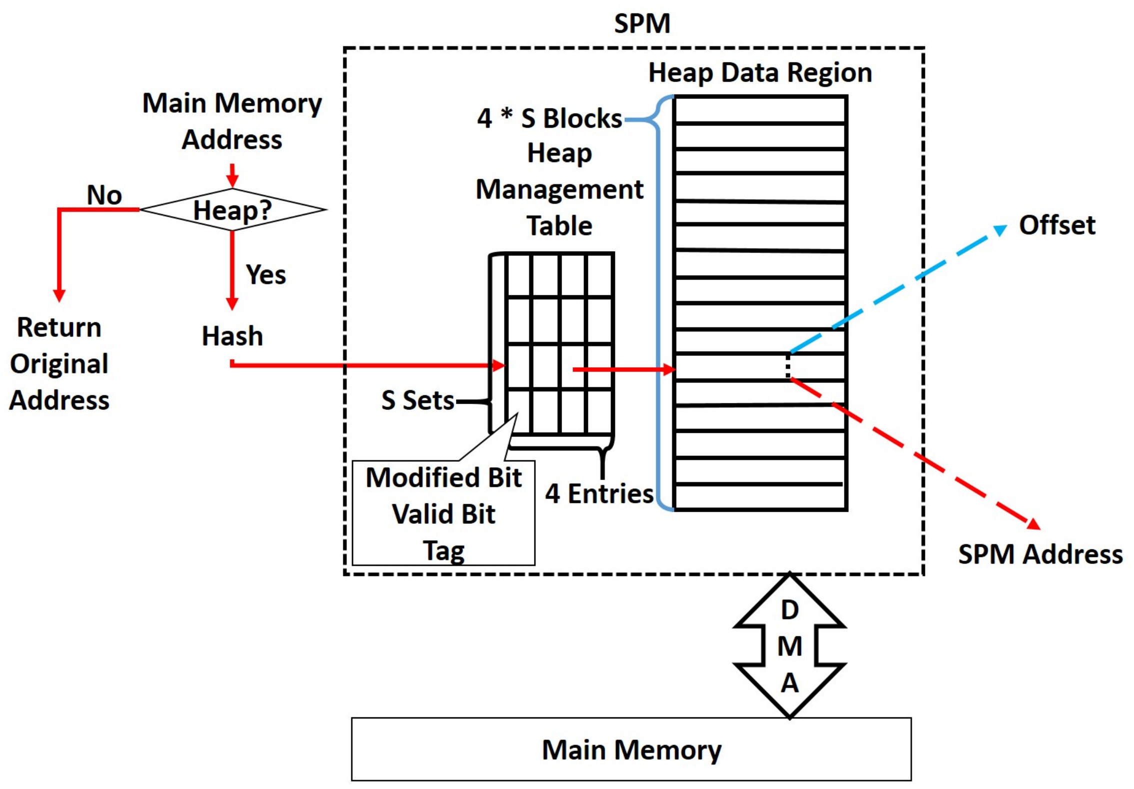


Figure 2.2: The State-of-the-art Heap Management Implements a 4-way Set-associative Software Cache on SPM.

shortly. However, it introduces very high management overhead. Our technique can significantly reduce overhead and improve performance.

Chapter 3

LIMITATION OF THE STATE OF THE ART

The state-of-the-art heap management Bai and Shrivastava (2013) emulates a 4-way set-associative cache on an SPM. The SPM is partitioned into a data region and a heap management table (HMT), as shown in Figure 2.2. The data region stores the actual heap data in fixed-sized blocks, while the management table stores a tag, 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 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

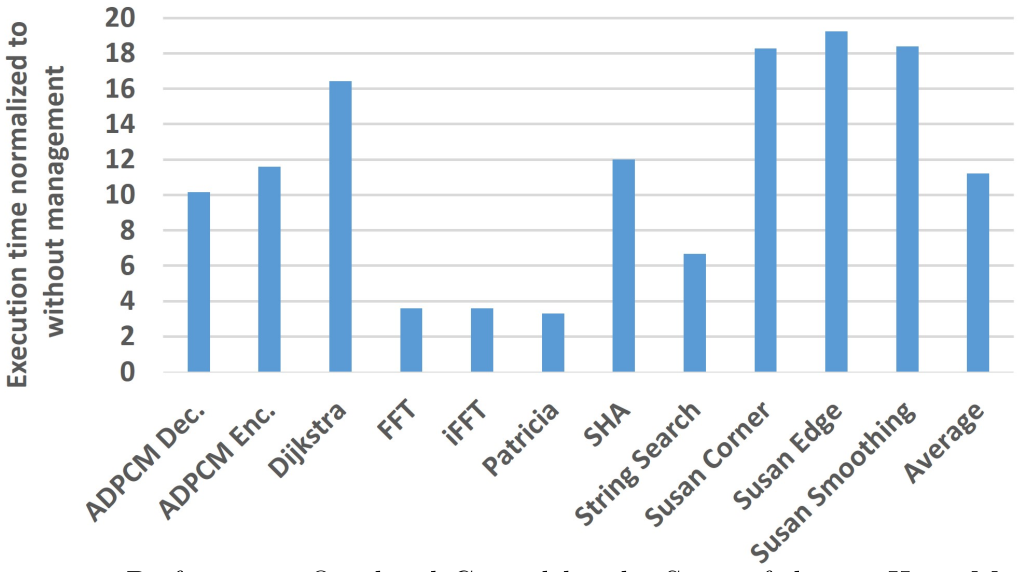


Figure 3.1: Performance Overhead Caused by the State-of-the-art Heap Management Approach.

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. If no available entry can be found in the set, the data block pointed by the victim index will be replaced by the new data block, and the corresponding entry in the management table is updated with the new tag accordingly. The evicted data block must be written back to the main memory if it has been modified. The victim index is increased by 1 and modulo 4 (the number of entries in each set). Eventually, the SPM address is calculated based on the set index, entry index and its offset within the data block, and used in the memory access.

Although the state of the art correctly manages the heap data of an application, it incurs high performance overhead. Figure 3.1 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) heap management function *g2l* called before each memory access. Since the management functions check if an access is to heap at runtime, it has to be called before each memory access (including those that are not to heap). The checking is expensive, not only because it happens at every memory access, but also because it involves branch operations, and potentially memory operations.

ii) set-associate heap management. The second major source of overhead comes from the fact the previous technique manages heap data in a set-associative manner. The software implementation of the set-associative structure 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 SPM address. The set index of the input main memory address is calculated with the following hash function:
$set\_index = ((mem\_addr >> log(block\_size)) \wedge (mem\_addr >> (log(block\_size) + 1)))\&(set\_num - 1)$,
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 as:
$spm\_addr = spm\_base + (set\_index * set\_assoc + entry\_index) * block\_size + mem\_addr \% block\_size$
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 a significant instruction overhead.

Chapter 4

KEY IDEAS OF OUR APPROACH

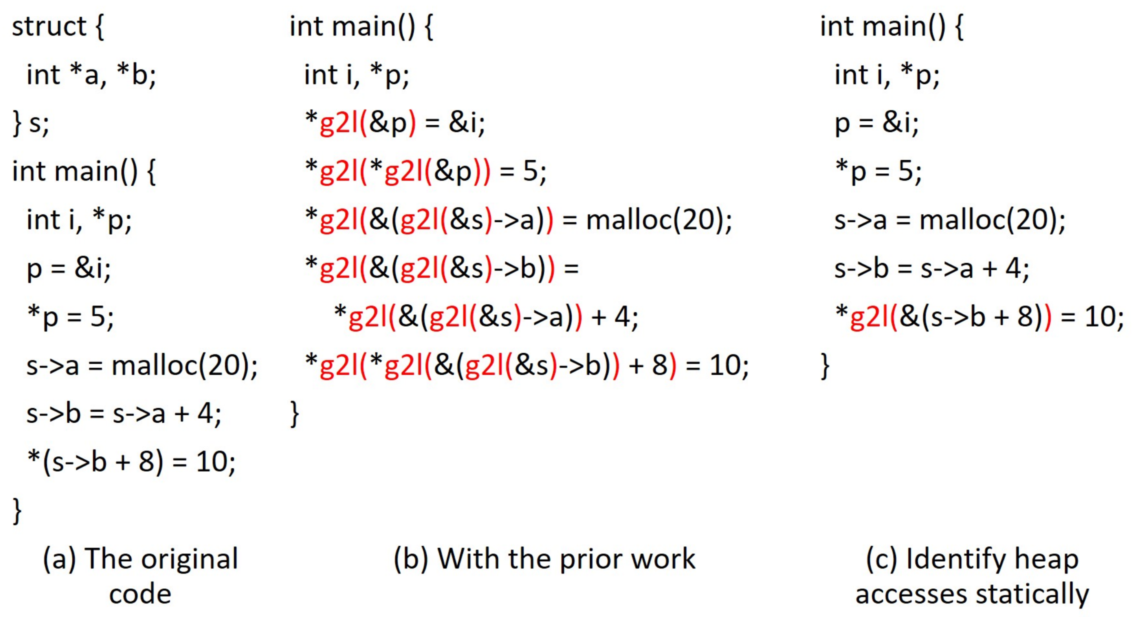
To solve the performance problems caused by the state-of-the-art approach, we use a series of optimizations that can greatly reduce the overhead of heap management on SMM architectures:

i) statically detecting heap accesses. This optimization identifies heap access at compile-time and eliminates heap management function *g2l* when the memory access is definitely not a heap accesses, and significantly reduces the number of (unnecessary) management calls at runtime. It also allows us to eliminate the runtime checking within the management function, if we know for sure that the memory access is a heap data access.

ii) simplifying management framework. We implemented a direct-mapped cache on SPM. In a direct-mapped cache, 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) inlining and combining management calls. Once *g2l* functions are inserted, we inline the function calls. We also remove the (redundant) heap management functions and execute them once before all the management calls. This optimization is particularly beneficial, when management functions are called within loop nests, and the common operations are hoisted to be outside of the loop nests.

iv) adjusting block size. All the aforementioned optimizations are generic, and do not require any profile information and thus are useful for all applications. However, for embedded systems, where profiling information can be useful, we further optimize heap data management by statically adjusting block size to avoid the type of cache misses an application primarily suffers from. 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. If the cache misses an application encounters are mostly conflict misses, we can reduce the block size to increase number of sets, so to lower the chances of mapping frequently-accessed memory addresses to the same SPM address. On other hand, if an application observes more cold misses, then we can increase the block size to refrain from such misses.



```
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;
}
```

```
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;
}
```

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

Figure 4.1: The Previous Approach Inserts *g2l* Before Every Memory Access, While Our Approach Tries to Identify the Heap Accesses Statically and Skip Unnecessary *g2l*s.

Chapter 5

DETAILS OF OUR APPROACH

We present the first three generic optimizations in detail in this section. The optimization for embedded systems that adjusts block size to reduce misses will be presented in the next section.

5.1 Statically Detecting Heap Accesses

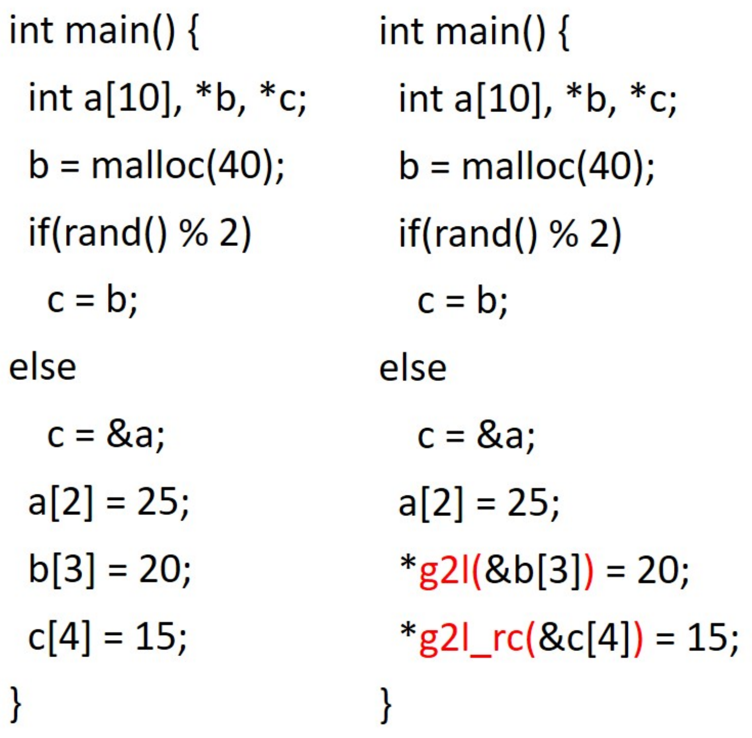
This optimization identifies heap accesses at compile-time, so that the management function *g2l* can be avoided at memory accesses that are definitely not to heap. Figure 4.1 illustrates the effect of this optimization.

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 Bai and Shrivastava (2013) will insert a *g2l* call at every memory access unnecessarily as shown in Figure 4.1(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 on 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 the pointers that directly point 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 as input the *main* function (line 2). The `getAlloc` procedure identifies all the invocation of memory allocators in the input function `F`, and record 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 stores 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 the *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 identified (line 3 to 5).

Once all the heap pointers are recognized, we can identify heap accesses and insert *g2l* function as follow. All the memory accesses (i.e. loads and stores) via any of the heap pointers identified in Algorithm 1 are considered as potential heap accesses. An *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.

There are cases when the compiler cannot determine whether a pointer refers to heap data. In Figure 5.1(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. We introduce a new management function, called *g2l*_`rc`, that checks at runtime and sees whether the memory address is in heap, similar to the previous work. When we are sure an access is to heap, we call the *g2l* function, which does not have any runtime checking. If an access may be to heap, we call *g2l*_rc. Otherwise, if we are sure an access is definitely not to heap, we do not call any heap management functions. Figure 5.1(b) shows the transformed code with heap management functions. We call *g2l* before accessing the data referred by the pointer `b`, because we are sure it is in heap. We call *g2l*_rc before accessing `c`, because it may refer to heap data, but are not for sure. We do not insert any heap management function when accessing `a`, because it is definitely in stack.



```
int main() {
  int a[10], *b, *c;
  b = malloc(40);
  if(rand() % 2)
    c = b;
else
    c = &a;
  a[2] = 25;
  b[3] = 20;
  c[4] = 15;
}
```

(a) Sample code

```
int main() {
  int a[10], *b, *c;
  b = malloc(40);
  if(rand() % 2)
    c = b;
else
    c = &a;
  a[2] = 25;
  *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

Figure 5.1: When a Memory Access May Be to Heap but Is Not for Certain, We Check at Runtime Before Managing the Access.

5.2 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 be noticeably simplified to execute much less instructions at run-time, compared to a set-associative cache. Figure 5.2(a) and Figure 5.2(b) shows two examples using the previous approach and our approach respectively. The edges in both figures specify dependencies between two steps. The previous approach as shown in Figure 5.2(a) calculate set index with the following function:

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

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 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 as

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

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. Notice this equation requires 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 5.2(a). On the other hand, our approach in Figure 5.2(b) simplifies the calculation of the set index of a memory address into $set\_index = mem\_addr >> log(block\_size)\%set\_num$,
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 $spm\_addr = spm\_base + mem\_addr\%(set\_num * block\_size)$.

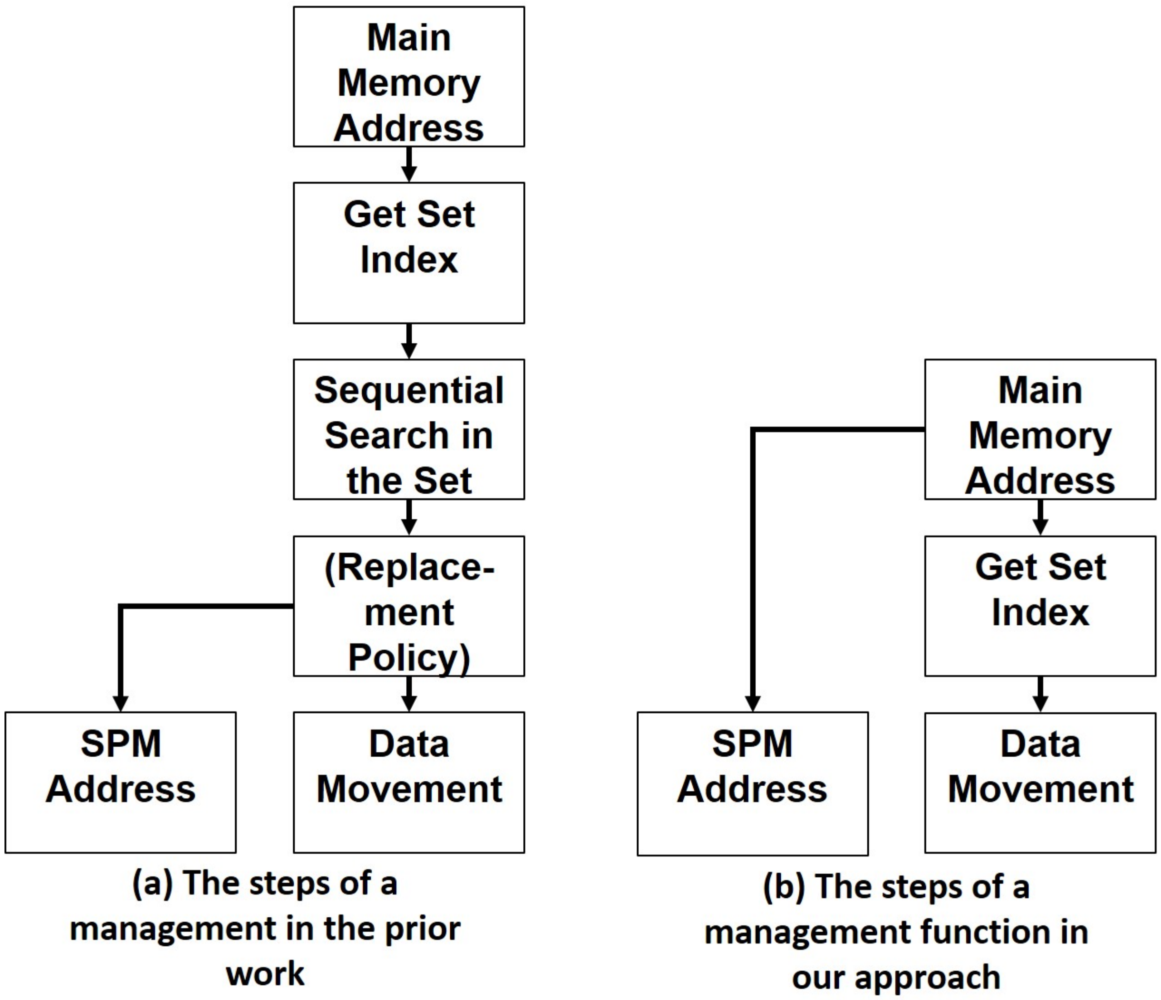
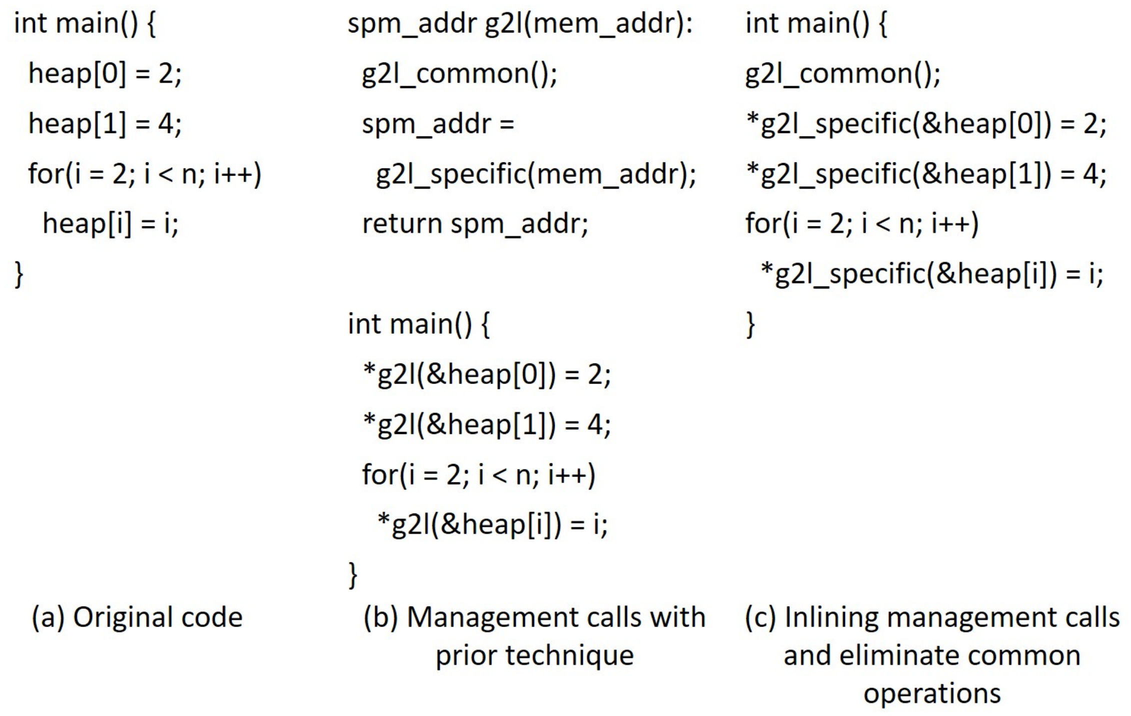


Figure 5.2: (A) the Steps of a Management in the Previous Work (B) the Steps of a Management Function in Our Approach (C) the Steps of a Management Function in the Previous Work and Our Approach.

In addition, the calculation of SPM address 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.

5.3 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 3 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



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

(a) Original code

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

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

Figure 5.3: Inlining Management Calls and Move Common Operations to the Beginning of the Caller Function.

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 5.3 illustrates the idea. Figure 5.3(a) shows the original code. Figure 5.3(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 $g2l_specific$ respectively in the example, but they are plain instructions in the actual implementation. In Figure 5.3(c), we inline the *g2l* calls, move and execute the common instructions at the beginning of the caller function. After the optimization, only call-specific instructions are executed at where a *g2l* had been called. While this optimization should definitely improve performance, its importance is maximized when *g2l* had been 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 outside the loop can significantly reduce such overheads.

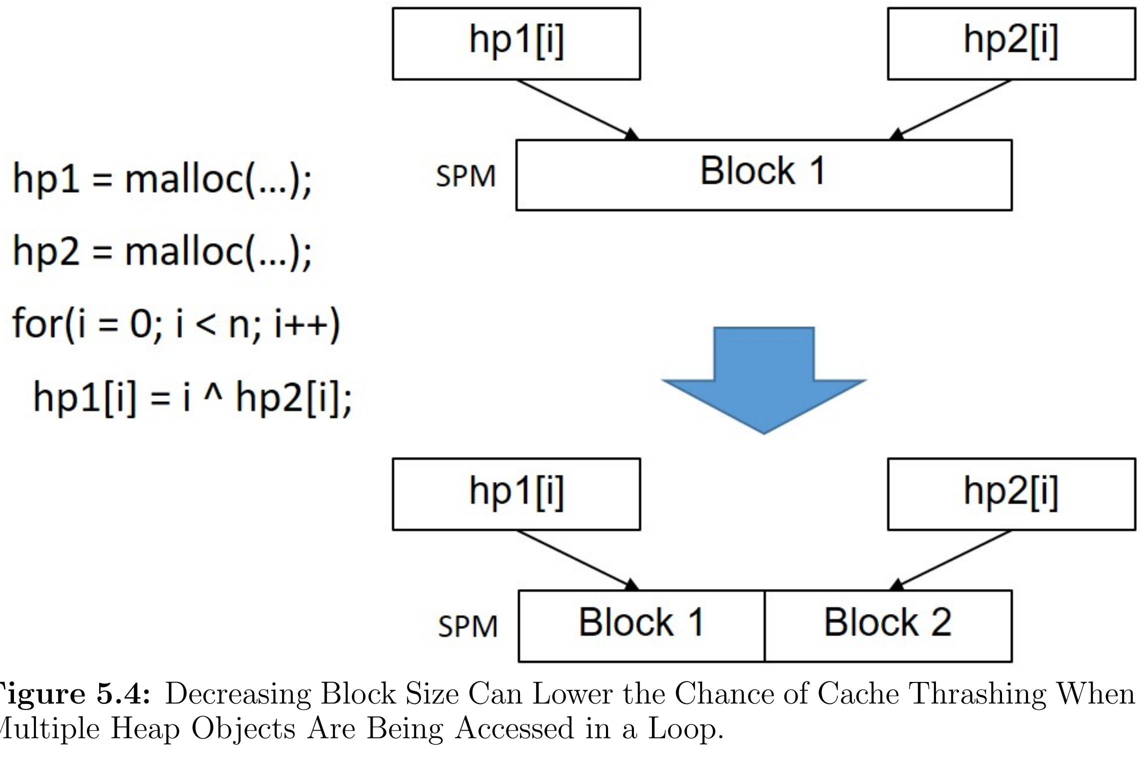
The algorithm of this optimization is shown in Algorithm 2. The compiler goes through every function in the program and inlines *g2l* calls with call-specific instructions. The common instructions are moved to the beginning of the function.

5.4 Adjusting Block Size for Embedded Applications

When the capacity and associativity of a cache are given, the block size decides the number of sets. Different choices of block size may end up causing drastically different performances. 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 decrease 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. Figure 5.4 shows an example. We assume direct-mapped cache in this example, therefore each set has only one block. Two heap objects of array type, `hp1` and `hp2`, are accessed in the same loop. The elements of the two arrays accessed in each iteration map to the same cache block and causes significant conflicts. In this case, if we decrease the block sizes and increase number of blocks, then the two accesses may map to different blocks thus different sets so that no conflicts will happen.

On the other hand, we can increase block size to improve spatial locality under certain circumstances. For instance, in Figure 5.5, only one array, `hp`, is accessed



```
hp1 = malloc(...);
hp2 = malloc(...);
for(i = 0; i < n; i++)
  hp1[i] = i ^ hp2[i];
```

Figure 5.4: Decreasing Block Size Can Lower the Chance of Cache Thrashing When Multiple Heap Objects Are Being Accessed in a Loop.

within a loop. Conflict misses will rarely happen in this case, so we can increase block size to have more elements and less data movement.

The heuristic we use to adjust block size at compile-time is presented in Figure 5.6 to statically adjust block size. The heuristic goes through all the innermost loops in a program. Whenever it identifies two or more heap objects 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. However, the choice of block size in both cases rely on profiling. Therefore, this optimization is the most effective for embedded applications using representative input.

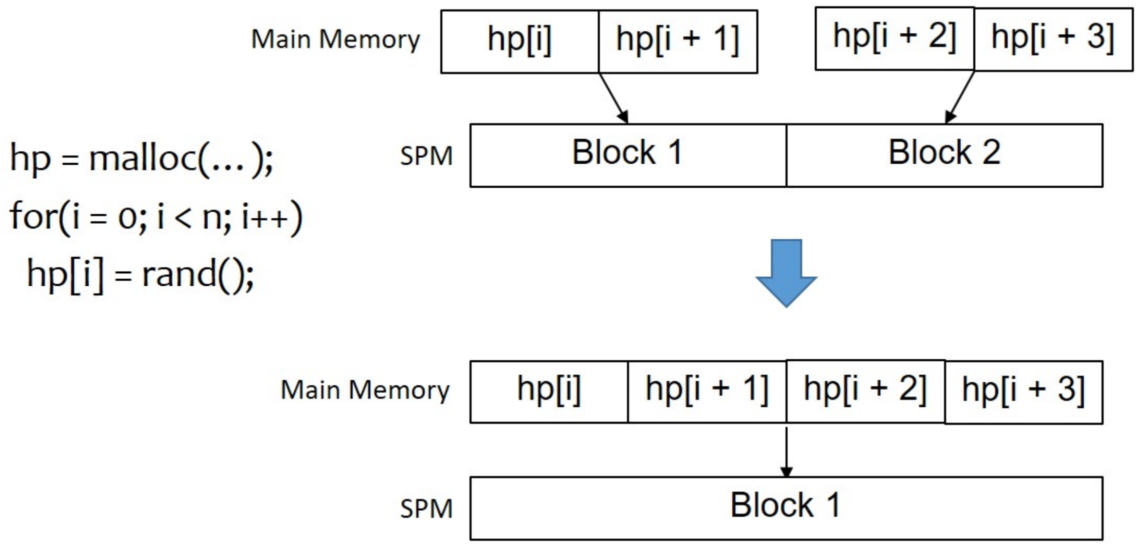


Figure 5.5: Increasing Block Size Can Improve Spatial Locality When Only One Heap Object Is Accessed in a Loop.

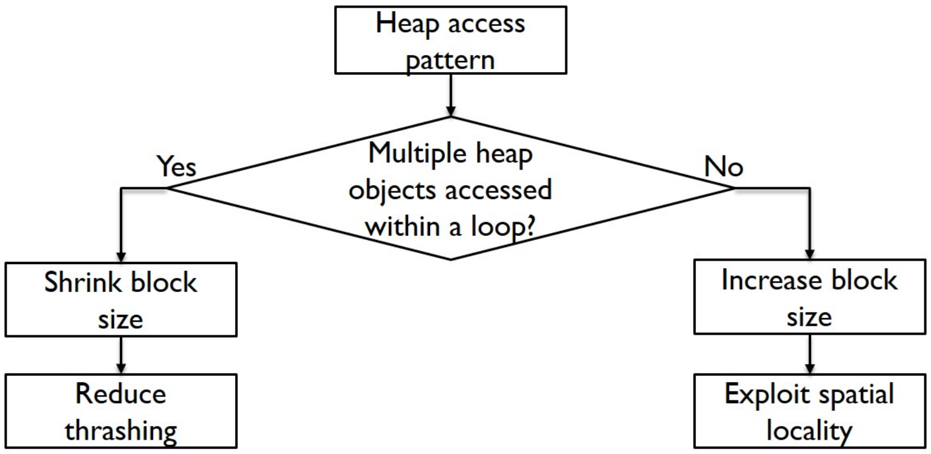


Figure 5.6: The Heuristic for Selecting a Block Size at Compile Time.

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 I in F do
8:         if I is a call to any memory allocator then
9:             Record destination pointer P as a heap pointer
10:        else
11:            if I is a call to any user function F' then
12:                getAlloc(F')
13: function GETALIAS(Function F)
14:     for each instruction I in F do
15:         if I is an assignment statement and one of the operands P is a heap pointer then
16:             Record destination pointer P' as an alias of P
17:         else
18:             if I is a call to any user function F' then
19:                 getAlias(F')
```

Algorithm 2 Inlining and combining *g2l* calls

```
1: function inlineManagementFunction(Function F)
2:     for each function F do
3:         if F has any call to $g2l$ then
4:             insert common operations of $g2l$ at the beginning of F
5:             for each $g2l$ call I in F do
6:                 inline the call
7:                 remove the common operations
```

Chapter 6

EXPERIMENTS

6.1 Experimental Setup

We implemented both the state-of-the-art technique Bai and Shrivastava (2013) and our technique as intermediate representation (IR) passes on LLVM 3.8 Lattner and Adve (2004) respectively.

We then compiled the same benchmarks with different heap management techniques, ran the executable code on Gem5 Binkert *et al.* (2011) and compared the performance. The block size in the software cache is set to 64 bytes in both tech-

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

Table 6.1: Maximum Heap Usage of Benchmarks.

niques by default. We only vary the block size during the fourth optimization that adjusts block size to reduce cache misses.

We emulated the SMM architecture on Gem5 by modifying the linker script and reserving part of the memory address space as the SPM. Gem5 is configured to have a single core with a single thread and run in system emulation mode. We implemented an DMA instruction that copies 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 Kistler *et al.* (2006).

We evaluated the proposed technique on benchmarks from Mibench benchmark suite Guthaus *et al.* (2001). Table 6.1 lists the maximum usage of heap size 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.

6.2 Significantly Reduces Execution Time

Figure 6.1 shows the execution time of our approach normalized to the previous work, when each of the optimization is incrementally introduced. Overall, our

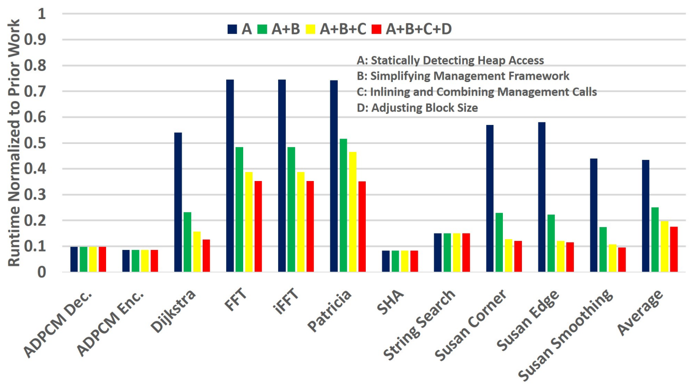


Figure 6.1: The Execution Time of Our Approach Normalized to the Previous Work with Optimizations Incrementally Added.

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.

Figure 6.2 shows the management instruction overhead of our approach normalized to the previous work, when each of the optimization is incrementally introduced. The first three generic optimizations, i.e. without adjusting block size reduces the management instruction overhead by 93%. When we apply all four optimizations, the execution time is reduced by 96% on average.

We can clearly see from the result in Figure 6.1 that 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, statically detecting heap access reduces the execution time by 57% on average. This is because of two reasons: reduced management calls and less executed instructions in each call. Table 6.2 shows the number of calls to the *g2l* function before and after statically detecting heap accesses in the previous work. The management calls is significantly reduced in all the benchmarks. For example, the number of management calls is reduced from 2628207 to 579221 in `Susan Edge`. In benchmarks that do not have any access, management calls are completely eliminated.

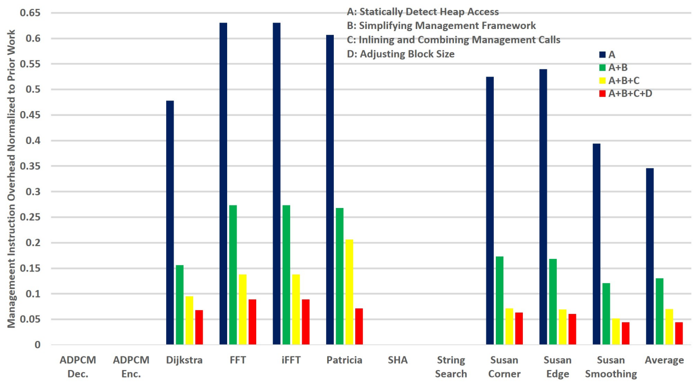


Figure 6.2: The Management Instruction Overhead of Our Approach Normalized to the Previous Work with Optimizations Incrementally Added.

Statically detecting heap accesses also allows us to eliminate runtime checking at *g2l*s, and thus reduces number of instructions executed in each *g2l*. Table 6.3 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 and an clean/unmodified data block is chosen to be evicted; a cache miss and a dirty/modified 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.

Benchmark	Before	After
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 6.2: Number of *g2l* Calls Called Before and After Identifying Heap Access Statically with the Previous Technique.

Case	Previous Work	Statically Detecting Heap Accesses	Simplifying Management Framework	Inlining and Combining Management Calls
read hit	52	46	19	8
write hit	59	53	23	10
read miss w/o write-back	145	139	41	36
write miss w/o write-back	145	139	44	37
read miss w. write-back	172	166	58	45
write miss w. write-back	172	166	58	45

Table 6.3: Instructions Executed Per *g2l* Under Different Cases With Optimizations Incrementally Added.

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 6.3 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 6.3 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 combining management calls can further reduce 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 6.3 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 combining management calls. Notice we apply this optimization after statically detecting heap accesses. So if the 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

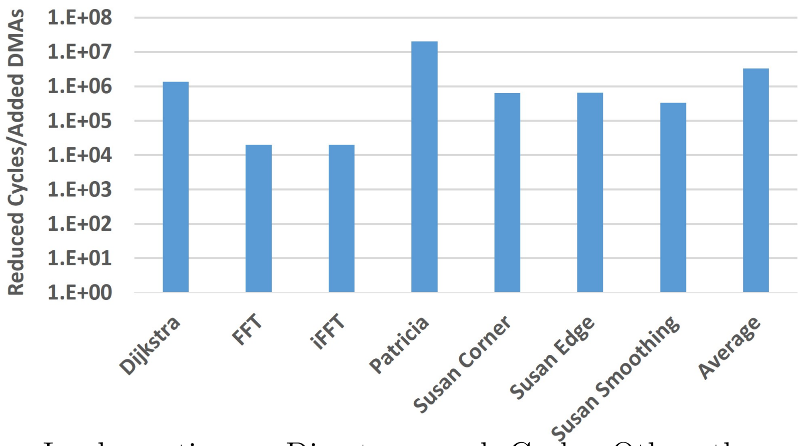


Figure 6.3: Implementing a Direct-mapped Cache Other than a 4-way Set-associative Cache Reduces More Execution Time Thanks to Simplified Management Functions, Compared to the Extra Time Introduced Due to Increased Cache Misses.

are no heap accesses in these benchmarks. The performance is therefore not further improved after the first optimization.

The block size is set to 64 bytes in the experiments of the previous three optimizations. We analyze programs and adjust block size in the fourth optimization. We set the block size to 16 bytes when the block size needs to be decreased, and 1024 bytes when the block size needs to be increased. The decision is based on profiling information. Adjusting block size can further reduce execution time by 11% (on top of the previous three optimizations).

6.3 Scales Well with SPM Size

In the above experiments, the SPM size is set to 4KB. Figure 6.4 shows the execution time before and after applying the three generic optimizations (excluding adjusting block size) to the previous technique, as the SPM size increases from 4KB to 64KB.

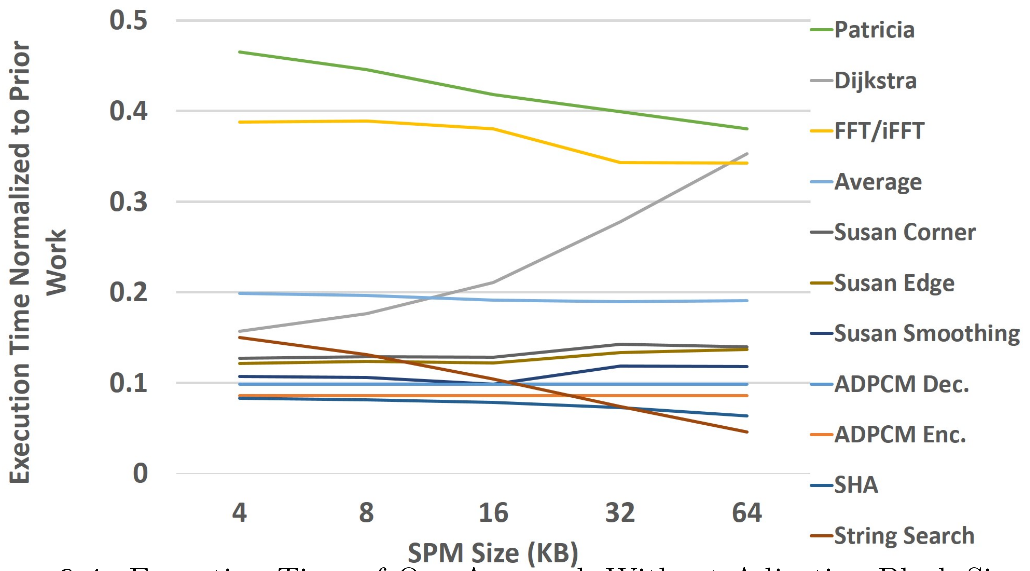


Figure 6.4: Execution Time of Our Approach Without Adjusting Block Size Normalized to the Previous Work, When the SPM Size Increases from 4KB to 64KB. The Order of the Names on the Right Side Is the Same as the Order of Endpoint of the Line for the Corresponding Benchmark.

Under any SPM size, our technique achieves significant improvement over the previous work. In addition, as the SPM size increases, the normalized execution time of most benchmarks decreases. The only exception is `Dijkstra`. The reason is because in our experiments, we assume the source code of library functions is not available. Therefore, we can not insert *g2l* function in the code of library functions. Notice this is a common problem for most if not all the compiler based approaches. As a result, all the modified data blocks in the software cache must be flushed to the main memory whenever a library function may modify heap data, to conservatively ensure the correctness of execution. In `Dijkstra`, `malloc` and `free` functions are extensively called. These two functions modify heap data, therefore we have to flush the software cache every time either of the functions is called. While both the previous technique and our technique suffer from the overhead, it is worse for our technique. The proof is briefly given as follows. When we flush a software cache, we have to check all the blocks in the cache and write back dirty data. Since the capacity and

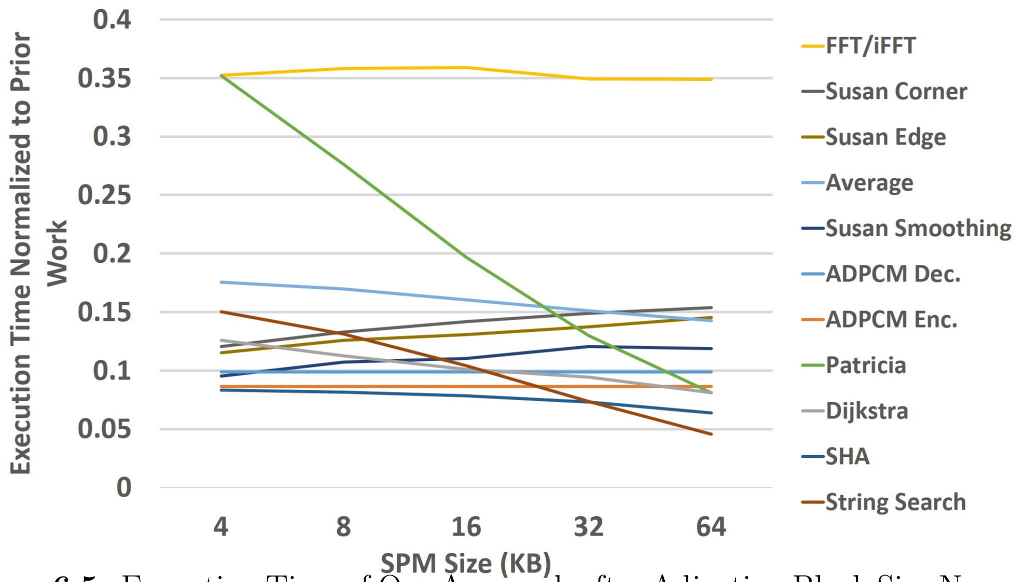


Figure 6.5: Execution Time of Our Approach after Adjusting Block Size Normalized to the Previous Work, When the SPM Size Increases from 4KB to 64KB. The Order of the Names on the Right Side Is the Same as the Order of Endpoint of the Line for the Corresponding Benchmark.

block size of the cache are the same for both techniques, the number of blocks to check are the same in both techniques. The overheads of flushing software cache are therefore roughly the same in both techniques. Let the time for flushing the software cache be T_{flush} in both techniques. Let the time for the rest of execution be T_1 with the previous technique, and T_2 with our technique. The normalized execution time can be calculated as $(T_{flush} + T_1)/(T_{flush} + T_2)$. As the cache size increases (the data block size remains), T_{flush} becomes higher, since the number of data blocks to check increases. Therefore, the normalized execution time becomes larger. For the similar reason, we can have the source code of library functions, we can eliminate the flushing in both techniques and the normalized execution time will be further reduced.

Figure 6.5 shows the execution time before and after applying all four optimizations (including adjusting block size) to the previous technique. Normalized execution time is further improved, as SPM size increases. Notice in this case the normalized execution time of `Dijkstra` decreases as the SPM size increases. It is because our technique increases the block size of the benchmark, and decreases the number of blocks to check thus lowers the overhead every time a flush happens, while the block size of the previous technique is not changed.

Chapter 7

CONCLUSION AND FUTURE WORK

Due to the expense of caches, some processor designers have opted to use SPM as an alternative. Such SPM-based processors have been widely used in various areas. However, the data management must be explicitly done on SPM. In this paper, we propose an efficient heap management that consists of three generic optimizations (statically detecting heap accesses, simplifying management framework, and inlining and combining management calls), and an additional optimization (adjusting block size) for embedded applications specifically. The experimental results show that with the three general optimizations, we can reduce the execution time by 80% on average compared to the state-of-the-art. If we apply all four optimizations, then we can reduce execution time by 83% on average. Additional experiments show that as SPM size increases, the execution time our approach normalized to the execution time of the state-of-the-art keeps reducing on average, with or without adjusting block size.

REFERENCES

Bai, K. and A. Shrivastava, “Automatic and efficient heap data management for limited local memory multicore architectures”, in “Design, Automation Test in Europe Conference Exhibition (DATE), 2013”, (2013).

Banakar, R., S. Steinke, B.-S. Lee, M. Balakrishnan and P. Marwedel, “Scratchpad memory: Design alternative for cache on-chip memory in embedded systems”, in “Proceedings of the Tenth International Symposium on Hardware/Software Codesign”, CODES ’02 (2002).

Binkert, N., 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).

Bournoutian, G. and A. Orailoglu, “Dynamic, Multi-core Cache Coherence Architecture for Power-sensitive Mobile Processors”, in “Proc. of CODES+ISSS”, (2011).

Carter, N. P., A. Agrawal, S. Borkar, R. Cledat, H. David, D. Dunning, J. Fryman, I. Ganev, R. A. Golliver, R. Knauerhase, R. Lethin, B. Meister, A. K. Mishra, W. R. Pinfold, J. Teller, J. Torrellas, N. Vasilache, G. Venkatesh and J. Xu, “Runnemede: An Architecture for Ubiquitous High-Performance Computing”, in “Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)”, HPCA ’13 (2013).

Chakraborty, P. and P. R. Panda, “Integrating software caches with scratch pad memory”, in “Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems”, CASES ’12 (2012).

Choi, B., 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).

Dominguez, A., S. Udayakumaran and R. Barua, “Heap data allocation to scratch-pad memory in embedded systems”, J. Embedded Comput. (2005).

Garcia-Guirado, A., R. Fernandez-Pascual, A. Ros and J. Garcia, “Energy-Efficient Cache Coherence Protocols in Chip-Multiprocessors for Server Consolidation”, in “Proc. of ICPP”, pp. 51–62 (2011).

Gschwind, M., H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe and T. Yamazaki, “Synergistic Processing in Cell’s Multicore Architecture”, IEEE Micro **26** (2006).

Guthaus, M. R., J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge and R. B. Brown, “Mibench: A free, commercially representative embedded benchmark suite”, in “Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on”, (2001).

Hallnor, E. G. and S. K. Reinhardt, “A fully associative software-managed cache design”, in “Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)”, (2000).

Kistler, M., M. Perrone and F. Petrini, “Cell multiprocessor communication network: Built for speed”, IEEE Micro (2006).

Lattner, C. and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”, in “Proc. of CGO”, (2004).

Lin, Y., 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).

McIlroy, R., P. Dickman and J. Sventek, “Efficient Dynamic Heap Allocation of Scratch-pad Memory”, in “Proceedings of the 7th International Symposium on Memory Management”, ISMM ’08 (2008).

Moritz, C. A., M. I. Frank and S. Amarasinghe, *FlexCache: A Framework for Flexible Compiler Generated Data Caching* (2001).

Niar, S., S. Meftali and J. L. Dekeyser, “Power consumption awareness in cache memory design with SystemC”, in “Proceedings. The 16th International Conference on Microelectronics, 2004. ICM 2004.”, (2004).

Olofsson, A., “Epiphany-V: A 1024 processor 64-bit RISC System-On-Chip”, CoRR (2016).

REX Computing, Inc., “THE NEO CHIP”, http://rexcomputing.com/ (2014).

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

Wilhelm, R., J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat and P. Stenström, “The Worst-case Execution-time Problem&Mdash;Overview of Methods and Survey of Tools”, ACM Trans. Embed. Comput. Syst. (2008).

Wilson, P. R., M. S. Johnstone, M. Neely and D. Boles, *Dynamic storage allocation: A survey and critical review* (1995).

Xu, Y., Y. Du, Y. Zhang and J. Yang, “A Composite and Scalable Cache Coherence Protocol for Large Scale CMPs”, in “Proc. of ICS”, (2011).