A Software-Only Scheme for Managing Heap Data on Limited Local Memory(LLM) Multicore Processors

KE BAI and AVIRAL SHRIVASTAVA, Arizona State University

This article presents a scheme for managing heap data in the local memory present in each core of a limited local memory (LLM) multicore architecture. Although managing heap data semi-automatically with software cache is feasible, it may require modifications of other thread codes. Crossthread modifications are very difficult to code and debug, and will become more complex and challenging as we increase the number of cores. In this article, we propose an intuitive programming interface, which is an automatic and scalable scheme for heap data management. Besides, for embedded applications, where the maximum heap size can be profiled, we propose several optimizations on our heap management to significantly decrease the library overheads. Our experiments on several benchmarks from MiBench executing on the Sony Playstation 3 show that our scheme is natural to use, and if we know the maximum size of heap data, our optimizations can improve application performance by an average of 14%.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Code generation, Compilers, Optimization*

General Terms: Algorithm, Design, Experimentation, Performance

Additional Key Words and Phrases: Heap data, local memory, scratch pad memory, embedded systems, multicore processor, IBM Cell BE, MPI

ACM Reference Format:
Bai, K. and Shrivastava, A. 2013. A software-only scheme for managing heap data on limited local memory (LLM) multicore processors. ACM Trans. Embedd. Comput. Syst. 13, 1, Article 5 (August 2013), 18 pages.
DOI: http://dx.doi.org/10.1145/2501626.2501632

1. INTRODUCTION

Scaling the memory architecture is one of the toughest and the most significant issue as we evolute from multicore (few cores) to many-core (thousands of cores). Meanwhile, providing the illusion of a singe unified memory space in hardware is becoming expensive for two main reasons: (i) automatically managing the memory in hardware, that is, by caches, becomes prohibitive, since it has higher power and performance overheads. Caches have already consumed about half of the processor energy on single-core processors [Banakar et al. 2002] and are expected to consume more as the number of cores increase. (ii) Cache protocols are not well scalable to many cores [Eichenberger et al. 2006]. As a result, limited local memory (LLM) multicore architecture with a small local memory on each core is coming up as a promising scalable memory architecture. Modern and futuristic processors, especially in the embedded domain, are

This research was partially funded by grants from National Science Foundation CCF-0916652, IIP-0856090, and NSF I/UCRC for Embedded Systems.
Authors' addresses: K. Bai and A. Shrivastava, Compiler and Microarchitecture Laboratory, Arizona State University, Tempe, AZ 85281; corresponding author's email: ke.bai@asu.edu.

DOI: http://dx.doi.org/10.1145/2501626.2501632

being designed in LLM multicore architectures. One such example is the IBM Cell BE [Flachs et al. 2006].

In an LLM multicore processor, each core can only directly access its local memory, and programmers are responsible for manually adding DMA commands for data transfers between the global memory and the local memory. For example, each Synergistic Processing Element (SPE) on the IBM Cell BE can only access its local memory, and this local memory is shared by all code and data of the thread mapped to the core. When developing applications on LLM multicore architectures, there are always two challenges. The first is that programmers must parallelize the given application at several levels, for example, thread level or data level. The second is that each thread on each core should be executed efficiently. When a thread can not be mapped to a core, programmers must change the way the application is parallelized. This can be extremely complex, because often applications have some natural parallelism themselves, and finding out some other ways to parallelize them can be formidable. Therefore, we primarily cope with the second challenge of executing (and efficiently executing) a thread of application on a core.

If all the code and data of an application can fit into the local memory, extreme efficiency is achieved—this is the promise of LLM multicore architectures. However, if they can not completely fit into the local memory, DMA commands must be inserted to bring the required data to the local memory before it is used and move some not-so-urgently-needed data back to the global memory. With limited memory resource in the local memory of LLM architectures, all code and data of a thread must be managed. However, managing heap data is especially important. First, it is dynamic in nature, and the size is always data dependent and can be unbounded. Second, in most systems, heap and stack grow towards each other and could easily overwrite each other. The gentle failure is that the application will crash or go to an infinite loop. The severe failure is that the program gives a wrong result without the awareness of the programmer. In fact, the Cell Programmer's Guide suggests to "avoid using heap variables". We believe this will extremely restrict a programmer's productivity and creativity. Consequently, we need a scheme to efficiently manage heap data in a constant and small amount of space in the local memory.

One way to semi-automatically manage heap data in the local memory on each core of an LLM multicore processor is through the use of the software cache [Angiolini et al. 2004]. Software cache is essentially software implemented in some data structures. Global data can take advantage of software cache better than other data types, since it is declared and allocated once. Heap data, which is dynamically allocated, can not use software cache directly. In fact, managing heap data of an application thread with software cache requires several non-intuitive and error-prone modifications in the application code. It requires not only the modification of the execution thread, but also several changes of the main thread. Namely, the user must create a new thread on the main core that listens to memory requests from the execution core. As the number of cores increases, this solution becomes more complicated to be implemented and debugged.

This article proposes a scheme for hiding the programming complexity in a library with simple programming interface. We modify the GCC compiler for IBM Cell BE to automate insertions of library functions, compile benchmarks from MiBench [Guthaus et al. 2001] and others using it, and then measure the runtime on the Sony Play Station 3. Our experiments show that our heap management scheme, while being transparent to programmers, performs on par with a software cache implementation. When the maximum heap size of the application can be known, our optimizations can improve application speeds by an average of 14%.

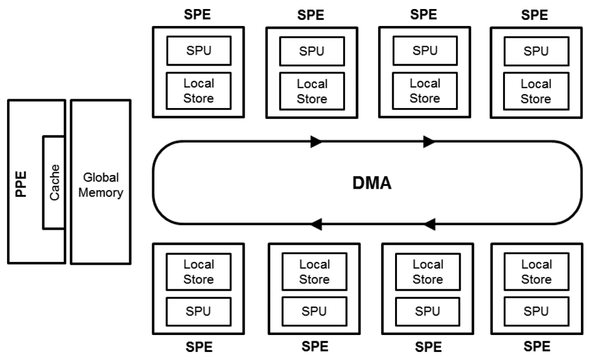

Fig. 1. The IBM Cell BE is a good example of limited local memory (LLM) architectures. There are eight synergistic processing elements, or SPEs. Each SPE can access only a small local memory, and all data transfers between the local memory and the global memory take place through explicit DMA calls.

2. BACKGROUND

2.1. Limited Local Memory Architecture

The IBM Cell Broadband Engine [Flachs et al. 2006] is a very good example of limited local memory (LLM) multicore architectures. As shown in Figure 1, it is a nine-core architecture, with one *main core* (the power processing element, or PPE, in the IBM Cell BE) and eight distributed *execution cores* (the synergistic processing elements, or SPEs, in the IBM Cell BE). The main core in the Cell BE is a two-way simultaneous multithreaded power 5 core, while each of the execution cores works on only one thread at a time in a non-preemptive fashion. Only the main core has an operating system, and it has direct access to the global memory through a coherent L2 cache. Each execution core has a 256 KB local memory and can not directly access the global memory and other local memories. Data communications between the local memory and the global memory should be explicitly managed in the software through the direct memory access (DMA) engine.

2.2. Thread-Based Programming Paradigm

Programming on an LLM multicore architecture is based on a Message Passing Interface (MPI) style *thread model*. It requires programmers to have a *main thread*. This main master thread is responsible for creating, distributing data and tasks, and even collecting results from *execution threads*. The main thread runs on the main core, while the execution threads are scheduled on execution cores. A very simple application in this multicore programming paradigm is illustrated in Figure 2. In the pseudocode, the main thread, executing on the main core, initiates several execution threads on the execution cores. In the execution thread, N number of ITEM data structures are initialized and accessed. ITEM data structures contain two fields, *id* (int) and *price* (float) for each item.

3. MOTIVATION

Generally, the local memory on the execution core is conceptually divided into four segments by the compiler: text region, global data region, heap data region, and stack data region. The text region is where the compiled code of the program itself resides. Function frames reside in the stack region, starting from the top of the memory and growing downwards, while heap variables (defined through *malloc*) are allocated in the heap region, starting from the top of code region and growing upwards. The four segments share the limited memory resource of local memory. Because the local memory

```
main(){
  for(speID=0;speID<NUM_SPEs;speID++){
    init_SPEs(speID);
  }
}
```
(a) PPE code

```
typedef struct{
  int id;
  float price;
} ITEM;

main(){
  for(i=0;i<N;i++){
    item[i] = malloc(sizeof(ITEM));
    item[i].id = i;
    printf("%d\n", item[i].id);
  }
}
```
(b) SPE code

Fig. 2. Outline of a threaded program on the Cell BE: (a) PPE creates a thread on each SPE; (b) on each SPE, some ITEM structures are allocated and accessed.

lacks any hardware protection, heap data can easily overflow into the stack region and corrupt the program state.

In Figure 2, for small N, the program will execute correctly, but large values of N can cause catastrophic failures, for example, the application crashes, the execution core goes into an infinite loop. However, the worst situation is that the output is just slightly incorrect. One way to avoid these problems is to avoid using heap variables; however, we believe that this is very limiting on both the creativity and the productivity of programmers. What is needed is a scheme that limited local memory multicore programmers can use to efficiently and automatically manage heap data of the application.

4. RELATED WORK

Local memory in each core of an LLM multicore architecture is a raw memory under software control. They are very similar to the Scratch Pad Memories (SPMs) popular in embedded systems. Banakar et al. [2002] proposed the use of raw memories in embedded systems when they noticed that caches consume a very significant portion of the power budget of even an embedded processor, like the Intel StrongARM [Montanaro et al. 1997]. They demonstrated that for the same memory area, SPMs consume 40% less energy and 34% less die area. However, the absence of memory management logic in the hardware shifts the burden of managing data to programmers.

Techniques have been proposed to manage code [Steinke et al. 2002a, 2002b; Angiolini et al. 2004; Verma et al. 2004, 2005; Nguyen et al. 2005; Egger et al. 2006a, 2006b; Udayakumaran et al. 2006; Janapsatya et al. 2006; Verma and Marwedel 2006], global data [Kandemir et al. 2001, 2002; Steinke et al. 2002b; Avissar et al. 2002; Verma et al. 2005; Li et al. 2005; Udayakumaran et al. 2006; Verma and Marwedel 2006] and stack data [Avissar et al. 2002; Francesco et al. 2004; Nguyen et al. 2005; Li et al. 2005; Udayakumaran et al. 2006] on the SPM, but little work has been done towards managing heap data [Francesco et al. 2004; Dominguez et al. 2005; McIlroy et al. 2008], not even to techniques for LLM multicore architectures.

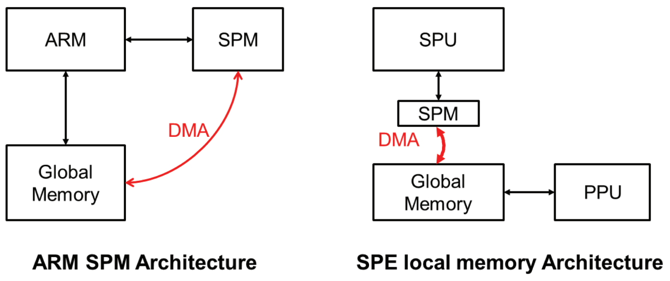

Fig. 3. In the ARM processor, the SPM is in addition to the regular cache hierarchy, while in the Cell BE, local memory/SPM is an essential part of the memory hierarchy.

Che et al. proposed techniques for compiling stream programs for IBM Cell BE [Che et al. 2010; Che and Chatha 2010, 2011a, 2011b]. We have already proposed schemes for managing code [Pabalkar et al. 2008; Jung et al. 2010], stack data [Kannan et al. 2009; Bai et al. 2011b], and heap data (in the form of *vector* for C++ language) [Bai et al. 2011a] for LLM multicore architectures. This work only focuses on managing heap data on local memories of LLM multicore processors, and fundamentally differs from the existing work on SPMs. The difference originates from the use of SPMs in embedded systems and local memories in LLM multicore processors. Figure 3 illustrates the difference. It shows that in the embedded systems, for example, the ARM architecture, the SPM is present in addition to the regular cache hierarchy of the processor. Programs could execute correctly without the use of SPM; they could however use SPM to improve power and performance. On the other hand, in the synergistic processing element (SPE) of IBM Cell BE, all code/data must go through the local memory. In other words, while the problem of using SPMs in embedded systems is that of optimization, the problem of using local memory/SPM in distributed memory multicore processors is to enable the execution of applications. As a result, previous SPM researches [Francesco et al. 2004; Dominguez et al. 2005; McIlroy et al. 2008] have focused on the question of "what to map" on the SPM. The "what to map" is not even an option for LLM multicore processors. Important questions in using local memories are (i) given that the application code has to be changed to make itself work, what would be a set of intuitive and simple APIs that must be inserted to the application program to make this happen, and (ii) these changes should eventually result in a smaller number of coarse-grain communications between the local memory and the global memory.

One way to manage heap data on the local memory of each core in an LLM processor, for example, the IBM Cell BE, is by using the software cache. However, there are several limitations in managing heap data through the software cache. We will discuss this approach and its limitations in greater detail in Section 5, describe our approach to meet this challenge in Section 6, and finally experimentally demonstrate the need and usefulness of our approach in Section 8.

5. HEAP DATA MANAGEMENT USING SOFTWARE CACHE

Software cache [Angiolini et al. 2004] is a software-managed data structure located in the global data segment of the local memory in each execution core. It can be used as a semi-automatic method to manage large amounts of data in a programmer-defined size. Before using software cache, the programmer needs to make some configurations for software cache, and then replaces every access of that data with a read/write from/to the software cache. For each read and write, the software cache first checks whether the data is in the cache or not. If it is, the program can directly read/write the data from/to the cache; otherwise, a direct memory access (DMA) is performed to get the

```
heapManage(){
  while(1){
    spe_out_mbox_read(speID, size ...);
    ppeAddr = malloc(size);
    spe_in_mbox_write(speID,&ppeAddr
        ...);
  }
}

main(){
  for(speID=0;speID<6;speID++){
    init_SPEs(speID);
  }
  /* Create a new thread*/
  pthread_create(..&heapManage..);
}
```

(a) PPE code

```
#define CACHE_NAME HEAP
                    ...
typedef struct{
  int id;
  float price;
}Item;

main(){
  for (i=0;i<N;i++){
    spu_write_out_mbox(sizeof(Item));
    item[i] = spu_read_in_mbox();
    cache_wr(HEAP, item[i].id, i);
    printf("%d\n",cache_rd(HEAP, item[
        i].id));
  }
}
```

(b) SPE code

Fig. 4. Using the software cache to manage heap data in the IBM Cell BE. Important observations: (a) the SPE must transmit all its memory management functions to the PPE; (b) an extra memory management thread is needed on the PPE for communications between PPE and SPE.

required data from the global memory to the local memory, and then the data can be used. As new data comes into the cache data structure, older data may be evicted out to the global memory.

We modified the application code described in Figure 2 to manage heap data through software cache. Note that the software cache is designed for managing large amount of data in the local memory and there is no published scheme for the purpose of managing heap data through software cache. We made this try to manage heap data in this way. In Figure 4(a), we can note that there is a separate thread, *heapManage*, created by the main thread. It waits for requests from the execution thread/core (SPE), allocates the requested data structure in the global memory, and sends back the allocated address to the execution thread/core (SPE). For the execution thread in Figure 4(b), the first line declares a software cache named HEAP. More configurations for cache are needed, for example, cache size and associativity, but are skipped here for succinctness. Since the number of items N can be large, depending on the N set by the programmer, the item data structures must be allocated in the global memory. However, the ITEM data structures are *malloc-ed* in the execution thread in the original code. Therefore, we need to use *heapManage* in the main thread/core (PPE) to take care of memory requirements from the execution threads/cores to the main thread/core. In the example shown in Figure 4, the execution thread/core (SPE) sends the size of *malloc* to the main thread/core (PPE) through mailbox. The main thread/core (PPE) allocates space for the ITEM data structure(s) in the global memory and sends its address back to the execution thread/core (SPE). The execution core (SPE) can then use this address to access the ITEM data structure that actually resides in the global memory, through the software cache. Similar steps need to be taken when *freeing* up the allocated memory, but are skipped for simplicity in the example. Some of the complexities and disadvantages of managing heap data through software cache are as follows.

(1) The data is required to be allocated on the global memory by the software cache, and each data access of execution cores must through a global address. To use the software cache for heap variables, each allocation or deallocation request in the execution thread/core must be transmitted to the main thread/core. This communication task should be manually accomplished by programmers. In addition, to

```
typedef struct node
{
   int weight;
   struct node *link;
}
(a) Original Struct
```

```
typedef struct node
{
   int weight;
   int link;
}
(b) Modified Struct
```

Fig. 5. Users need to change the pointer type to any other non-pointer/non-structure type in order to use it in software cache, for example, here we change it to *int*.

allow the main thread to do some other parallel tasks during the execution time of execution cores, programmers should create a new thread, which waits and serves all the requests from execution threads. Another aspect of managing heap data through software cache is that normally the main core is the master core which is responsible for distributing tasks and data, but now the main core has to serve requests from each execution core. This reversal of roles makes this programming non-intuitive and complicated.

(2) The interface of the software cache only supports one data type in a declared cache. Given one example of a data structure shown in Figure 5, we will learn how complicated using the software cache is. The structure *node* contains two elements, weight and a pointer to a similar structure. One thing that should be noted is that the software cache does not support *pointer* type elements, and it must be renamed as any other non-structure and non-pointer data type. For example, we can change this element to *integer* type for the purpose that two elements can use the same cache instead of two different caches. This is unnatural for C programming and severely reduces readability.

(3) Managing heap data through the software cache requires users to replace each data access with a cache read/write operation. Hence, even if we know that the data is in the cache, we still need to use cache functions *cache_rd* and *cache_wr* to access data from software cache. We can not avoid looking up the cache table, and therefore there is little scope for reducing the management overhead.

6. OUR APPROACH

6.1. Overview

The objective of our approach is to hide the additional complexity in managing heap memory in a limited space on the local memory. The heap management library should be intuitive to use and can be automated with the help of a compiler. In our scheme, library functions can handle the data type issue, which is one limitation of using the software cache. Our library does not require users to declare different regions for different data types, and it also can support different types of pointer values. Figure 6 shows the pseudocode of how to use our heap management on the example shown in Figure 2. Note that the heap is declared and allocated/freed only on the execution thread/core (SPE). Since heap memory allocation and deallocation thread is one of our heap management library, programmers do not need to write the extra thread on the main core (PPE). Therefore, the main thread does not change at all. In addition, users do not need to consider the redistribution of heap data; they can continue to program as if each execution core has enough memory to manage (almost) unlimited heap data. They even do not need to insert the function *_p2s* before and the function *_s2p* after any access to heap variables with our modified GCC compiler. In addition, we also expose both global addresses and local addresses, therefore we do not need to perform checking every time, as one of the disadvantages of using software cache.

```
/*PPU heap region*/
int ppe_heap[MAX] ;


main(){
  for(speID=0;speID<NUM_SPEs;speID
      ++){
    init_SPEs(speID);
  }
}
```

(a) PPE code

```
main(){
  for(i=0;i<N;i++){
    item[i] = malloc(sizeof(Item));

    item[i] = _p2s(item[i]);
    item[i].id = i;
    printf("%d\n",item[i].id);
    item[i] = _s2p(item[i]);
  }
}
```

(b) SPE code

Fig. 6. Using our approach to manage heap data. (a) We redefine *malloc* and *free* on the SPE to automatically interact with the PPE. (b) Our modified GCC compiler automatically inserts a call to *_p2s* function before and a call to *_s2p* function after accessing each heap variable.

6.2. Application Programming Interface (API)

The fundamental challenge in limited local memory (LLM) multicore architectures is that every variable can have two addresses, a *global address* and a *local address*, depending on where the variable is located. The software cache hides local addresses to programmers. It only exposes global addresses of variables, and users must use them to access variables through the software cache. The interface of the software cache simulates the functionalities of the cache in the cache-based architectures. However, it requires the address translation every time when the variable is accessed, and therefore incurs high overhead. To solve this problem, our heap data management approach exposes both addresses of variables to programmers. With the local address, users can directly access the variable and do not need to perform the address translation every time. If a required variable is not in the local memory, the library function *_p2s(global address ga)* brings it from the global address *ga* to the local memory and returns the local address *la* of the variable. The counterpart functionality is encapsulated in the function *_s2p(local address la)*. Besides introducing two newly implemented functions, we also re-implement two existing functions, *malloc* and *free*. If there is enough memory space in the heap region of the local memory, the *malloc* function can directly return a pointer. Otherwise, the function will first evict the oldest heap variable(s) to the global memory to make sufficient space for the coming heap variable, and then return a pointer. One important point to note here is that even if the *malloc* function may allocate space from the local memory, it still returns the global memory address of the allocated heap variable each time. This is because different heap variables can have the same local memory address, but definitely have unique global memory addresses. Thus, we should always access heap variables through global addresses. The *free* function also uses the global address of the variable.

6.3. Implementation Details

We expose both global addresses and local addresses to programmers. Therefore, our library keeps a mapping between these two addresses in a data structure called the heap management table.

6.3.1. Bookkeeping Data Structure – Heap Management Table. Each entry of the management table consists of the following information (it is not the real data structure for each entry).

For each heap variable, its size should be kept in *_chunkSize*. The size can be one heap object or any other sizes, depending on the granularity. For two different addresses, we have *_speAddress* and *_ppeAddress*, accordingly. One other important information

```
_HMDSEntry{
    _chunkSize,   /* size of heap object */
    _speAddress,  /* local memory address */
    _ppeAddress,  /* global memory address */
    _isFree,      /* indicates whether this heap was freed */
    _valid,       /* indicates whether the entry is valid */
    _inSPM,       /* indicates location of heap variable */
    _timeStamp,   /* for LRU replacement policy */
}
```

is the location of the heap variable, and therefore *_inSPM* is used. If the heap data is in the local memory, it is set to 1; otherwise, it is 0. Again, heap data is dynamic in nature and it can be allocated and deallocated at any time. This is the reason why we introduce *_isFree*. The 1 indicates this heap variable is freed. When the space in the local memory for heap is not sufficient, some old heap objects should be evicted to the global memory. We implement LRU replacement policy for our heap management, since the access pattern of the heap data is complex in its analysis. *_timeStamp* shows when this heap variable starts to be located in the local memory. Finally, because the heap management table might be accessed frequently, it is beneficial to keep more valid entries at a given table size in the local memory. It can reduce data transfers between the global memory and the local memory. Two ways can be used to achieve this objective. The first option is to reuse the table entries. This can be achieved by the flag *_valid*. The 0 of the flag means this entry can be reused for other heap objects later. The second choice is to keep every entry of management table as small as possible. One thing that should be emphasized is that the fields previously shown are not the real elements in each entry of our heap management table. We only use 18 bits for *_speAddress*, 1 bit for *_valid*, and 1 bit for *_inSPM*. In total, each entry occupies 16 bytes.

6.3.2. Interaction between Functions and Heap Management Table. The *malloc* function adds a new entry for every allocated heap object to the heap management table (HMT). Reversely, *free* may result in the removal of an entry in the table. Both the function *_p2s* and the function *_s2p* access HMT when they are called every time. *_p2s* takes in the global address *ga*, and then uses it to look up the table to find the right entry E by checking *ga* with all global addresses in HMT until it is found. From the element *_inSPM* in E, we know where the heap data is locating. If it is in the local memory, the function just returns *_speAddress* in E. Otherwise, *_p2s* looks up the table again to find the oldest heap data h in the local memory and evicts it to its global address ga'. After the eviction and the updating of *_inSPM* for h, the right data located in *ga* will be fetched to the local address *la* of h, and finally *la* is returned and *_timeStamp* is updated. Contrarily, the process in *_s2p* is simpler. It only maps the local address *la* back to its corresponding global address *ga*. It is done by checking the HMT until the local address in that entry matches the *la* and the *_inSPM* indicates the heap data is in the local memory.

6.4. Global Memory Management

In order to support (almost) unlimited heap memory, we have to manage heap data and the heap management table in the global memory dynamically. This essentially requires a separate memory management thread running on the main core. Our implementation is similar to the one described in Section 5, however, this separate thread is a part of the library in our implementation, and the user does not need to explicitly write it. In Figure 7, we can see the functionality of our global memory management thread. When the execution thread wants to put some old heap data to the global

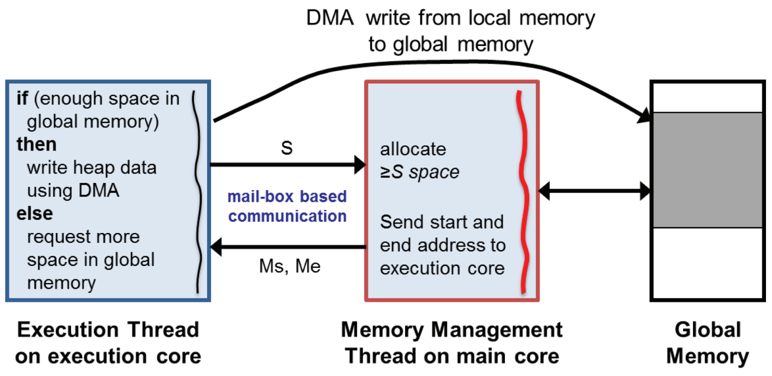
DMA write from local memory to global memory
if (enough space in global memory)
then
write heap data using DMA
else
request more space in global memory
S
mail-box based communication
Ms, Me
allocate ≥S space
Send start and end address to execution core
Execution Thread on execution core
Memory Management Thread on main core
Global Memory

Fig. 7. Global memory management thread on PPE.

memory, it first checks whether there is enough space there. If there is enough space in the global memory, it can directly and efficiently leverage DMA to transfer the data. Otherwise, there must be a memory request from the execution thread. The request is achieved through mailbox facility provided by the IBM Cell SDK. Once the memory management thread receives the size of space, it allocates the memory in the main core and returns the start and end addresses of the allocated space to the execution core. At this point, the DMA can be used to evict contents from execution core to main core. As for *free()* function, the execution core sends a signal and the memory size to the main core, and the global memory management thread will free that size of memory.

With global memory management thread, we solve the problem of "how to send" and "where to send". However, another consideration is "how much to send". In our heap management implementation, we introduce the concept of granularity. The unit of data transfer between the local memory and the global memory is called the granularity of management. Heap data can be managed at various granularities, right from word-level to the whole heap space allocated in the local memory. Again, we can look at the sample code in Figure 2. The program accesses one field (item.id) of the data structure after initialization. When the program accesses any part of a allocated data structure, if only the exact field (item.id) is brought into the local memory, the heap management is done at word level of granularity. If the whole data structure is brought into the local memory, the heap management is done at programmer-defined granularity. How to define the granularity depends on the structure of the application. If the allocated data structures are very large and only a small part of them are used each time, we believe that a finer granularity of heap management is beneficial. When the allocated objects are very small, heap management can perform at a coarser granularity by grouping the allocated objects into a block, and if a part of any of them is accessed, a whole block of them are brought into the local memory. One important advantage of our software implemented heap management is that it can be tuned to the application demands, rather than block size being fixed for a given processor implementation in traditional cache architectures.

6.5. Local Heap Management

In our heap management, the functions *_p2s* and *_s2p* look up the management table at every function call. Since this looking up overhead can be large, one is tempted to maintain the whole table in the local memory. However, the size of the table may also grow arbitrarily large if the number of heap objects is large. Therefore, heap management approach should support the requirement of maintaining a portion of management table in the local memory. The space we defined in the local memory for heap S is divided into a constant space H for heap data, and a constant space T for heap management table, such that $S = H + T$. All the sizes, S, H, and T can be fixed at compile time. Every time when *malloc* wants to add a new entry, it checks if there is still place. If yes, it can just write the new entry; otherwise it can only write the new entry after making its space by evicting some of the older entries to the global memory. The global memory management thread described in the previous section provides space in the global memory for older entries. Besides, we need to consider the granularity when we evict the entries. The heap table management can also be performed at several granularities, from a single entry to the entire table size we set at the compile time. We leave the exploration of the effect of the granularity of table management as a future concern. In this work, we manage the heap management table at the whole table size granularity. Namely, we evict the whole table, and bring a full table back into the local memory, when needed.

7. OPTIMIZATION FOR EMBEDDED SYSTEMS

In order to support (almost) unlimited heap data, we implement a thread in the main core to dynamically allocate memory for heap data and the heap management table. Fundamentally this requires some communications between execution cores and the main core, which can interpret messages from local threads. In the Cell BE, we can achieve this through another thread on the main processor and a mailbox-based communication between execution cores and the main core. This communication overhead is in addition to the actual heap data transfers. Clearly this has high overhead and will become more expensive as the number of cores increases.

In embedded software, where the upper bound of the heap size can be profiled, we can do some optimizations to minimize the overhead. By profiling, we can get and keep this maximum size. Then we can define static data structures, for example, arrays, to contiguously accommodate heap data and heap management table entries from local cores. When heap data is needed, we can resolve the global address in the execution core so that a DMA can be directly used to transfer the data from the global memory. This completely eliminates the need for the extra thread in the main core, and therefore avoids all the performance overheads associated with the communication. In addition, if possible, and especially because the heap management table entries may be much smaller than heap data, the whole heap management table may be housed in the local memory, resulting in additional performance optimization.

8. EXPERIMENTS

8.1. Experimental Setup

We conduct our experiments on the IBM Cell BE in Sony Playstation 3, which runs a Linux Fedora 9[1] and gives us access to six of eight SPEs. We implement our scheme on the single-threaded benchmarks from the Mibench suite [Guthaus et al. 2001] and self-implement other applications which use heap variables. We modify those single-threaded benchmarks to multithreaded applications, in which the PPE thread performs all the input/output and the SPE threads perform all computing tasks. We evaluate the effectiveness of our heap management technique and optimization by comparing the runtime of (i) benchmarks without any heap management, (ii) benchmarks with heap management to support arbitrary heap data size, and (iii) benchmarks with heap management optimizations. We use *_mftb()* and *spu_decrementer()* for measuring the runtime of PPE and SPE individually. The details of our benchmarks are listed in Table I. *dijkstra, fft, fft_inv*, and *stringsearch* are from the Mibench suite [Guthaus

[1]http://fedoraproject.org/wiki/releases/9.

Table I. Several Benchmarks from MiBench and Elsewhere that Use Heap Variables

Benchmarks	Source	Description	Heap Size (bytes)
Dijkstra	Mibench	find the shortest path	5,040
fft	Mibench	fft algorithm	16,416
fft_inv	Mibench	fft_inv algorithm	16,416
stringsearch	Mibench	search strings	4,096
DFS	self-implemented	depth first search algorithm	16,000
MST	self-implemented	minimum spanning tree algorithm	336
rbTree	self-implemented	red black tree data structure	2,476

Note: The table shows the maximum heap data each application needs.

et al. 2001], while *DFS*, *MST*, and *red black tree* are some other algorithms that are very likely to be used in the application domain developed for the Cell BE. Only the maximum heap size demands for all the benchmarks are noted in Table I, the size of other data is skipped. Although the heap data can fit in the 256 KB of the local memory, the whole application can not be accommodated in it. Most of our experiments are conducted on the configuration of one PPE and only one SPE. The scalability of our technique is explored in our last experiment where multiple identical threads are created on various number of cores.

8.2. Unrestricted Heap Size

To demonstrate the value of our heap management, we execute one benchmark, *rbTree*, with and without heap management. It is a binary search benchmark. Each node in the tree data structure is 24 bytes large and is dynamically allocated. In the benchmark, 241 KB can be shared by the heap data and stack data, and the remaining 15 KB are occupied by the code and global data. We can allocate only $n_0 = 6{,}800$ nodes (almost 160 KB in heap) without any heap management, exceeding which the program crashes. We run the benchmark using our heap management scheme with nodes from 1 to 65,536, which is almost ten times larger than that which can be executed without heap management. We initially allocate 150 KB for heap data in the local memory. Therefore, no heap data DMA happens between the global memory and the local memory until the 150 KB space is full. Furthermore, our heap management table consumes 4 KB, which means we have 256 entries in our table. We choose these parameters for the fair comparison of time with and without heap management scheme.

The first observation from Figure 8 is that our technique seems to support any heap size of the application. We dynamically manage both the heap management table and the memory allocation in the global memory. The runtime increases as the number of nodes in *rbTree* becomes larger, for the reason that DMA needs to be performed in our heap management scheme for heap data and heap management table. It is also the reason why there is a leap after allocating more than 6,800 nodes. However, our technique enables the execution of the application for any program parameters, without any further modifications.

8.3. Impact of Heap Management Parameters

With the heap management, the performance of applications are most affected by two parameters: (i) the amount of memory in the execution core that is used for storing heap objects and the heap management table; (ii) the granularity that we choose for heap data management.

The total space S we defined in the local memory for heap can be partitioned as $S = H + T$. Assume the number of heap chunks that can be located in the fixed heap region is n_H and the size of each heap chunk is s_H, then the total space for heap data

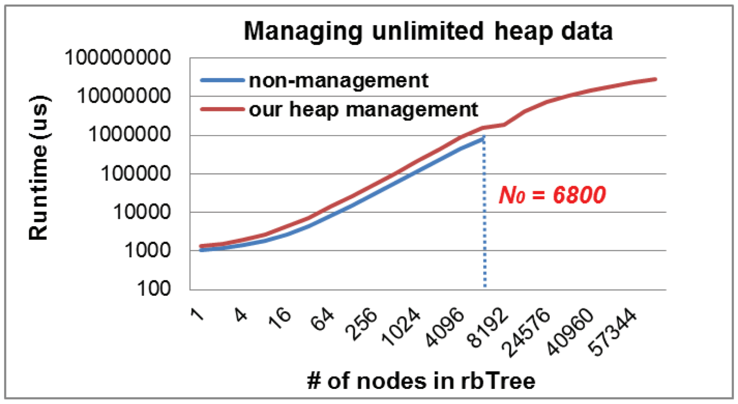

Fig. 8. Without heap management, the program will run with at most 6,800 nodes; with management, we get a flexibility of using applications that require larger heap.

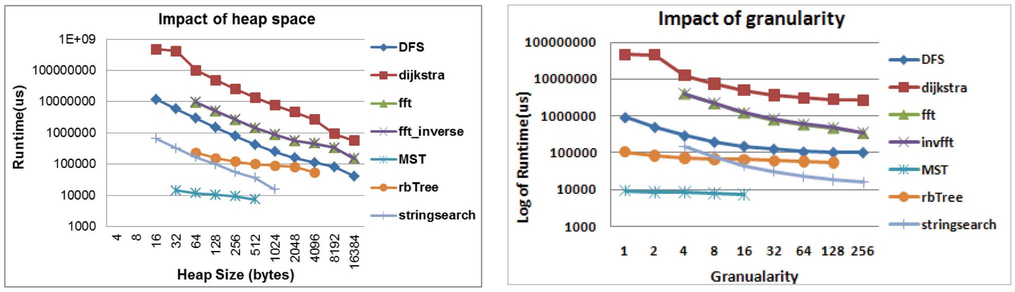

Fig. 9. (a) Heap management with dynamic memory allocation in *global memory* of PPE for heap objects from *local memory* of SPE; the heap region size for each benchmark is set from the minimum size to the maximum size. (b) Benchmarks with different granularities. Granularity defines the number of heap objects we consider for DMA transfers at a time.

is $H = n_H * s_H$. Also, T can be interpreted as $T = n_E * s_E$, where n_E is the number of entries in the local memory and s_E is the size of one entry. For a given S, there are several schemes to partition S into H and T. How to partition the memory is our future concern, and therefore here we keep a unique partition of the total space as $n_H = n_E$. We get the minimum and the maximum heap size that is required for each benchmark by profiling and run the application for the entire range of heap sizes. The runtime of applications are shown in Figure 9. The most clear observation from the graph is that the performance improves as we allocate more memory for heap data.

The second main factor that affects the performance of our heap management is the granularity of communication between the local memory and the global memory. We bind several memory blocks that are malloc-ed as one granularity. To get rid of the influence of other factors and discover the single factor (granularity), we set the size of heap region in the local memory as 4 KB. From Figure 9(b), we can see the effect of the granularity is that the runtime decreases as we increase the granularity. There are two main reasons. On the one hand, the bandwidth of the interconnected network of the IBM Cell BE is big. The data transfer time between the global memory and the local memory will not increase too much as we increase DMA transfer sizes. Therefore, it is beneficial to have coarser granularity. One the other hand, the latency for DMA is not small. Coarser granularity can decrease the number of DMA calls, which in turn decrease the performance penalties caused by DMAs.

Table II. Overheads Caused by Calling Heap Management Functions

Benchmarks	Overhead(us) Caused by Calling				Fraction of total time
	malloc	free	p2s	s2p	
Dijkstra	6,232	76	289,937	234,758	24%
fft	244	1	50,579	21,891	23%
fft_inv	514	1	102,589	44,421	45%
stringsearch	131	2	902	669	11%
DFS	381	0	11,339	4,057	17%
MST	391	0	96	77	7%
rbTree	155	0	1,071	970	4%

Note: For all benchmarks, we set 8,192 bytes for heap region, 256 for heap data granularity, and the whole heap management table is in the local memory of each core.

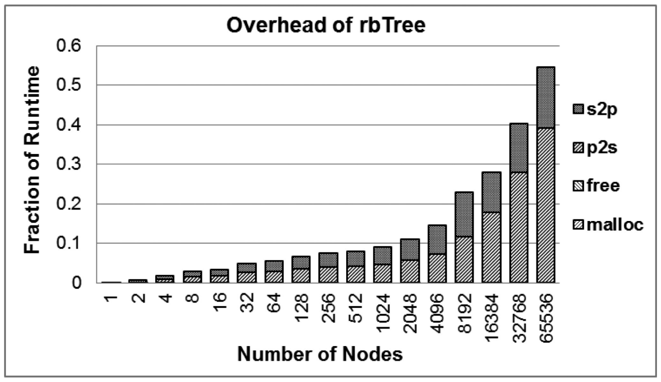

Fig. 10. The overhead varies from 1% to 55% when the input size of *rbTree* changes from 1 to 65,536 nodes, because the number of library function calls increases as heap objects increase.

8.4. Overhead of Heap Management

Our heap management can enable the programs that can not execute on limited local memory (LLM) multicore architectures. However, heap management causes overheads inevitably. Lots of factors can affect the application performance: Parameters, such as the region sizes defined in the local memory for heap data and heap management table, granularity used for heap data and the heap management table, and memory distribution between heap data and the heap management table, so on and so forth. The frequency of heap variable accesses in the application is another important effect for the performance penalty.

As shown in Table II, the average fraction of heap management overhead for all benchmarks is 18%. In this experiment, we set 8,192 bytes for heap data, 256 as the granularity for heap data, and the whole management table located in the local memory. The overhead of *fft_inv* benchmark is very high, which is about 45% of the total execution time. This is because there are few operations other than accessing the heap variables. For benchmarks *MST* and *rbTree*, the overheads caused by heap management are very small, since the heap size 8,192 bytes we set for heap data is larger than their memory demands.

To delve deeper into high overhead benchmarks, Figure 10 shows the variation of the overhead as we increase the data size of the red black tree *rbtree*. The horizontal axis is the number of nodes in *rbtree* and is equal to the number of heap objects created in the application. *rbtree* has several accesses to heap variables, and therefore makes

Table III. Average Performance Enhancement for Each Benchmark with Different Heap Region Sizes and Different Granularities is 14%

Benchmarks	Dynamic Management (us)	Static Management (us)	Average Improvement
Dijkstra	48,167,280	39,317,728	18%
fft	2,837,261	2,708,675	5%
fft_inv	2,923,532	2,788,252	5%
stringsearch	130,029	127,650	2%
DFS	1,000,519	957,732	4%
MST	9,909	5,615	43%
rbTree	98,044	77,376	21%

Note: The reason being that if a static buffer can be assigned in the memory for heap management, it eliminates the blocking DMA calls due to the global memory management thread in the PPE.

a lot of heap management function calls, especially *_p2s* and *_s2p*. In a sense, this is an extreme benchmark that exercises our heap management technique to its extreme. Figure 10 shows that the data management overhead comprises of 1% to 55% of the runtime as the input size (N) changes.

Figure 10 also shows the contribution of each data management function in the overhead. High penalties are caused mostly due to the functions *_p2s* and *_s2p* that are added before and after every heap variable access, because when an application has heap, heap accesses are very high. Consequently, lots of DMA-based data communications between the global memory and the local memory are needed, which increases the expense of managing data.

8.5. Optimization for Embedded Applications

For extremely embedded applications where we can get the maximum heap size of the application by profiling, we can improve the application performance by initially allocating a static buffer in the global memory. By doing this, the management will not incur any additional penalties, for example, communication overhead caused by the use of mailbox between the SPE and the PPE. If there is residual space in the local memory, it is also beneficial to get that whole memory for our heap objects and heap management table. In addition, increasing the data transfer granularity also helps a lot. We conduct these optimization techniques on every benchmark and show the un-optimized and optimized runtime in Table III. From Table III, we see that we can obtain performance improvement by an average of 14% for all benchmarks.

8.6. Scalability Experiment

To illustrate the scalability of our technique, we execute the same application on different number of cores. In addition, we set the least heap region size in the local memory for heap variables and the smallest granularity with dynamic memory allocation. We believe this is the most aggressive configuration which has worst performance to test our technique.

In Figure 11, we observe that the runtime increases gradually as we scale the number of cores. This is mostly caused by the competition of DMA requests and the uses of mailbox from different cores. The increases are larger for benchmarks *rbTree* and *MST*, since they have less frequent localized read/write operations than the rest of benchmarks have. The scattered heap access leads to frequent access to the global memory, and increases the latency to serve each access when the number of cores increases.

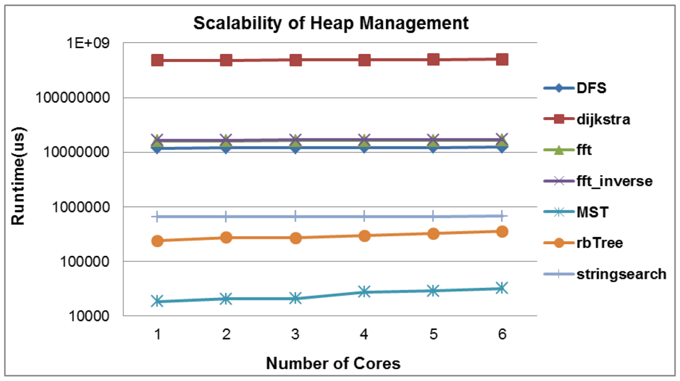

Fig. 11. Benchmarks with dynamic memory allocation in the global memory of PPE, configuring with different number of cores.

9. SUMMARY AND FUTURE WORK

With the increasing number of cores, how to scale the memory architecture becomes a great challenge. Limited local memory (LLM) multicore architectures become one option to meet this needs. They are scalable memory architectures that are popular in modern and futuristic embedded processors, for example, the IBM Cell BE. Such architectures have a local memory in each core which is under software control. If all the application code and data can fit into the limited local memory of the core, the efficiency of execution is achieved; however it is not always the case, then the application code and data must be managed between the local memory and the global memory by explicitly inserting DMA commands in the application. While management is needed for all data, it is extremely challenging and important to manage heap data since it is dynamic in nature and can therefore easily overwrite stack data and generate unexpected results. One possibility of managing heap data is through the use of the software cache; however, it requires programmers to modify the thread code as well as the main thread code, which can be not only counterintuitive and laborious, but also error-prone. In this article, we present a framework to automatically manage heap data by providing a simple and intuitive programming interface, which is composed of two re-implemented function *malloc* and *free* and two new functions *_p2s* and *_s2p* that have to be inserted before and after each heap variable access individually. Our experiments on the IBM Cell BE demonstrate that (i) our scheme is intuitive and easy to use; (ii) it can support almost any amount of heap data; and (iii) it scales well with different number of cores. In addition, the single global memory management thread on the main core we provided in our library can serve all the memory requests from execution cores. Finally, we propose some optimizations for our heap management in extremely embedded systems, which can further improve the runtime of applications by an average of 14%.

Our technique can also gain improvement in the following directions. First, the number of table entries is the same as the number of heap objects in the local memory. In fact, given a total space in the local memory for heap, including heap variables and heap management table entries, we can partition it in different ways to minimize the total DMA transfer time. Second, we can reduce the number of calls to *_p2s* and *_s2p* functions before/after each heap variable access by predicting if the variable will access this heap data again at a later stage. This can be achieved by analyzing the data flow graph and control flow graph. Finally, we can further optimize our heap management by using prefetching and double buffering techniques, since they can overlap DMA transfers with the task of data computing.

As multicore architectures with limited local memories become a trend, more and more not-so-embedded applications are being and will be developed. In order to improve development productivity, we are affronting the need and challenge of executing applications on such architectures without too many changes of the natural way of programming. We hope our work inspires more good work to meet this new challenge.

REFERENCES

ANGIOLINI, F., MENICHELLI, F., FERRERO, A., BENINI, L., AND OLIVIERI, M. 2004. A post-compiler approach to scratchpad mapping of code. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. ACM, New York, NY, 259–267.

AVISSAR, O., BARUA, R., AND STEWART, D. 2002. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Trans. Embed. Comput. Sys. 1,* 1, 6–26.

BAI, K., LU, D., AND SHRIVASTAVA, A. 2011a. Vector class on limited local memory (LLM) multi-core processors. In *Proceedings of the 14th International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. 215–224.

BAI, K., SHRIVASTAVA, A., AND KUDCHADKER, S. 2011b. Stack data management for limited local memory (LLM) multi-core processors. In *Proceedings of the International Conference on Application Specific Systems, Architectures and Processors*. 231–234.

BANAKAR, R., STEINKE, S., LEE, B.-S., BALAKRISHNAN, M., AND MARWEDEL, P. 2002. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign*. ACM, New York, NY, 73–78.

CHE, W. AND CHATHA, K. 2011a. Compilation of stream programs onto scratchpad memory based embedded multicore processors through retiming. In *Proceedings of the 48th Design Automation Conference*. ACM, New York, NY, 122–127.

CHE, W. AND CHATHA, K. 2011b. Scheduling of stream programs onto spm enhanced processors with code overlay. In *Proceedings of the 9th IEEE/ACM Symposium on Embedded Systems and Real-Time Multimedia*.

CHE, W. AND CHATHA, K. S. 2010. Scheduling of synchronous data flow models on scratchpad memory based embedded processors. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. 205–212.

CHE, W., PANDA, A., AND CHATHA, K. S. 2010. Compilation of stream programs for multicore processors that incorporate scratchpad memories. In *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, Belgium, 1118–1123.

DOMINGUEZ, A., UDAYAKUMARAN, S., AND BARUA, R. 2005. Heap data allocation to scratch-pad memory in embedded systems. *Embed. Comput. 1,* 4, 521–540.

EGGER, B., KIM, C., JANG, C., NAM, Y., LEE, J., AND MIN, S. L. 2006a. A dynamic code placement technique for scratchpad memory using postpass optimization. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. ACM, New York, NY, 223–233.

EGGER, B., LEE, J., AND SHIN, H. 2006b. Scratchpad memory management for portable systems with a memory management unit. In *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*. ACM, New York, NY, 321–330.

EICHENBERGER, A., O'BRIEN, J. K., O'BRIEN, K. M., WU, P., CHEN, T., ODEN, P. H., PRENER, D. A., SHEPARD, J. C., SO, B., SURA, Z., WANG, A., ZHANG, T., ZHAO, P., GSCHWIND, M. K., ARCHAMBAULT, R., GAO, Y., AND KOO, R. 2006. Using advanced compiler technology to exploit the performance of the cell broadband engineTM architecture. *IBM Syst. J. 45,* 1, 59–84.

FLACHS, B., ASANO, S., DHONG, S., HOFSTEE, H., GERVAIS, G., KIM, R., LE, T., LIU, P., LEENSTRA, J., LIBERTY, J., MICHAEL, B., OH, H.-J., MUELLER, S., TAKAHASHI, O., HATAKEYAMA, A., WATANABE, Y., YANO, N., BROKENSHIRE, D., PEYRAVIAN, M., TO, V., AND IWATA, E. 2006. The microarchitecture of the synergistic processor for a cell processor. *IEEE Solid-State Circuits 41,* 1, 63–70.

FRANCESCO, P., MARCHAL, P., ATIENZA, D., BENINI, L., CATTHOOR, F., AND MENDIAS, J. M. 2004. An integrated hardware/software approach for run-time scratchpad management. In *Proceedings of the 41st Annual Design Automation Conference*. ACM, New York, NY, 238–243.

GUTHAUS, M., RINGENBERG, J., ERNST, D., AUSTIN, T., MUDGE, T., AND BROWN, R. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE International Workshop on Workload Characterization.* 3–14.

Janapsatya, A., Ignjatović, A., and Parameswaran, S. 2006. A novel instruction scratchpad memory optimization method based on concomitance metric. In *Proceedings of the Conference on Asia South Pacific Design Automation*. IEEE Press, Piscataway, NJ, 612–617.

Jung, S. C., Shrivastava, A., and Bai, K. 2010. Dynamic code mapping for limited local memory systems. In *Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors*. 13–20.

Kandemir, M., Ramanujam, J., and Choudhary, A. 2002. Exploiting shared scratch pad memory space in embedded multiprocessor systems. In *Proceedings of the 39th Annual Design Automation Conference*. ACM, New York, NY, 219–224.

Kandemir, M., Ramanujam, J., Irwin, J., Vijaykrishnan, N., Kadayif, I., and Parikh, A. 2001. Dynamic management of scratch-pad memory space. In *Proceedings of the 38th Annual Design Automation Conference*. ACM, New York, NY, 690–695.

Kannan, A., Shrivastava, A., Pabalkar, A., and Lee, J.-E. 2009. A software solution for dynamic stack management on scratch pad memory. In *Proceedings of the Asia and South Pacific Design Automation Conference*. IEEE Press, Piscataway, NJ, 612–617.

Li, L., Gao, L., and Xue, J. 2005. Memory coloring: A compiler approach for scratchpad memory management. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, Washington, DC, 329–338.

McIlroy, R., Dickman, P., and Sventek, J. 2008. Efficient dynamic heap allocation of scratch-pad memory. In *Proceedings of the 7th International Symposium on Memory Management*. ACM Press, New York, NY, 31–40.

Montanaro, J., Witek, R. T., Anne, K., Black, A. J., Cooper, E. M., Dobberpuhl, D. W., Donahue, P. M., Eno, J., Hoeppner, G. W., Kruckemyer, D., Lee, T. H., Lin, P. C. M., Madden, L., Murray, D., Pearce, M. H., Santhanam, S., Snyder, K. J., Stephany, R., and Thierauf, S. C. 1997. A 160-mhz, 32-b, 0.5-w CMOS RISC microprocessor. *Digital Tech. J. 9,* 1, 49–62.

Nguyen, N., Dominguez, A., and Barua, R. 2005. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. ACM, New York, NY, 115–125.

Pabalkar, A., Shrivastava, A., Kannan, A., and Lee, J. 2008. SDRM: Simultaneous determination of regions and function-to-region mapping for scratchpad memories. In *Proceedings of the International Conference on High Performance Computing (HiPC)*.

Steinke, S., Grunwald, N., Wehmeyer, L., Banakar, R., Balakrishnan, M., and Marwedel, P. 2002a. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *Proceedings of the 15th International Symposium on System Synthesis*. ACM, New York, NY, 213–218.

Steinke, S., Wehmeyer, L., Lee, B., and Marwedel, P. 2002b. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of the Conference on Design, Automation and Test in Europe*. IEEE Computer Society, Los Alamitos, CA, 409.

Udayakumaran, S., Dominguez, A., and Barua, R. 2006. Dynamic allocation for scratch-pad memory using compile-time decisions. *Trans. Embed. Comput. Sys. 5,* 2, 472–511.

Verma, M. and Marwedel, P. Aug. 2006. Overlay techniques for scratchpad memories in low power embedded processors. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 14,* 8, 802–815.

Verma, M., Petzold, K., Wehmeyer, L., Falk, H., and Marwedel, P. 2005. Scratchpad sharing strategies for multiprocess embedded systems: A first approach. In *Proceedings of the 3rd Workshop on Embedded Systems for Real-Time Multimedia (ESTImedia)*. 115–120.

Verma, M., Wehmeyer, L., and Marwedel, P. 2004. Cache-aware scratchpad allocation algorithm. In *Proceedings of the Conference on Design, Automation and Test in Europe*. Vol. 2. IEEE Computer Society, Washington, DC, 21264.

Received November 2010; revised July 2011; accepted January 2012